IBM Db2 V11.5

SQL Reference

2023-02-07

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

© Copyright IBM Corp. 2016, 2023

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:
© (your company name) (year).
Portions of this code are derived from IBM Corp. Sample Programs.
© Copyright IBM Corp. _enter the year or years_.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Contents

Tables

Chapter 1. Structured Query Language (SQL)

SQL is a standardized language for defining and manipulating data in a relational database.

In accordance with the relational model of data, the database is treated as a set of tables, relationships are represented by values in tables, and data is retrieved by specifying a result table that can be derived from one or more base tables.

SQL statements are executed by a database manager. One of the functions of the database manager is to transform the specification of a result table into a sequence of internal operations that optimize data retrieval. The transformation occurs in two phases: preparation and binding.

All executable SQL statements must be prepared before they can be executed. The result of preparation is the executable or operational form of the statement. The method of preparing an SQL statement and the persistence of its operational form distinguish static SQL from dynamic SQL.

How to read the syntax diagrams

This topic describes the structure of SQL syntax diagrams.

Read the syntax diagrams from left to right and top to bottom, following the path of the line.

The double right arrowhead and line symbol ►►— indicates the beginning of a syntax diagram.

The line and single right arrowhead symbol —► indicates that the syntax is continued on the next line.

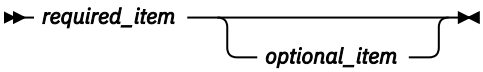
The right arrowhead and line symbol ►— indicates that the syntax is continued from the previous line.

The line, right arrowhead, and left arrowhead symbol —►◄ symbol indicates the end of a syntax diagram.

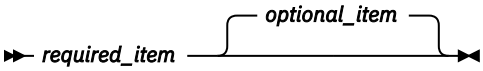
Syntax fragments start with the pipe and line symbol |— and end with the —| line and pipe symbol.

Required items appear on the horizontal line (the main path).

Optional items appear below the main path.

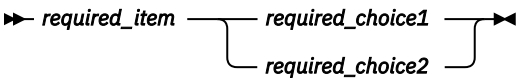


If an optional item appears above the main path, that item has no effect on execution, and is used only for readability.



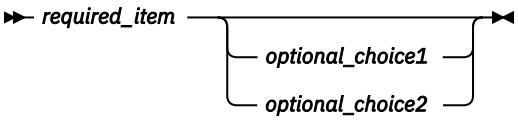
If you can choose from two or more items, they appear in a stack.

If you *must* choose one of the items, one item of the stack appears on the main path.

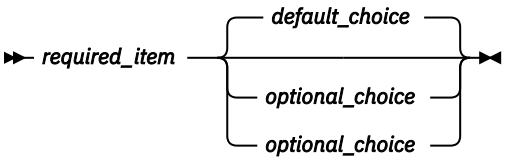


If choosing one of the items is optional, the entire stack appears below the main path.

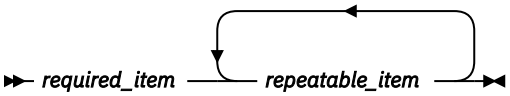
© Copyright IBM Corp. 2016, 2023



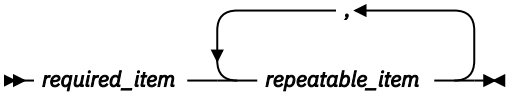
If one of the items is the default, it will appear above the main path, and the remaining choices will be shown below.



An arrow returning to the left, above the main line, indicates an item that can be repeated. In this case, repeated items must be separated by one or more blanks.



If the repeat arrow contains a comma, you must separate repeated items with a comma.

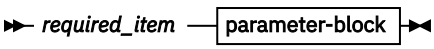


A repeat arrow above a stack indicates that you can make more than one choice from the stacked items or repeat a single choice.

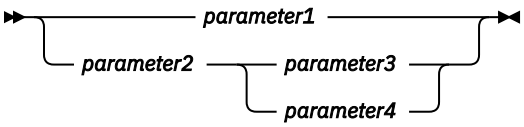
Keywords appear in uppercase (for example, FROM). They must be spelled exactly as shown. Variables appear in lowercase (for example, `column-name`). They represent user-supplied names or values in the syntax.

If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Sometimes a single variable represents a larger fragment of the syntax. For example, in the following diagram, the variable `parameter-block` represents the whole syntax fragment that is labeled **parameter-block**:



parameter-block



Adjacent segments occurring between "large bullets" (●) may be specified in any sequence.

The above diagram shows that item2 and item3 may be specified in either order. Both of the following are valid:

```
required_item item1 item2 item3 item4
required_item item1 item3 item2 item4
```

Conventions used for the SQL topics

Error conditions

An error condition is indicated within the text of the manual by listing the SQLSTATE associated with the error in parentheses.

For example:

```
A duplicate signature returns an SQL error (SQLSTATE 42723).
```

Highlighting conventions

This topic covers the conventions used in the SQL Reference.

- **Bold** indicates commands, keywords, and other items whose names are predefined by the system.
- *Italics* indicates one of the following items:
  - Names or values (variables) that must be supplied by the user
  - General emphasis
  - The introduction of a new term
  - A reference to another source of information

Conventions describing Unicode data

When a specific Unicode code point is referenced, it is expressed as U+*n* where *n* is four to six hexadecimal digits, using the digits 0-9 and uppercase letters A-F.

Leading zeros are omitted unless the code point would have fewer than four hexadecimal digits. The space character, for example, is expressed as U+0020. In most cases, the *n* value is the same as the UTF-16BE encoding.

Language elements

Characters

The basic symbols of keywords and operators in the SQL language are single-byte characters that are part of all IBM® character sets. Characters of the language are classified as letters, digits, or special characters.

A *letter* is any of the 26 uppercase (A through Z) or 26 lowercase (a through z) letters. Letters also include three code points reserved as alphabetic extenders for national languages (#, @, and $ in the United States). However these three code points should be avoided, especially for portable applications, because they represent different characters depending on the CCSID. Letters also include the alphabetics from the extended character sets. Extended character sets contain additional alphabetic characters; for example, those with diacritical marks (´ is an example of a diacritical mark). The available characters depend on the code page in use.

A *digit* is any of the characters 0 through 9.

A *special character* is any of the characters listed in the following table:

Table 1. Special characters

Character	Description	Character	Description
	space or blank	-	minus sign
"	quotation mark or double quote or double quotation mark	.	period

Table 1. Special characters (continued)

Character	Description	Character	Description
%	percent	/	slash
&	ampersand	:	colon
'	apostrophe or single quote or single quotation mark	;	semicolon
(	left parenthesis	<	less than
)	right parenthesis	=	equals
*	asterisk	>	greater than
+	plus sign	?	question mark
,	comma	_	underline or underscore
|	vertical bar[1]	^	caret
!	exclamation mark	[	left bracket
{	left brace	]	right bracket
}	right brace	\	reverse solidus or back slash[2]

[1] Using the vertical bar (|) character might inhibit code portability between IBM relational products. Use the CONCAT operator in place of the || operator.

[2] Some code pages do not have a code point for the reverse solidus (\) character. When entering Unicode string constants, the UESCAPE clause can be used to specify a Unicode escape character other than reverse solidus.

All multi-byte characters are treated as letters, except for the double-byte blank, which is a special character.

Tokens

Tokens are the basic syntactical units of SQL. A *token* is a sequence of one or more characters.

A token cannot contain blank characters, unless it is a string constant or a delimited identifier, which may contain blanks.

Tokens are classified as ordinary or delimiter:

- An *ordinary token* is a numeric constant, an ordinary identifier, a host identifier, or a keyword.

  Examples

  ```
  1       .1       +2       SELECT       E       3
  ```

- A *delimiter token* is a string constant, a delimited identifier, an operator symbol, or any of the special characters shown in the syntax diagrams. A question mark is also a delimiter token when it serves as a parameter marker.

  Examples

  ```
  ,       'string'       "fld1"       =       .
  ```

Spaces: A space is a sequence of one or more blank characters. Tokens other than string constants and delimited identifiers must not include a space. Any token may be followed by a space. Every ordinary token must be followed by a space or a delimiter token if allowed by the syntax.

Comments: SQL comments are either bracketed (introduced by /* and end with */) or simple (introduced by two consecutive hyphens and end with the end of line). Static SQL statements can include host language comments or SQL comments. Comments can be specified wherever a space can be specified, except within a delimiter token or between the keywords EXEC and SQL.

Case sensitivity: Any token may include lowercase letters, but a lowercase letter in an ordinary token is folded to uppercase, except for host variables in the C language, which has case-sensitive identifiers. Delimiter tokens are never folded to uppercase. Thus, the statement:

```
select * from EMPLOYEE where lastname = 'Smith';
```

is equivalent, after folding, to:

```
SELECT * FROM EMPLOYEE WHERE LASTNAME = 'Smith';
```

Multi-byte alphabetic letters are not folded to uppercase. Single-byte characters (a to z) *are* folded to uppercase.

For characters in Unicode:

- A character is folded to uppercase, if applicable, if the uppercase character in UTF-8 has the same length as the lowercase character in UTF-8. For example, the Turkish lowercase dotless 'i' is not folded, because in UTF-8, that character has the value X'C4B1', but the uppercase dotless 'I' has the value X'49'.
- The folding is done in a locale-insensitive manner. For example, the Turkish lowercase dotted 'i' is folded to the English uppercase (dotless) 'I'.
- Both halfwidth and fullwidth alphabetic letters are folded to uppercase. For example, the fullwidth lowercase 'a' (U+FF41) is folded to the fullwidth uppercase 'A' (U+FF21).

Identifiers

An *identifier* is a token that is used to form a name. An identifier in an SQL statement is either an SQL identifier or a host identifier.

- SQL identifiers

  There are two types of SQL identifiers: ordinary and delimited.

  – An ordinary identifier is an uppercase letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character. Note that lowercase letters can be used when specifying an ordinary identifier, but they are converted to uppercase when processed. An ordinary identifier should not be a reserved word.

    The following example shows ordinary identifiers:

    ```
    WKLYSAL     WKLY_SAL
    ```

  – A delimited identifier is a sequence of one or more characters enclosed by double quotation marks. Leading blanks in the sequence are significant. Trailing blanks in the sequence are not significant, although they are stored with the identifier. Two consecutive quotation marks are used to represent one quotation mark within the delimited identifier. A delimited identifier can be used when the sequence of characters does not qualify as an ordinary identifier. In this way an identifier can include lowercase letters.

    The following example shows a series of delimited identifiers:

    ```
    "WKLY_SAL"     "WKLY SAL"     "UNION"     "wkly_sal"
    ```

  Character conversion of identifiers created on a double-byte code page, but used by an application or database on a multi-byte code page, may require special consideration: After conversion, such identifiers may exceed the length limit for an identifier.

- Host identifiers

A host identifier is a name declared in the host program. The rules for forming a host identifier are the rules of the host language. A host identifier should not be greater than 255 bytes in length and should not begin with SQL or DB2 (in uppercase or lowercase characters).

Naming conventions and implicit object name qualifications

The rules for forming a database object name depend on the type of the object designated by the name. A name may consist of a single SQL identifier or it may be qualified with one or more identifiers that more specifically identify the object. A period must separate each identifier.

The syntax diagrams use different terms for different types of names. The following list defines these terms.

alias-name
A schema-qualified name that designates an alias.

attribute-name
An identifier that designates an attribute of a structured data type.

array-type-name
A qualified or unqualified name that designates a user-defined array type. The unqualified form of array-type-name is an SQL identifier. An unqualified array type name in an SQL statement is implicitly qualified. The implicit qualifier is a schema name or a module name, which is determined by the context in which array-type-name appears. The qualified form is a schema-name followed by a period and an SQL identifier or a module-name (which can also be qualified by a schema-name) followed by a period and an SQL identifier. If the array type is defined in a module and used outside of the same module, it must be qualified by the module-name.

authorization-name
An identifier that designates a user, group, or role. For a user or a group:

- Valid characters are: 'A' through 'Z'; 'a' through 'z'; '0' through '9'; '#'; '@'; '$'; '_'; '!'; ' '(';')'; '{'; '}'; '-'; '.'; and '^'.
- The following characters must be delimited with quotation marks when entered through the command line processor: '!'; ' '(';')'; '{'; '}'; '-'; '.'; and '^'.
- The name must not begin with the characters 'SYS', 'IBM', or 'SQL'.
- The name must not be: 'ADMINS', 'GUESTS', 'LOCAL', 'PUBLIC', or 'USERS'.
- A delimited authorization ID must not contain lowercase letters.

bufferpool-name
An identifier that designates a buffer pool.

column-name
A qualified or unqualified name that designates a column of a table or view. The qualifier is a table name, a view name, a nickname, or a correlation name.

component-name
An identifier that designates a security label component.

condition-name
A qualified or unqualified name that designates a condition. An unqualified condition name in an SQL statement is implicitly qualified, depending on its context. If the condition is defined in a module and used outside of the same module, it must be qualified by the module-name.

constraint-name
An identifier that designates a referential constraint, primary key constraint, unique constraint, or a table check constraint.

correlation-name
An identifier that designates a result table.

cursor-name
An identifier that designates an SQL cursor. For host compatibility, a hyphen character may be used in the name.

cursor-type-name
A qualified or unqualified name that designates a user-defined cursor type. The unqualified form of cursor-type-name is an SQL identifier. An unqualified cursor-type-name in an SQL statement is implicitly qualified, depending on context. The implicit qualifier is a schema name or a module name, which is determined by the context in which cursor-type-name appears. The qualified form is a schema-name followed by a period and an SQL identifier or a module-name (which can also be qualified by a schema-name) followed by a period and an SQL identifier. If the cursor type is defined in a module and used outside of the same module, it must be qualified by the module-name.

cursor-variable-name
A qualified or unqualified name that designates a global variable, local variable or an SQL parameter of a cursor type. An unqualified cursor variable name in an SQL statement is implicitly qualified, depending on context.

data-source-name
An identifier that designates a data source. This identifier is the first part of a three-part remote object name.

db-partition-group-name
An identifier that designates a database partition group.

descriptor-name
A colon followed by a host identifier that designates an SQL descriptor area (SQLDA). For the description of a host identifier, see "References to host variables" on page 20. Note that a descriptor name never includes an indicator variable.

distinct-type-name
A qualified or unqualified name that designates a distinct type. The unqualified form of distinct-type-name is an SQL identifier. An unqualified distinct type name in an SQL statement is implicitly qualified. The implicit qualifier is a schema name or a module name, which is determined by the context in which distinct-type-name appears. The qualified form is a schema-name followed by a period and an SQL identifier or a module-name (which can also be qualified by a schema-name) followed by a period and an SQL identifier. If the distinct type is defined in a module and used outside of the same module, it must be qualified by the module-name.

event-monitor-name
An identifier that designates an event monitor.

function-mapping-name
An identifier that designates a function mapping.

function-name
A qualified or unqualified name that designates a function. The unqualified form of function-name is an SQL identifier. An unqualified function name in an SQL statement is implicitly qualified. The implicit qualifier is a schema name, which is determined by the context in which the function appears. The qualified form could be is a schema-name followed by a period and an SQL identifier or a module-name followed by a period and an SQL identifier. If the function is published in a module and used outside of the same module, it must be qualified by the module-name.

global-variable-name
A qualified or unqualified name that designates a global variable. The unqualified form of global-variable-name is an SQL identifier. An unqualified global variable name in an SQL statement is implicitly qualified. The implicit qualifier is a schema name or a module name, which is determined by the context in which global-variable-name appears. The qualified form is a schema-name followed by a period and an SQL identifier or a module-name (which can also be qualified by a schema-name) followed by a period and an SQL identifier. If the global variable is defined in a module and used outside of the same module, it must be qualified by the module-name.

group-name
An unqualified identifier that designates a transform group defined for a structured type.

host-variable
A sequence of tokens that designates a host variable. A host variable includes at least one host identifier, explained in "References to host variables" on page 20.

index-name
A schema-qualified name that designates an index or an index specification.

label
An identifier that designates a label in an SQL procedure.

method-name
An identifier that designates a method. The schema context for a method is determined by the schema of the subject type (or a supertype of the subject type) of the method.

module-name
A qualified or unqualified name that designates a module. An unqualified module-name in an SQL statement is implicitly qualified. The implicit qualifier is a schema name, which is determined by the context in which the module-name appears. The qualified form is a schema-name followed by a period and an SQL identifier.

nickname
A schema-qualified name that designates a federated server reference to a table or a view.

package-name
A qualified or unqualified name that designates a package.

parameter-name
An identifier that designates a parameter that can be referenced in a procedure, user-defined function, method, or index extension.

partition-name
An identifier that designates a data partition in a partitioned table.

period-name
An identifier that designates a period. SYSTEM_TIME and BUSINESS_TIME are the only supported period names.

procedure-name
A qualified or unqualified name that designates a procedure. The unqualified form of procedure-name is an SQL identifier. An unqualified procedure name in an SQL statement is implicitly qualified. The implicit qualifier is a schema name, which is determined by the context in which the procedure appears. The qualified form is a schema-name followed by a period and an SQL identifier or a module-name followed by a period and an SQL identifier. If the procedure is defined in a module and used outside of the same module, it must be qualified by the module-name.

remote-authorization-name
An identifier that designates a data source user. The rules for authorization names vary from data source to data source.

remote-function-name
A name that designates a function registered to a data source database.

remote-object-name
A three-part name that designates a data source table or view, and that identifies the data source in which the table or view resides. The parts of this name are data-source-name, remote-schema-name, and remote-table-name.

remote-schema-name
A name that designates the schema to which a data source table or view belongs. This name is the second part of a three-part remote object name.

remote-table-name
A name that designates a table or view at a data source. This name is the third part of a three-part remote object name.

remote-type-name
A data type supported by a data source database. Do not use the long form for built-in types (use CHAR instead of CHARACTER, for example).

role-name
An identifier that designates a role.

row-type-name
A qualified or unqualified name that designates a user-defined row type. The unqualified form of row-type-name is an SQL identifier. An unqualified row-type-name in an SQL statement is implicitly qualified. The implicit qualifier is a schema name or a module name, which is determined by the context in which the row-type-name appears. The qualified form is a schema-name followed by a period and an SQL identifier or a module-name (which can also be qualified by a schema-name) followed by a period and an SQL identifier. If the row type is defined in a module and used outside of the same module, it must be qualified by the module-name.

savepoint-name
An identifier that designates a savepoint.

schema-name
An identifier that provides a logical grouping for SQL objects. A schema name used as a qualifier for the name of an object may be implicitly determined:

- from the value of the CURRENT SCHEMA special register
- from the value of the QUALIFIER precompile/bind option
- on the basis of a resolution algorithm that uses the CURRENT PATH special register
- on the basis of the schema name for another object in the same SQL statement.

To avoid complications, it is recommended that the name SESSION not be used as a schema, except as the schema for declared global temporary tables (which *must* use the schema name SESSION).

security-label-name
A qualified or unqualified name that designates a security label. An unqualified security label name in an SQL statement is implicitly qualified by the applicable security-policy-name, when one applies. If no security-policy-name is implicitly applicable, the name must be qualified.

security-policy-name
An identifier that designates a security policy.

sequence-name
An identifier that designates a sequence.

server-name
An identifier that designates an application server. In a federated system, the server name also designates the local name of a data source.

specific-name
A qualified or unqualified name that designates a specific name. An unqualified specific name in an SQL statement is implicitly qualified, depending on context.

SQL-variable-name
The name of a local variable in an SQL procedure statement. SQL variable names can be used in other SQL statements where a host variable name is allowed. The name can be qualified by the label of the compound statement that declared the SQL variable.

statement-name
An identifier that designates a prepared SQL statement.

storagegroup-name
An identifier that designates a storage group.

supertype-name
A qualified or unqualified name that designates the supertype of a type. An unqualified supertype name in an SQL statement is implicitly qualified, depending on context.

table-name
A schema-qualified name that designates a table.

table-reference
A qualified or unqualified name that designates a table. An unqualified table reference in a common table expression is implicitly qualified by the default schema.

tablespace-name
An identifier that designates a table space.

trigger-name
A schema-qualified name that designates a trigger.

type-mapping-name
An identifier that designates a data type mapping.

type-name
A qualified or unqualified name that designates a type. An unqualified type name in an SQL statement is implicitly qualified, depending on context.

typed-table-name
A schema-qualified name that designates a typed table.

typed-view-name
A schema-qualified name that designates a typed view.

usage-list-name
A schema-qualified name that designates a usage list.

user-defined-type-name
A qualified or unqualified name that designates a user-defined data type. The unqualified form of user-defined-type-name is an SQL identifier. An unqualified user-defined-type-name in an SQL statement is implicitly qualified. The implicit qualifier is a schema name or a module name, which is determined by the context in which user-defined-type-name appears. The qualified form is a schema-name followed by a period and an SQL identifier or a module-name (which can also be qualified by a schema-name) followed by a period and an SQL identifier. If the user-defined data type is defined in a module and used outside of the same module, it must be qualified by the module-name.

view-name
A schema-qualified name that designates a view.

wrapper-name
An identifier that designates a wrapper.

XML-schema-name
A qualified or unqualified name that designates an XML schema.

xsrobject-name
A qualified or unqualified name that designates an object in the XML schema repository.

Aliases for database objects

An alias can be thought of as an alternative name for an SQL object. An SQL object, therefore, can be referred to in an SQL statement by its name or by an alias.

A public alias is an alias which can always be referenced without qualifying its name with a schema name. The implicit qualifier of a public alias is SYSPUBLIC, which can also be specified explicitly.

Aliases are also known as synonyms.

An alias can be used wherever the object it is based on can be used. An alias can be created even if the object does not exist (although it must exist by the time a statement referring to it is compiled). It can refer to another alias if no circular or repetitive references are made along the chain of aliases. An alias can only refer to a module, nickname, sequence, table, view, or another alias within the same database. An alias name cannot be used where a new object name is expected, such as in the CREATE TABLE or CREATE VIEW statements; for example, if the table alias name PERSONNEL has been created, subsequent statements such as CREATE TABLE PERSONNEL... will return an error.

The option of referring to an object by an alias is not explicitly shown in the syntax diagrams, or mentioned in the descriptions of SQL statements.

A new unqualified alias of a given object type, say for a sequence, cannot have the same fully-qualified name as an existing object of that object type. For example, a sequence alias named ORDERID cannot be defined in the KANDIL schema for the sequence named KANDIL.ORDERID.

The effect of using an alias in an SQL statement is similar to that of text substitution. The alias, which must be defined by the time that the SQL statement is compiled, is replaced at

statement compilation time by the qualified object name. For example, if PBIRD.SALES is an alias for DSPN014.DIST4_SALES_148, then at compilation time:

```
SELECT * FROM PBIRD.SALES
```

effectively becomes

```
SELECT * FROM DSPN014.DIST4_SALES_148
```

Authorization IDs and authorization names

An *authorization ID* is a character string that is obtained by the database manager when a connection is established between the database manager and either an application process or a program preparation process. It designates a set of privileges. It may also designate a user or a group of users, but this property is not controlled by the database manager.

Authorization IDs are used by the database manager to provide:

- Authorization checking of SQL statements
- A default value for the QUALIFIER precompile/bind option and the CURRENT SCHEMA special register. The authorization ID is also included in the default CURRENT PATH special register and the FUNCPATH precompile/bind option.

An authorization ID applies to every SQL statement. The authorization ID that applies to a static SQL statement is the authorization ID that is used during program binding. The authorization ID that applies to a dynamic SQL statement is based on the DYNAMICRULES option supplied at bind time, and on the current runtime environment for the package issuing the dynamic SQL statement:

- In a package that has bind behavior, the authorization ID used is the authorization ID of the package owner.
- In a package that has define behavior, the authorization ID used is the authorization ID of the corresponding routine's definer.
- In a package that has run behavior, the authorization ID used is the current authorization ID of the user executing the package.
- In a package that has invoke behavior, the authorization ID used is the authorization ID currently in effect when the routine is invoked. This is called the runtime authorization ID.

For more information, see “Dynamic SQL characteristics at run time” on page 12.

An *authorization name* specified in an SQL statement should not be confused with the authorization ID of the statement. An authorization name is an identifier that is used within various SQL statements. An authorization name is used in the CREATE SCHEMA statement to designate the owner of the schema. An authorization name is used in the GRANT and REVOKE statements to designate a target of the grant or revoke operation. Granting privileges to *X* means that *X* (or a member of the group or role *X*) will subsequently be the authorization ID of statements that require those privileges.

Examples

- Assume that SMITH is the user ID and the authorization ID that the database manager obtained when a connection was established with the application process. The following statement is executed interactively:

  ```
  GRANT SELECT ON TDEPT TO KEENE
  ```

  SMITH is the authorization ID of the statement. Therefore, in a dynamic SQL statement, the default value of the CURRENT SCHEMA special register is SMITH, and in static SQL, the default value of the QUALIFIER precompile/bind option is SMITH. The authority to execute the statement is checked against SMITH, and SMITH is the *table-name* implicit qualifier based on qualification rules described in “Naming conventions and implicit object name qualifications” on page 6.

KEENE is an authorization name specified in the statement. KEENE is given the SELECT privilege on SMITH.TDEPT.

- Assume that SMITH has administrative authority and is the authorization ID of the following dynamic SQL statements, with no SET SCHEMA statement issued during the session:

```
DROP TABLE TDEPT
```

Removes the SMITH.TDEPT table.

```
DROP TABLE SMITH.TDEPT
```

Removes the SMITH.TDEPT table.

```
DROP TABLE KEENE.TDEPT
```

Removes the KEENE.TDEPT table. Note that KEENE.TDEPT and SMITH.TDEPT are different tables.

```
CREATE SCHEMA PAYROLL AUTHORIZATION KEENE
```

KEENE is the authorization name specified in the statement that creates a schema called PAYROLL. KEENE is the owner of the schema PAYROLL and is given CREATEIN, ALTERIN, and DROPIN privileges, with the ability to grant them to others.

Dynamic SQL characteristics at run time

The BIND option DYNAMICRULES determines the authorization ID that is used for checking authorization when dynamic SQL statements are processed. In addition, the option also controls other dynamic SQL attributes, such as the implicit qualifier that is used for unqualified object references, and whether certain SQL statements can be invoked dynamically.

The set of values for the authorization ID and other dynamic SQL attributes is called the dynamic SQL statement behavior. The four possible behaviors are run, bind, define, and invoke. As the following table shows, the combination of the value of the DYNAMICRULES BIND option and the runtime environment determines which of the behaviors is used. DYNAMICRULES RUN, which implies run behavior, is the default.

Table 2. How DYNAMICRULES and the runtime environment determine dynamic SQL statement behavior

DYNAMICRULES value	**Behavior of dynamic SQL statements in a stand-alone program environment**	**Behavior of dynamic SQL statements in a routine environment**
BIND	Bind behavior	Bind behavior
RUN	Run behavior	Run behavior
DEFINEBIND	Bind behavior	Define behavior
DEFINERUN	Run behavior	Define behavior
INVOKEBIND	Bind behavior	Invoke behavior
INVOKERUN	Run behavior	Invoke behavior

Run behavior
The authorization ID of the user (the ID that initially connected to the database) that executes the package is used as the value for authorization checking of dynamic SQL statements. This authorization ID is also used as the initial value for implicit qualification of unqualified object references within dynamic SQL statements.

Bind behavior
At run time, all the rules that apply to static SQL for authorization and qualification are used. The authorization ID of the package owner is used as the value for authorization checking of dynamic

SQL statements. The package default qualifier is used for implicit qualification of unqualified object references within dynamic SQL statements.

Define behavior
Define behavior applies only if the dynamic SQL statement is in a package that is run within a routine context, and the package was bound with DYNAMICRULES DEFINEBIND or DYNAMICRULES DEFINERUN. The authorization ID of the routine definer (not the routine's package binder) is used as the value for authorization checking of dynamic SQL statements. This authorization ID is also used for implicit qualification of unqualified object references within dynamic SQL statements within that routine.

Invoke behavior
Invoke behavior applies only if the dynamic SQL statement is in a package that is run within a routine context, and the package was bound with DYNAMICRULES INVOKEBIND or DYNAMICRULES INVOKERUN. The statement authorization ID in effect when the routine is invoked is used as the value for authorization checking of dynamic SQL. This authorization ID is also used for implicit qualification of unqualified object references within dynamic SQL statements within that routine. This is summarized by the following table.

Invoking Environment	ID Used
any static SQL	implicit or explicit value of the OWNER of the package the SQL invoking the routine came from
used in definition of view or trigger	definer of the view or trigger
dynamic SQL from a bind behavior package	implicit or explicit value of the OWNER of the package the SQL invoking the routine came from
dynamic SQL from a run behavior package	ID used to make the initial connection to the database
dynamic SQL from a define behavior package	definer of the routine that uses the package that the SQL invoking the routine came from
dynamic SQL from an invoke behavior package	the current authorization ID invoking the routine

Restricted statements when run behavior does not apply
When bind, define, or invoke behavior is in effect, you cannot use the following dynamic SQL statements: GRANT, REVOKE, ALTER, CREATE, DROP, COMMENT, RENAME, SET INTEGRITY, SET EVENT MONITOR STATE; or queries that reference a nickname.

Considerations regarding the DYNAMICRULES option
The CURRENT SCHEMA special register cannot be used to qualify unqualified object references within dynamic SQL statements executed from bind, define or invoke behavior packages. This is true even after you issue the SET CURRENT SCHEMA statement to change the CURRENT SCHEMA special register; the register value is changed but not used.

In the event that multiple packages are referenced during a single connection, all dynamic SQL statements prepared by those packages will exhibit the behavior specified by the DYNAMICRULES option for that specific package and the environment in which they are used.

It is important to keep in mind that when a package exhibits bind behavior, the binder of the package should not have any authorities granted that the user of the package should not receive, because a dynamic statement will be using the authorization ID of the package owner. Similarly, when a package exhibits define behavior, the definer of the routine should not have any authorities granted that the user of the package should not receive.

Authorization IDs and statement preparation

If the VALIDATE BIND option is specified at bind time, the privileges required to manipulate tables and views must also exist at bind time. If these privileges or the referenced objects do not exist, and the SQLERROR NOPACKAGE option is in effect, the bind operation will be unsuccessful. If the SQLERROR

CONTINUE option is specified, the bind operation will be successful, and any statements in error will be flagged. Any attempt to execute such a statement will result in an error.

If a package is bound with the VALIDATE RUN option, all normal bind processing is completed, but the privileges required to use the tables and views that are referenced in the application need not exist yet. If a required privilege does not exist at bind time, an incremental bind operation is performed whenever the statement is first executed in an application, and all privileges required for the statement must exist. If a required privilege does not exist, execution of the statement is unsuccessful.

Authorization checking at run time is performed using the authorization ID of the package owner.

Column names

The meaning of a *column name* depends on its context. A column name can be used to:

- Declare the name of a column, as in a CREATE TABLE statement.
- Identify a column, as in a CREATE INDEX statement.
- Specify values of the column, as in the following contexts:
  - In an aggregate function, a column name specifies all values of the column in the group or intermediate result table to which the function is applied. For example, MAX(SALARY) applies the function MAX to all values of the column SALARY in a group.
  - In a GROUP BY or ORDER BY clause, a column name specifies all values in the intermediate result table to which the clause is applied. For example, ORDER BY DEPT orders an intermediate result table by the values of the column DEPT.
  - In an expression, a search condition, or a scalar function, a column name specifies a value for each row or group to which the construct is applied. For example, when the search condition CODE = 20 is applied to some row, the value specified by the column name CODE is the value of the column CODE in that row.
- Temporarily rename a column, as in the *correlation-clause* of a *table-reference* in a FROM clause.

Qualified column names

A qualifier for a column name may be a table, view, nickname, alias, or correlation name.

Whether a column name may be qualified depends on its context:

- Depending on the form of the COMMENT ON statement, a single column name may need to be qualified. Multiple column names must be unqualified.
- Where the column name specifies values of the column, it may be qualified at the user's option.
- In the assignment-clause of an UPDATE statement, it may be qualified at the user's option.
- In all other contexts, a column name must not be qualified.

Where a qualifier is optional, it can serve two purposes. They are described under “Column name qualifiers to avoid ambiguity” on page 16 and “Column name qualifiers in correlated references” on page 18.

Correlation names

A *correlation name* can be defined in the FROM clause of a query and in the first clause of an UPDATE or DELETE statement. For example, the clause FROM X.MYTABLE Z establishes Z as a correlation name for X.MYTABLE.

```
    FROM X.MYTABLE Z
```

With Z defined as a correlation name for X.MYTABLE, only Z can be used to qualify a reference to a column of that instance of X.MYTABLE in that SELECT statement.

A correlation name is associated with a table, view, nickname, alias, nested table expression, table function, or data change table reference only within the context in which it is defined. Hence, the same correlation name can be defined for different purposes in different statements, or in different clauses of the same statement.

As a qualifier, a correlation name can be used to avoid ambiguity or to establish a correlated reference. It can also be used merely as a shorter name for a table reference. In the example, Z might have been used merely to avoid having to enter X.MYTABLE more than once.

If a correlation name is specified for a table, view, nickname, or alias name, any qualified reference to a column of that instance of the table, view, nickname, or alias must use the correlation name, rather than the table, view, nickname, or alias name. For example, the reference to EMPLOYEE.PROJECT in the following example is incorrect, because a correlation name has been specified for EMPLOYEE:

Example

```
FROM EMPLOYEE E
  WHERE EMPLOYEE.PROJECT='ABC'      * incorrect*
```

The qualified reference to PROJECT should instead use the correlation name, "E", as shown in the following example:

```
FROM EMPLOYEE E
  WHERE E.PROJECT='ABC'
```

Names specified in a FROM clause are either *exposed* or *non-exposed*. A table, view, nickname, or alias name is said to be exposed in the FROM clause if a correlation name is not specified. A correlation name is always an exposed name. For example, in the following FROM clause, a correlation name is specified for EMPLOYEE but not for DEPARTMENT, so DEPARTMENT is an exposed name, and EMPLOYEE is not:

```
FROM EMPLOYEE E, DEPARTMENT
```

A table, view, nickname, or alias name that is exposed in a FROM clause may be the same as any other table name, view name or nickname exposed in that FROM clause or any correlation name in the FROM clause. This may result in ambiguous column name references which returns an error (SQLSTATE 42702).

The first two FROM clauses shown in the following list are correct, because each one contains no more than one reference to EMPLOYEE that is exposed:

1. Given the FROM clause:

   ```
   FROM EMPLOYEE E1, EMPLOYEE
   ```

   a qualified reference such as EMPLOYEE.PROJECT denotes a column of the second instance of EMPLOYEE in the FROM clause. A qualified reference to the first instance of EMPLOYEE must use the correlation name "E1" (E1.PROJECT).
2. Given the FROM clause:

   ```
   FROM EMPLOYEE, EMPLOYEE E2
   ```

   a qualified reference such as EMPLOYEE.PROJECT denotes a column of the first instance of EMPLOYEE in the FROM clause. A qualified reference to the second instance of EMPLOYEE must use the correlation name "E2" (E2.PROJECT).
3. Given the FROM clause:

   ```
   FROM EMPLOYEE, EMPLOYEE
   ```

   the two exposed table names included in this clause (EMPLOYEE and EMPLOYEE) are the same. This is allowed, but references to specific column names would be ambiguous (SQLSTATE 42702).
4. Given the following statement:

```
SELECT *
  FROM EMPLOYEE E1, EMPLOYEE E2             * incorrect *
  WHERE EMPLOYEE.PROJECT = 'ABC'
```

the qualified reference EMPLOYEE.PROJECT is incorrect, because both instances of EMPLOYEE in the FROM clause have correlation names. Instead, references to PROJECT must be qualified with either correlation name (E1.PROJECT or E2.PROJECT).

5. Given the FROM clause:

```
FROM EMPLOYEE, X.EMPLOYEE
```

a reference to a column in the second instance of EMPLOYEE must use X.EMPLOYEE (X.EMPLOYEE.PROJECT). If X is the CURRENT SCHEMA special register value in dynamic SQL or the QUALIFIER precompile/bind option in static SQL, then the columns cannot be referenced since any such reference would be ambiguous.

The use of a correlation name in the FROM clause also allows the option of specifying a list of column names to be associated with the columns of the result table. As with a correlation name, these listed column names become the *exposed* names of the columns that must be used for references to the columns throughout the query. If a column name list is specified, then the column names of the underlying table become *non-exposed*.

Given the FROM clause:

```
FROM DEPARTMENT D (NUM,NAME,MGR,ANUM,LOC)
```

a qualified reference such as D.NUM denotes the first column of the DEPARTMENT table that is defined in the table as DEPTNO. A reference to D.DEPTNO using this FROM clause is incorrect since the column name DEPTNO is a non-exposed column name.

Column name qualifiers to avoid ambiguity

In the context of a function, a GROUP BY clause, ORDER BY clause, an expression, or a search condition, a column name refers to values of a column in some table, view, nickname, nested table expression or table function. The tables, views, nicknames, nested table expressions and table functions that might contain the column are called the *object tables* of the context. Two or more object tables might contain columns with the same name; one reason for qualifying a column name is to designate the table from which the column comes. Qualifiers for column names are also useful in SQL procedures to distinguish column names from SQL variable names used in SQL statements.

A nested table expression or table function will consider *table-references* that precede it in the FROM clause as object tables. The *table-references* that follow are not considered as object tables.

Table designators

A qualifier that designates a specific object table is called a *table designator*. The clause that identifies the object tables also establishes the table designators for them. For example, the object tables of an expression in a SELECT clause are named in the FROM clause that follows it:

```
SELECT CORZ.COLA, OWNY.MYTABLE.COLA
  FROM OWNX.MYTABLE CORZ, OWNY.MYTABLE
```

Table designators in the FROM clause are established as follows:

- A name that follows a table, view, nickname, alias, nested table expression or table function is both a correlation name and a table designator. Thus, CORZ is a table designator. CORZ is used to qualify the first column name in the select list.
- An exposed table, view name, nickname or alias is a table designator. Thus, OWNY.MYTABLE is a table designator. OWNY.MYTABLE is used to qualify the second column name in the select list.

When qualifying a column with the exposed table name form of a table designator, either the qualified or unqualified form of the exposed table name can be used. If the qualified form is used, the qualifier must be the same as the default qualifier for the exposed table name.

For example, assume that the current schema is CORPDATA.

```
SELECT CORPDATA.EMPLOYEE.WORKDEPT FROM EMPLOYEE
```

is valid because the EMPLOYEE table referenced in the FROM clause fully qualifies to CORPDATA.EMPLOYEE, which matches the qualifier for the WORKDEPT column.

```
SELECT EMPLOYEE.WORKDEPT, REGEMP.WORKDEPT
  FROM CORPDATA.EMPLOYEE, REGION.EMPLOYEE REGEMP
```

is also valid, because the first select list column references the unqualified exposed table designator CORPDATA.EMPLOYEE, which is in the FROM clause, and the second select list column references the correlation name REGEMP of the table object REGION.EMPLOYEE, which is also in the FROM clause.

Now assume that the current schema is REGION.

```
SELECT CORPDATA.EMPLOYEE.WORKDEPT FROM EMPLOYEE
```

is not valid because the EMPLOYEE table referenced in the FROM clause fully qualifies to REGION.EMPLOYEE, and the qualifier for the WORKDEPT column represents the CORPDATA.EMPLOYEE table.

Each table designator should be unique within a particular FROM clause to avoid the possibility of ambiguous references to columns.

Avoiding undefined or ambiguous references

When a column name refers to values of a column, exactly one object table must include a column with that name. The following situations are considered errors:

- No object table contains a column with the specified name. The reference is undefined.
- The column name is qualified by a table designator, but the table designated does not include a column with the specified name. Again the reference is undefined.
- The name is unqualified, and more than one object table includes a column with that name. The reference is ambiguous.
- The column name is qualified by a table designator, but the table designated is not unique in the FROM clause and both occurrences of the designated table include the column. The reference is ambiguous.
- The column name is in a nested table expression which is not preceded by the TABLE keyword or in a table function or nested table expression that is the right operand of a right outer join or a full outer join and the column name does not refer to a column of a *table-reference* within the nested table expression's fullselect. The reference is undefined.

Avoid ambiguous references by qualifying a column name with a uniquely defined table designator. If the column is contained in several object tables with different names, the table names can be used as designators. Ambiguous references can also be avoided without the use of the table designator by giving unique names to the columns of one of the object tables using the column name list following the correlation name.

When qualifying a column with the exposed table name form of a table designator, either the qualified or unqualified form of the exposed table name may be used. However, the qualifier used and the table used must be the same after fully qualifying the table name, view name or nickname and the table designator.

1. If the authorization ID of the statement is CORPDATA:

   ```
   SELECT CORPDATA.EMPLOYEE.WORKDEPT
     FROM EMPLOYEE
   ```

   is a valid statement.

2. If the authorization ID of the statement is REGION:

```
SELECT CORPDATA.EMPLOYEE.WORKDEPT
  FROM EMPLOYEE                          * incorrect *
```

   is invalid, because EMPLOYEE represents the table REGION.EMPLOYEE, but the qualifier for WORKDEPT represents a different table, CORPDATA.EMPLOYEE.

Column name qualifiers in correlated references

A *fullselect* is a form of a query that may be used as a component of various SQL statements. A fullselect used within a search condition of any statement is called a *subquery*. A fullselect used to retrieve a single value as an expression within a statement is called a *scalar fullselect* or *scalar subquery*. A fullselect used in the FROM clause of a query is called a *nested table expression*. Subqueries in search conditions, scalar subqueries and nested table expressions are referred to as subqueries through the remainder of this topic.

A subquery may include subqueries of its own, and these may, in turn, include subqueries. Thus an SQL statement may contain a hierarchy of subqueries. Those elements of the hierarchy that contain subqueries are said to be at a higher level than the subqueries they contain.

Every element of the hierarchy contains one or more table designators. A subquery can reference not only the columns of the tables identified at its own level in the hierarchy, but also the columns of the tables identified previously in the hierarchy, back to the highest level of the hierarchy. A reference to a column of a table identified at a higher level is called a *correlated reference*.

For compatibility with existing standards for SQL, both qualified and unqualified column names are allowed as correlated references. However, it is good practice to qualify all column references used in subqueries; otherwise, identical column names may lead to unintended results. For example, if a table in a hierarchy is altered to contain the same column name as the correlated reference and the statement is prepared again, the reference will apply to the altered table.

When a column name in a subquery is qualified, each level of the hierarchy is searched, starting at the same subquery as the qualified column name appears and continuing to the higher levels of the hierarchy until a table designator that matches the qualifier is found. Once found, it is verified that the table contains the given column. If the table is found at a higher level than the level containing column name, then it is a correlated reference to the level where the table designator was found. A nested table expression must be preceded with the optional TABLE keyword in order to search the hierarchy above the fullselect of the nested table expression.

When the column name in a subquery is not qualified, the tables referenced at each level of the hierarchy are searched, starting at the same subquery where the column name appears and continuing to higher levels of the hierarchy, until a match for the column name is found. If the column is found in a table at a higher level than the level containing column name, then it is a correlated reference to the level where the table containing the column was found. If the column name is found in more than one table at a particular level, the reference is ambiguous and considered an error.

In either case, T, used in the following example, refers to the table designator that contains column C. A column name, T.C (where T represents either an implicit or an explicit qualifier), is a correlated reference if, and only if, these conditions are met:

- T.C is used in an expression of a subquery.
- T does not designate a table used in the from clause of the subquery.
- T designates a table used at a higher level of the hierarchy that contains the subquery.

Since the same table, view or nickname can be identified at many levels, unique correlation names are recommended as table designators. If T is used to designate a table at more than one level (T is the table name itself or is a duplicate correlation name), T.C refers to the level where T is used that most directly contains the subquery that includes T.C. If a correlation to a higher level is needed, a unique correlation name must be used.

The correlated reference T.C identifies a value of C in a row or group of T to which two search conditions are being applied: condition 1 in the subquery, and condition 2 at some higher level. If condition 2 is used in a WHERE clause, the subquery is evaluated for each row to which condition 2 is applied. If condition 2 is used in a HAVING clause, the subquery is evaluated for each group to which condition 2 is applied.

For example, in the following statement, the correlated reference X.WORKDEPT (in the last line) refers to the value of WORKDEPT in table EMPLOYEE at the level of the first FROM clause. (That clause establishes X as a correlation name for EMPLOYEE.) The statement lists employees who make less than the average salary for their department.

```
SELECT EMPNO, LASTNAME, WORKDEPT
  FROM EMPLOYEE X
  WHERE SALARY < (SELECT AVG(SALARY)
                    FROM EMPLOYEE
                    WHERE WORKDEPT = X.WORKDEPT)
```

The next example uses THIS as a correlation name. The statement deletes rows for departments that have no employees.

```
DELETE FROM DEPARTMENT THIS
   WHERE NOT EXISTS(SELECT *
                      FROM EMPLOYEE
                      WHERE WORKDEPT = THIS.DEPTNO)
```

References to variables

A *variable* in an SQL statement specifies a value that can be changed when the SQL statement is executed. There are several types of variables used in SQL statements:

host variable
Host variables are defined by statements of a host language. For more information about how to refer to host variables, see “References to host variables” on page 20.

transition variable
Transition variables are defined in a trigger and refer to either the old or new values of columns. For more information about how to refer to transition variables, see "CREATE TRIGGER statement" in the *SQL Reference Volume 2* .

SQL variable
SQL variables are defined by an SQL compound statement in an SQL function, SQL method, SQL procedure, trigger, or dynamic SQL statement. For more information about SQL variables, see "References to SQL parameters, SQL variables, and global variables" in the *SQL Reference Volume 2* .

global variable
Global variables are defined by the CREATE VARIABLE statement. For more information about global variables, see "CREATE VARIABLE" and "References to SQL parameters, SQL variables, and global variables" in the *SQL Reference Volume 2* .

module variable
Module variables are defined by the ALTER MODULE statement using the ADD VARIABLE or PUBLISH VARIABLE operation. For more information about module variables, see "ALTER MODULE" in the *SQL Reference Volume 2* .

SQL parameter
SQL parameters are defined by a CREATE FUNCTION, CREATE METHOD, or CREATE PROCEDURE statement. For more information about SQL parameters, see "References to SQL parameters, SQL variables, and global variables" in the *SQL Reference Volume 2* .

parameter marker
Parameter markers are specified in a dynamic SQL statement where host variables would be specified if the statement were a static SQL statement. An SQL descriptor or parameter binding is used to associate a value with a parameter marker during dynamic SQL statement processing. For more information about parameter markers, see "Parameter markers" in the *SQL Reference Volume 2* .

References to host variables

A *host variable* is either:

- A variable in a host language such as a C variable, a C++ variable, a COBOL data item, a FORTRAN variable, or a Java™ variable

or:

- A host language construct that was generated by an SQL precompiler from a variable declared using SQL extensions

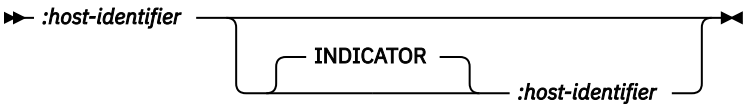
that is referenced in an SQL statement. Host variables are either directly defined by statements in the host language or are indirectly defined using SQL extensions.

A host variable in an SQL statement must identify a host variable described in the program according to the rules for declaring host variables.

All host variables used in an SQL statement must be declared in an SQL DECLARE section in all host languages except REXX. No variables may be declared outside an SQL DECLARE section with names identical to variables declared inside an SQL DECLARE section. An SQL DECLARE section begins with BEGIN DECLARE SECTION and ends with END DECLARE SECTION.

The meta-variable *host-variable*, as used in the syntax diagrams, shows a reference to a host variable. A host-variable as the target variable in a SET variable statement or in the INTO clause of a FETCH, SELECT INTO, or VALUES INTO statement, identifies a host variable to which a value from a column of a row or an expression is assigned. In all other contexts a host-variable specifies a value to be passed to the database manager from the application program.

The meta-variable *host-variable* in syntax diagrams can generally be expanded to:



Each *host-identifier* must be declared in the source program. The variable designated by the second host-identifier must have a data type of small integer.

The first host-identifier designates the *main variable*. Depending on the operation, it either provides a value to the database manager or is provided a value from the database manager. An input host variable provides a value in the runtime application code page. An output host variable is provided a value that, if necessary, is converted to the runtime application code page when the data is copied to the output application variable. A given host variable can serve as both an input and an output variable in the same program.

The second host-identifier designates its *indicator variable*. Indicator variables appear in two forms; normal indictor variables, and extended indicator variables.

The normal indicator variable has the following purposes:

- Specify a non-null value. A 0 (zero), or positive value of the indicator variable specifies that the associated, first, *host-identifier* provides the value of this host variable reference.
- Specify the null value. A negative value of the indicator variable specifies the null value.
- On output, indicate that a numeric conversion error (such as division by 0 or overflow) has occurred, if the **dft_sqlmathwarn** database configuration parameter is set to "yes" (or was set to "yes" during binding of a static SQL statement). A -2 value of the indicator variable indicates a null result because of either numeric truncation or friendly arithmetic warnings.
- On output, report the original length of a truncated string (if the source of the value is not a large object type).
- On output, report the seconds portion of a time if the time is truncated on assignment to a host variable.

Extended indicator variables are limited to the input of host variables. The extended indicator variable has the following purposes:

- Specify a non-null value. A 0 (zero), or positive value specifies that the associated, first, *host-identifier* provides the value of this host variable reference.
- Specify the null value. A -1, -2, -3, -4, or -6 value specifies the null value.
- Specify the default value. A -5 value specifies the target column for this host variable is to be set to its default value.
- Specify an unassigned value. A -7 value specifies the target column for this host variable is to be treated as if it had not been specified in the statement.

Extended indicator variables are only enabled if requested, and all indicator variables are otherwise normal indicator variables. In comparison to normal indicator variables, extended indicator variables have no additional restrictions for where the values for null and non-null can be used. There are no restrictions against using extended indicator variable values in indicator structures with host structures. Restrictions on where extended indicator variable values default and unassigned are allowed apply uniformly, no matter how they are represented in the host application. The default and unassigned extended indicator variable values may only appear in limited, specified uses. They may appear in expressions containing only a single host variable, or a host variable being explicitly cast (assigned to a column). Output indicator variable values are never extended indicator variables.

When extended indicator variables are enabled, there are no restrictions against use of 0 (zero), or positive indicator variable values. However, negative indicator variable values outside the range -1 through -7 must not be input (SQLSTATE 22010). When enabled, the default and unassigned extended indicator variable values must not appear in contexts in which they are not supported (SQLSTATE 22539).

When extended indicator variables are enabled, rules for data type validation in assignment and comparison are loosened for host variables whose extended indicator values are negative. Data type assignment and comparison validation rules will not be enforced for host variables having the values null, default, or unassigned.

For example, if :HV1:HV2 is used to specify an insert or update value, and if HV2 is negative, the value specified is the null value. If HV2 is not negative the value specified is the value of HV1.

Similarly, if :HV1:HV2 is specified in an INTO clause of a FETCH, SELECT INTO, or VALUES INTO statement, and if the value returned is null, HV1 is not changed, and HV2 is set to a negative value. If the database is configured with **dft_sqlmathwarn** yes (or was during binding of a static SQL statement), HV2 could be -2. If HV2 is -2, a value for HV1 could not be returned because of an error converting to the numeric type of HV1, or an error evaluating an arithmetic expression that is used to determine the value for HV1. If the value returned is not null, that value is assigned to HV1 and HV2 is set to zero (unless the assignment to HV1 requires string truncation of a non-LOB string; in which case HV2 is set to the original length of the string). If an assignment requires truncation of the seconds part of a time, HV2 is set to the number of seconds.

If the second host identifier is omitted, the host-variable does not have an indicator variable. The value specified by the host-variable reference :HV1 is always the value of HV1, and null values cannot be assigned to the variable. Thus, this form should not be used in an INTO clause unless the corresponding column cannot contain null values. If this form is used and the column contains nulls, the database manager will generate an error at run time.

An SQL statement that references host variables must be within the scope of the declaration of those host variables. For host variables referenced in the SELECT statement of a cursor, that rule applies to the OPEN statement rather than to the DECLARE CURSOR statement.

Example

Using the PROJECT table, set the host variable PNAME (VARCHAR(26)) to the project name (PROJNAME), the host variable STAFF (DECIMAL(5,2)) to the mean staffing level (PRSTAFF), and the host variable MAJPROJ (CHAR(6)) to the major project (MAJPROJ) for project (PROJNO) 'IF1000'. Columns PRSTAFF and MAJPROJ may contain null values, so provide indicator variables STAFF_IND (SMALLINT) and MAJPROJ_IND (SMALLINT).

```
SELECT PROJNAME, PRSTAFF, MAJPROJ
  INTO :PNAME, :STAFF :STAFF_IND, :MAJPROJ :MAJPROJ_IND
```

```
    FROM PROJECT
    WHERE PROJNO = 'IF1000'
```

MBCS Considerations: Whether multi-byte characters can be used in a host variable name depends on the host language.

Variables in dynamic SQL

In dynamic SQL statements, parameter markers are used instead of host variables. A parameter marker represents a position in a dynamic SQL statement where the application will provide a value; that is, where a host variable would be found if the statement string were a static SQL statement. The following example shows a static SQL statement using host variables:

```
INSERT INTO DEPARTMENT
  VALUES (:HV_DEPTNO, :HV_DEPTNAME, :HV_MGRNO, :HV_ADMRDEPT)
```

This example shows a dynamic SQL statement using unnamed parameter markers:

```
INSERT INTO DEPARTMENT VALUES (?, ?, ?, ?)
```

This example shows a dynamic SQL statement using named parameter markers:

```
INSERT INTO DEPARTMENT
  VALUES (:DEPTNO, :DEPTNAME, :MGRNO, :ADMRDEPT)
```

Named parameter markers can be used to improve the readability of dynamic statement. Although named parameter markers look like host variables, named parameter markers have no associated value and therefore a value must be provided for the parameter marker when the statement is executed. If the INSERT statement using named parameter markers has been prepared and given the prepared statement name of DYNSTMT, then values can be provided for the parameter markers using the following statement:

```
EXECUTE DYNSTMT
  USING :HV_DEPTNO, :HV_DEPTNAME :HV_MGRNO, :HV_ADMRDEPT
```

This same EXECUTE statement could be used if the INSERT statement using unnamed parameter markers had been prepared and given the prepared statement name of DYNSTMT.

References to LOB variables

Regular BLOB, CLOB, and DBCLOB variables, LOB locator variables (see “References to LOB locator variables” on page 22), and LOB file reference variables (see “References to LOB file reference variables” on page 23) can be defined in all host languages. Where LOBs are allowed, the term *host-variable* in a syntax diagram can refer to a regular host variable, a locator variable, or a file reference variable. Since these are not native data types, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable. In the case of REXX, LOBs are mapped to strings.

It is sometimes possible to define a large enough variable to hold an entire large object value. If this is true and if there is no performance benefit to be gained by deferred transfer of data from the server, a locator is not needed. However, since host language or space restrictions will often dictate against storing an entire large object in temporary storage at one time or because of performance benefit, a large object may be referenced via a locator and portions of that object may be selected into or updated from host variables that contain only a portion of the large object at one time.

References to LOB locator variables

A *locator variable* is a host variable that contains the locator representing a LOB value on the application server.

A locator variable in an SQL statement must identify a locator variable described in the program according to the rules for declaring locator variables. This is always indirectly through an SQL statement.

The term locator variable, as used in the syntax diagrams, shows a reference to a locator variable. The meta-variable *locator-variable* can be expanded to include a *host-identifier* the same as that for *host-variable*.

As with all other host variables, a large object locator variable may have an associated indicator variable. Indicator variables for large object locator host variables behave in the same way as indicator variables for other data types. When a null value is returned from the database, the indicator variable is set and the locator host variable is unchanged. This means a locator can never point to a null value.

If a locator-variable that does not currently represent any value is referenced, an error is raised (SQLSTATE 0F001).

At transaction commit, or any transaction termination, all locators acquired by that transaction are released.

References to LOB file reference variables

BLOB, CLOB, and DBCLOB file reference variables are used for direct file input and output for LOBs, and can be defined in all host languages. Since these are not native data types, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable. In the case of REXX, LOBs are mapped to strings.

A file reference variable represents (rather than contains) the file, just as a LOB locator represents, rather than contains, the LOB bytes. Database queries, updates and inserts may use file reference variables to store or to retrieve single column values.

A file reference variable has the following properties:

Data Type
BLOB, CLOB, or DBCLOB. This property is specified when the variable is declared.

Direction
This must be specified by the application program at run time (as part of the File Options value). The direction is one of:

- Input (used as a source of data on an EXECUTE statement, an OPEN statement, an UPDATE statement, an INSERT statement, or a DELETE statement).
- Output (used as the target of data on a FETCH statement or a SELECT INTO statement).

File name
This must be specified by the application program at run time. It is one of:

- The complete path name of the file (which is advised).
- A relative file name. If a relative file name is provided, it is appended to the current path of the client process.

Within an application, a file should only be referenced in one file reference variable.

File Name Length
This must be specified by the application program at run time. It is the length of the file name (in bytes).

File Options
An application must assign one of a number of options to a file reference variable before it makes use of that variable. Options are set by an INTEGER value in a field in the file reference variable structure. One of the following values must be specified for each file reference variable:

- Input (from client to server)

  SQL_FILE_READ
  This is a regular file that can be opened, read and closed. (The option is SQL-FILE-READ in COBOL, sql_file_read in FORTRAN, and READ in REXX.)

- Output (from server to client)

SQL_FILE_CREATE
Create a new file. If the file already exists, an error is returned. (The option is SQL-FILE-CREATE in COBOL, sql_file_create in FORTRAN, and CREATE in REXX.)

SQL_FILE_OVERWRITE (Overwrite)
If an existing file with the specified name exists, it is overwritten; otherwise a new file is created. (The option is SQL-FILE-OVERWRITE in COBOL, sql_file_overwrite in FORTRAN, and OVERWRITE in REXX.)

SQL_FILE_APPEND
If an existing file with the specified name exists, the output is appended to it; otherwise a new file is created. (The option is SQL-FILE-APPEND in COBOL, sql_file_append in FORTRAN, and APPEND in REXX.)

Data Length
This is unused on input. On output, the implementation sets the data length to the length of the new data written to the file. The length is in bytes.

As with all other host variables, a file reference variable may have an associated indicator variable.

Example of an output file reference variable (in C)

Given a declare section coded as:

```
EXEC SQL BEGIN DECLARE SECTION
   SQL TYPE IS CLOB_FILE  hv_text_file;
   char  hv_patent_title[64];
EXEC SQL END DECLARE SECTION
```

Following preprocessing this would be:

```
EXEC SQL BEGIN DECLARE SECTION
   /* SQL TYPE IS CLOB_FILE  hv_text_file; */
   struct {
       unsigned long  name_length; //  File Name Length
       unsigned long  data_length; //  Data Length
       unsigned long  file_options; // File Options
       char           name[255];   // File Name
   } hv_text_file;
   char  hv_patent_title[64];
EXEC SQL END DECLARE SECTION
```

Then, the following code can be used to select from a CLOB column in the database into a new file referenced by :hv_text_file.

```
strcpy(hv_text_file.name, "/u/gainer/papers/sigmod.94");
hv_text_file.name_length = strlen("/u/gainer/papers/sigmod.94");
hv_text_file.file_options = SQL_FILE_CREATE;

EXEC SQL SELECT content INTO :hv_text_file from papers
     WHERE TITLE = 'The Relational Theory behind Juggling';
```

Example of an input file reference variable (in C)

Given the same declare section as the previous one, the following code can be used to insert the data from a regular file referenced by :hv_text_file into a CLOB column.

```
strcpy(hv_text_file.name, "/u/gainer/patents/chips.13");
hv_text_file.name_length = strlen("/u/gainer/patents/chips.13");
hv_text_file.file_options = SQL_FILE_READ:
strcpy(:hv_patent_title, "A Method for Pipelining Chip Consumption");

EXEC SQL INSERT INTO patents( title, text )
         VALUES(:hv_patent_title, :hv_text_file);
```

References to structured type host variables

Structured type variables can be defined in all host languages except FORTRAN, REXX, and Java. Since these are not native data types, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable.

As with all other host variables, a structured type variable may have an associated indicator variable. Indicator variables for structured type host variables behave in the same way as indicator variables for other data types. When a null value is returned from the database, the indicator variable is set and the structured type host variable is unchanged.

The actual host variable for a structured type is defined as a built-in data type. The built-in data type associated with the structured type must be assignable:

- from the result of the FROM SQL transform function for the structured type as defined by the specified TRANSFORM GROUP option of the precompile command; and
- to the parameter of the TO SQL transform function for the structured type as defined by the specified TRANSFORM GROUP option of the precompile command.

If using a parameter marker instead of a host variable, the appropriate parameter type characteristics must be specified in the SQLDA. This requires a "doubled" set of SQLVAR structures in the SQLDA, and the SQLDATATYPE_NAME field of the secondary SQLVAR must be filled with the schema and type name of the structured type. If the schema is omitted in the SQLDA structure, an error results (SQLSTATE 07002).

Example

Define the host variables *hv_poly* and *hv_point* (of type POLYGON, using built-in type BLOB(1048576)) in a C program.

```
EXEC SQL BEGIN DECLARE SECTION;
      static SQL
         TYPE IS POLYGON AS BLOB(1M)
         hv_poly, hv_point;
EXEC SQL END DECLARE SECTION;
```

SQL path

The SQL path is an ordered list of schema names. The database manager uses the SQL path to resolve the schema name for unqualified data type names (both built-in types and distinct types), global variable names, module names, function names, and procedure names that appear in any context other than as the main object of a CREATE, DROP, COMMENT, GRANT or REVOKE statement. For details, see "Qualification of unqualified object names".

For example, if the SQL path is SYSIBM. SYSFUN, SYSPROC, SYSIBMADM, SMITH, XGRAPHICS2 and an unqualified distinct type name MYTYPE was specified, the database manager looks for MYTYPE first in schema SYSIBM, then SYSFUN, then SYSPROC, then SYSIBMADM, then SMITH, and then XGRAPHICS2.

The SQL path used depends on the SQL statement:

- For static SQL statements (except for a CALL variable statement), the SQL path used is the SQL path specified when the containing package, procedure, function, trigger, or view was created.
- For dynamic SQL statements (and for a CALL variable statement), the SQL path is the value of the CURRENT PATH special register. CURRENT PATH can be set by the SET PATH statement.

If the SQL path is not explicitly specified, the SQL path is the system path followed by the authorization ID of the statement. .

Qualification of unqualified object names

Unqualified object names are implicitly qualified. The rules for qualifying a name differ depending on the type of object that the name identifies.

Unqualified alias, index, package, sequence, table, trigger, and view names

Unqualified alias, index, package, sequence, table, trigger, and view names are implicitly qualified by the default schema.

For static SQL statements, the default schema is the default schema specified when the containing function, package, procedure, or trigger was created.

For dynamic SQL statements, the default schema is the default schema specified for the application process. The default schema can be specified for the application process by using the SET SCHEMA statement. If the default schema is not explicitly specified, the default schema is the authorization ID of the statement.

Unqualified user-defined type, function, procedure, specific, global variable and module names

The qualification of data type (both built-in types and distinct types), global variable, module, function, procedure, and specific names depends on the SQL statement in which the unqualified name appears:

- If an unqualified name is the main object of a CREATE, ALTER, COMMENT, DROP, GRANT, or REVOKE statement, the name is implicitly qualified using the same rules as for qualifying unqualified table names (See “Unqualified alias, index, package, sequence, table, trigger, and view names” on page 26). The main object of an ADD, COMMENT, DROP, or PUBLISH operation of the ALTER MODULE statement must be specified without any qualifier.
- If the context of the reference is within a module, the database manager searches the module for the object, applying the appropriate resolution for the type of object to find a match. If no match is found, the search continues as specified in the next bullet.
- Otherwise, the implicit schema name is determined as follows:
  - For distinct type names, the database manager searches the SQL path and selects the first schema in the SQL path such that the data type exists in the schema.
  - For global variables, the database manager searches the SQL path and selects the first schema in the SQL path such that the global variable exists in the schema.
  - For procedure names, the database manager uses the SQL path in conjunction with procedure resolution.
  - For function names, the database manager uses the SQL path in conjunction with function resolution .
  - For specific names specified for sourced functions, see "CREATE FUNCTION (Sourced)".

New SYSIBM functions override unqualified user-defined functions with the same name

An existing user-defined function or a user-defined procedure might have the same name and signature as a new built-in function or SQL administrative routine. In such cases, an unqualified reference to those functions or routines in a dynamic SQL statement runs the built-in function or SQL administrative routine instead of the user-defined one.

The default SQL path contains the schemas SYSIBM, SYSFUN, SYSPROC, and SYSIBMADM before the schema name that is the value of the USER special register. These system schemas are also included in the SQL path when it is explicitly set with the SET PATH statement or the FUNCPATH bind option. During function resolution and procedure resolution, the built-in functions and SQL administrative routines in the SYSIBM, SYSFUN, SYSPROC, and SYSIBMADM schemas are encountered before user-defined functions and user-defined procedures.

This change does not affect static SQL in packages or SQL objects such as views, triggers, or SQL functions. In these cases, the user-defined function or procedure continues to run until an explicit bind of the package, or drop and create of the SQL object.

To run an unqualified user-defined routine instead of a new SYSIBM function with the same name, rename the user-defined routine or fully qualify the name before you run it. Alternatively, place in the SQL path

the schema in which the user-defined routine exists before the schema in which the built-in functions and SQL administrative routines exist. However, promoting the schema in the SQL path increases the resolution time for all built-in functions and SQL administrative routines because the system schemas are considered first.

Resolving qualified object names

Objects that are defined in a module that are available for use outside the module must be qualified by the module name. Since a module is a schema object that can also be implicitly qualified, the published module objects can be qualified using an unqualified module name or a schema-qualified module name. When an unqualified module name is used, the reference to the module object appears the same as a schema-qualified object that is not part of a module. Within a specific scope, such as a compound SQL statement, a two-part identifier could also be:

- a column name qualified by a table name
- a row field name qualified by a variable name
- a variable name qualified by a label
- a routine parameter name qualified by a routine name

These objects are resolved within their scope, before considering either schema objects or module object. The following process is used to resolve objects with two-part identifiers that could be a schema object or a module object.

- If the context of the reference is within a module and the qualifier matches the module name, the database manager searches the module for the object, applying the appropriate resolution for the type of object to find a match among published and unpublished module objects. If no match is found, the search continues as specified in the next bullets.
- Assume that the qualifier is a schema name and, if the schema exists, resolve the object in the schema.
- If the qualifier is not an existing schema or the object is not found in the schema that matches the qualifier and the qualifier did not match the context module name, search for the first module that matches the qualifier in the schemas on the SQL path. If authorized to the matching module, resolve to the object in that module, considering only published module objects.
- If the qualifier is not found as a module on the SQL path and the qualifier did not match the context module name, check for a module public synonym that matches the qualifier. If found, resolve the object in the module identified by the module public synonym, considering only published module objects.

Reserved package names

A specific set of package names have been explicitly reserved for system use. To avoid collision with system use of these names it is recommended that these package names not be used by any application.

The set of package names that are reserved meet one of the following criteria:

- The package schema is a reserved schema
- The package schema is NULLID and the package name matches a reserved package name or contains a prefix that matches a reserved package name prefix.

The following package names are reserved within the NULLID schema:

- AGGDISC
- PRINTSG
- TUPLEWRT

The following package name prefixes are reserved within the NULLID schema:

- AOT
- ATS

- CADM
- CLI
- DB2
- POLY
- REVA
- SPIM
- SPUT
- SQL
- SYS
- TOOL

Data types

The smallest unit of data that can be manipulated in SQL is called a *value*. Values are interpreted according to the data type of their source.

Sources include:

- Constants
- Columns
- Functions
- Expressions
- Special registers
- Variables (such as host variables, SQL variables, global variables, parameter markers, module variable, and parameters of routines)
- Boolean values

All data types include the null value as a possible value. The null value is a special value that is distinct from all non-null values and thereby denotes the absence of a (non-null) value. Although all data types include the null value, a column defined as NOT NULL cannot contain null values.

A Unicode database also supports national character strings that are synonyms for graphic strings.

Built-in and user-defined data types

Data types that are pre-defined for use within the database management system are called *built-in data types*. Figure 1 on page 29 shows the supported built-in data types.

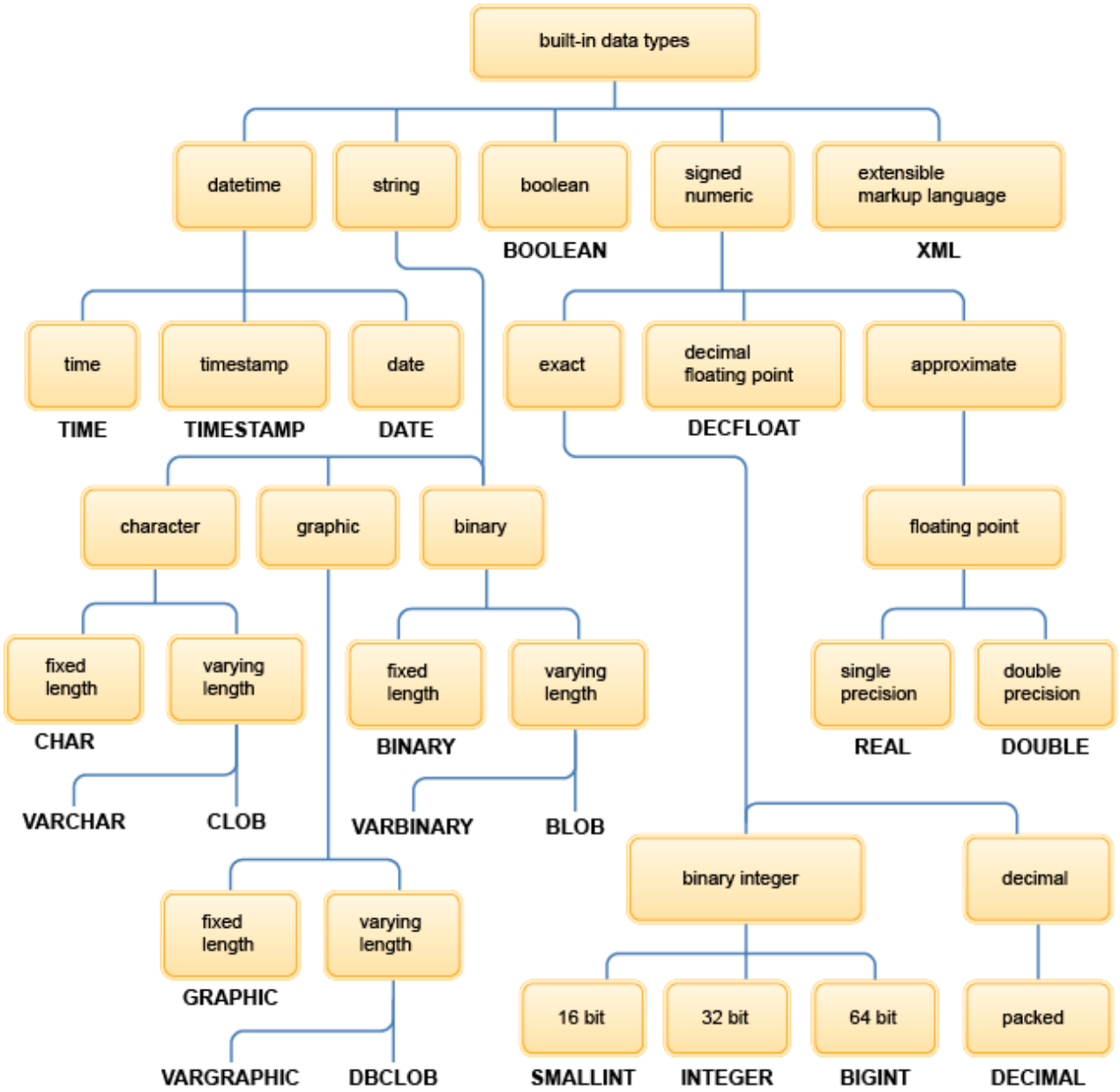


Figure 1. Built-in Data Types

Support for the following user-defined data types is also provided:

- Array
- Cursor
- Distinct
- Row
- Structured

Data type list

Numbers

The numeric data types are integer, decimal, floating-point, and decimal floating-point.

The numeric data types are categorized as follows:

- Exact numerics: integer and decimal

- Decimal floating-point
- Approximate numerics: floating-point

Integer includes small integer, large integer, and big integer. Integer numbers are exact representations of integers. Decimal numbers are exact representations of numbers with a fixed precision and scale. Integer and decimal numbers are considered exact numeric types.

Decimal floating-point numbers can have a precision of 16 or 34. Decimal floating-point supports both exact representations of real numbers and approximation of real numbers and so is not considered either an exact numeric type or an approximate numeric type.

Floating-point includes single precision and double precision. Floating-point numbers are approximations of real numbers and are considered approximate numeric types.

All numbers have a *sign*, a *precision*, and a *scale*. For all numbers except decimal floating-point, if a column value is zero, the sign is positive. Decimal floating-point numbers include negative and positive zeros. Decimal floating-point has distinct values for a number and the same number with various exponents (for example: 0.0, 0.00, 0.0E5, 1.0, 1.00, 1.0000). The precision is the total number of decimal digits, excluding the sign. The scale is the total number of decimal digits to the right of the decimal point. If there is no decimal point, the scale is zero.

See also the data type section in the description of "CREATE TABLE statement" in the *SQL Reference Volume 2*.

.

Small integer (SMALLINT)

A *small integer* is a two-byte integer with a precision of 5 digits. The range of small integers is -32 768 to 32 767.

Large integer (INTEGER)

A *large integer* is a four-byte integer with a precision of 10 digits. The range of large integers is -2 147 483 648 to +2 147 483 647.

Big integer (BIGINT)

A *big integer* is an eight-byte integer with a precision of 19 digits. The range of big integers is -9 223 372 036 854 775 808 to +9 223 372 036 854 775 807.

Decimal (DECIMAL or NUMERIC)

A *decimal* value is a packed decimal number with an implicit decimal point. The position of the decimal point is determined by the precision and the scale of the number. The scale, which is the number of digits in the fractional part of the number, cannot be negative or greater than the precision. The maximum precision is 31 digits.

All values in a decimal column have the same precision and scale. The range of a decimal variable or the numbers in a decimal column is -*n* to +*n*, where the absolute value of *n* is the largest number that can be represented with the applicable precision and scale. The maximum range is $-10^{31}+1$ to $10^{31}-1$.

Single-precision floating-point (REAL)

A *single-precision floating-point* number is a 32-bit approximation of a real number. The number can be zero or can range from -3.4028234663852886e+38 to -1.1754943508222875e-38, or from 1.1754943508222875e-38 to 3.4028234663852886e+38.

Double-precision floating-point (DOUBLE or FLOAT)

A *double-precision floating-point* number is a 64-bit approximation of a real number. The number can be zero or can range from -1.7976931348623158e+308 to -2.2250738585072014e-308, or from 2.2250738585072014e-308 to 1.7976931348623158e+308.

Decimal floating-point (DECFLOAT)

A *decimal floating-point* value is an IEEE 754r number with a decimal point. The position of the decimal point is stored in each decimal floating-point value. The maximum precision is 34 digits. The range of a decimal floating-point number is either 16 or 34 digits of precision, and an exponent range of 10^{-383} to 10^{+384} or 10^{-6143} to 10^{+6144}, respectively. The minimum exponent, E_{min}, for DECFLOAT values is -383 for DECFLOAT(16) and -6143 for DECFLOAT(34). The maximum exponent, E_{max}, for DECFLOAT values is 384 for DECFLOAT(16) and 6144 for DECFLOAT(34).

In addition to finite numbers, decimal floating-point numbers are able to represent one of the following named decimal floating-point special values:

- Infinity - a value that represents a number whose magnitude is infinitely large
- Quiet NaN - a value that represents undefined results and that does not cause an invalid number warning
- Signalling NaN - a value that represents undefined results and that causes an invalid number warning if used in any numeric operation

When a number has one of these special values, its coefficient and exponent are undefined. The sign of an infinity value is significant, because it is possible to have positive or negative infinity. The sign of a NaN value has no meaning for arithmetic operations.

Subnormal numbers and underflow

Nonzero numbers whose adjusted exponents are less than E_{min} are called subnormal numbers. These subnormal numbers are accepted as operands for all operations and can result from any operation.

For a subnormal result, the minimum values of the exponent become E_{min} - (precision-1), called E_{tiny}, where precision is the working precision. If necessary, the result is rounded to ensure that the exponent is no smaller than E_{tiny}. If the result becomes inexact during rounding, an underflow warning is returned. A subnormal result does not always return the underflow warning.

When a number underflows to zero during a calculation, its exponent will be E_{tiny}. The maximum value of the exponent is unaffected.

The maximum value of the exponent for subnormal numbers is the same as the minimum value of the exponent that can arise during operations that do not result in subnormal numbers. This occurs when the length of the coefficient in decimal digits is equal to the precision.

Character strings

A *character string* is a sequence of code units. The length of the string is the number of code units in the sequence. If the length is zero, the value is called the *empty string*, which should not be confused with the null value.

Fixed-length character string (CHAR)

All values in a fixed-length string column have the same length, which is determined by the length attribute of the column. The length attribute must be in the range 1 - 255, inclusive, unless the string unit is CODEUNITS32, which has a range of 1 - 63, inclusive.

Varying-length character strings

There are two types of varying-length character strings:

VARCHAR
A VARCHAR value can be up to 32,672 bytes long. If the string unit is CODEUNITS32, the length can be up to 8,168 string units.

CLOB
A character large object (CLOB) value can be up to 2 gigabytes minus 1 byte (2,147,483,647 bytes) long or, if the string unit is CODEUNITS32, up to 536,870,911 string units. A CLOB is used to store large SBCS or mixed (SBCS and MBCS) character-based data (such as documents written with a single character set) and, therefore, has an SBCS or mixed code page that is associated with it.

Special restrictions apply to expressions that result in a CLOB data type, and to structured type columns; such expressions and columns are not permitted in:

- A SELECT list that is preceded by the DISTINCT clause
- A GROUP BY clause
- An ORDER BY clause
- A subselect of a set operator other than UNION ALL
- A basic, quantified, BETWEEN, or IN predicate
- An aggregate function
- VARGRAPHIC, TRANSLATE, and datetime scalar functions
- The pattern operand in a LIKE predicate, or the search string operand in a POSSTR function
- The string representation of a datetime value.

The functions in the SYSFUN schema taking a VARCHAR as an argument will not accept VARCHARs greater than 4,000 bytes long as an argument. However, many of these functions also have an alternative signature accepting a CLOB(1M). For these functions, the user can explicitly cast the greater than 4,000 VARCHAR strings into CLOBs and then recast the result back into VARCHARs of the required length.

NUL-terminated character strings that are found in C are handled differently, depending on the standards level of the precompile option.

Each character string is further defined as one of:

Bit data
Data that is not associated with a code page.

Single-byte character set (SBCS) data
Data in which every character is represented by a single byte.

Mixed data
Data that might contain a mixture of characters from a single-byte character set and a multi-byte character set (MBCS).

Unicode data
Data that contains characters that are represented by one or more bytes. Each Unicode character string is encoded by using UTF-8. The CCSID for UTF-8 is 1208.

Note: The LONG VARCHAR data type continues to be supported but is deprecated, not recommended, and might be removed in a future release.

String units specification for character strings

The unit of length for the character string data type is OCTETS or CODEUNITS32. The unit of length defines the counting method that is used to determine the length of the data.

OCTETS
Indicates that the units for the length attribute are bytes. This unit of length applies to all non-Unicode character string data types. For a Unicode character string data type, OCTETS can be explicitly specified or determined based on an environment setting.

CODEUNITS32
Indicates that the units for the length attribute are Unicode UTF-32 code units which approximate counting in characters. This unit of length does not affect the underlying code page of the data type. The actual length of a data value is determined by counting the UTF-32 code units as if the data was converted to UTF-32. A string unit of CODEUNITS32 can be used only for a Unicode character string data type. CODEUNITS32 can be explicitly specified or determined based on an environment setting.

For a non-Unicode character string data type, the string unit is always OCTETS and cannot be changed. For a Unicode character string data type, the string units can be explicitly specified with the length attribute of a character string data type, or it can default based on an environment setting. If FOR BIT DATA is also specified for the character string data type, CODEUNITS32 cannot be specified and an environment setting of CODEUNITS32 does not apply.

The environment setting for string units is based on the value for the NLS_STRING_UNITS global variable, or the **string_units** database configuration parameter. The database configuration parameter can be set to either SYSTEM or CODEUNITS32. The global variable can also be set to either SYSTEM or CODEUNITS32, but also can be set to NULL. The NULL value indicates that the SQL session should use the **string_units** database configuration parameter setting. If the value for the environment setting is SYSTEM, then OCTETS is used as the default string units setting.

String units in built-in functions

The ability to specify string units for certain built-in string functions allows you to process string data in a more "character-based manner" than a "byte-based manner". The *string unit* determines the unit that is used for length or position when you execute the function. You can specify CODEUNITS16, CODEUNITS32, or OCTETS as the string unit for some string functions. When no string unit is specified, the default string unit is usually determined by the string units of the source string argument (refer to the description of the function that you are using for details). The string units argument can be specified for string functions that support the parameter in Unicode or non-Unicode databases.

CODEUNITS16
Specifies that Unicode UTF-16 is the unit for the operation. CODEUNITS16 is useful when an application is processing data in code units that are 2 bytes in width. Note some characters, which are known as *supplementary characters*, require two UTF-16 code units to be encoded. For example, the musical symbol G clef requires two UTF-16 code units (X'D834' and X'DD1E' in UTF-16BE).

CODEUNITS32
Specifies that Unicode UTF-32 is the unit for the operation. CODEUNITS32 is useful for applications that process data in a simple, fixed-length format, and that must return the same answer regardless of the storage format of the data (ASCII, UTF-8, or UTF-16).

Note: The storage on disk uses UTF-8 encoding.

OCTETS
Specifies that bytes are the units for the operation. OCTETS is often used when an application is interested in allocating buffer space or when operations need to use simple byte processing.

The calculated length of a string computed using OCTETS (bytes) might differ from that computed using CODEUNITS16 or CODEUNITS32. When you are using OCTETS, the length of the string is determined by simply counting the number of bytes in the string. When you are using CODEUNITS16 or CODEUNITS32, the length of the string is determined by counting the number of 16-bit or 32-bit code units necessary to represent the string in UTF-16 or UTF-32. A length that is determined using CODEUNITS16 and CODEUNITS32 is identical unless the data contains supplementary characters (see “Difference between CODEUNITS16 and CODEUNITS32” on page 34).

For example, assume that NAME, a VARCHAR(128) column that is encoded in Unicode UTF-8, contains the value 'Jürgen'. The following two queries, which count the length of the string in CODEUNITS16 or CODEUNITS32, return the same value (6).

```
SELECT CHARACTER_LENGTH(NAME,CODEUNITS16) FROM T1
  WHERE NAME = 'Jürgen'
```

```
SELECT CHARACTER_LENGTH(NAME,CODEUNITS32) FROM T1
  WHERE NAME = 'Jürgen'
```

The next query, which counts the length of the string in OCTETS, returns the value 7.

```
SELECT CHARACTER_LENGTH(NAME,OCTETS) FROM T1
  WHERE NAME = 'Jürgen'
```

These values represent the length of the string that is expressed in the specified string unit.

The following table shows the UTF-8, UTF-16BE (big-endian), and UTF-32BE (big-endian) representations of the name 'Jürgen':

```
Format    Representation of the name 'Jürgen'
--------  ------------------------------------
UTF-8      X'4AC3BC7267656E'
UTF-16BE     X'004A00FC007200670065006E'
UTF-32BE     X'0000004A000000FC00000072000000670000006500000006E'
```

The representation of the character 'ü' differs among the three string units:

- The UTF-8 representation of the character 'ü' is X'C3BC'.
- The UTF-16BE representation of the character 'ü' is X'00FC'.
- The UTF-32BE representation of the character 'ü' is X'000000FC'.

Specifying string units for a built-in function does not affect the data type, the string units, or the code page of the result of the function. If necessary, the data is converted to Unicode for evaluation when CODEUNITS16 or CODEUNITS32 is specified.

When OCTETS is specified for the LOCATE or POSITION function, and the code pages of the string arguments differ, the data is converted to the code page of the *source-string* argument. In this case, the result of the function is in the code page of the *source-string* argument. When OCTETS is specified for functions that take a single string argument, the data is evaluated in the code page of the string argument, and the result of the function is in the code page of the string argument.

Difference between CODEUNITS16 and CODEUNITS32

When CODEUNITS16 or CODEUNITS32 is specified, the result is the same except when the data contains Unicode supplementary characters. This is because a supplementary character is represented by two UTF-16 code units or one UTF-32 code unit. In UTF-8, a non-supplementary character is represented by 1 to 3 bytes, and a supplementary character is represented by 4 bytes. In UTF-16, a non-supplementary character is represented by one CODEUNITS16 code unit or 2 bytes, and a supplementary character is represented by two CODEUNITS16 code units or 4 bytes. In UTF-32, a character is represented by one CODEUNITS32 code unit or 4 bytes.

For example, the following table shows the hexadecimal values for the mathematical bold capital A and the Latin capital letter A. The mathematical bold capital A is a supplementary character that is represented by 4 bytes in UTF-8, UTF-16, and UTF-32.

Character	UTF-8 representation	UTF-16BE representation	UTF-32BE representation
Unicode value X'1D400' - 'A'; mathematical bold capital A	X'F09D9080'	X'D835DC00'	X'0001D400'
Unicode value X'0041' - 'A'; latin capital letter A	X'41'	X'0041'	X'00000041'

Assume that C1 is a VARCHAR(128) column, encoded in Unicode UTF-8, and that table T1 contains one row with the value of the mathematical bold capital A (X'F09D9080'). The following queries return different results:

```
Query                                              Returns
-----                                              -------
SELECT CHARACTER_LENGTH(C1,CODEUNITS16) FROM T1     2

SELECT CHARACTER_LENGTH(C1,CODEUNITS32) FROM T1     1

SELECT CHARACTER_LENGTH(C1,OCTETS) FROM T1          4
```

Graphic strings

A *graphic string* is a sequence of code units that represents double-byte character data.

The length of the string is the number of code units in the sequence. If the length is zero, the value is called the *empty string*. This value should not be confused with the null value.

Graphic strings are not supported in a database that is defined with a single-byte code page.

Graphic strings are not checked to ensure that their values contain only double-byte character code points. (The exception to this rule is an application that is precompiled with the WCHARTYPE CONVERT option. In this case, validation does occur.) Rather, the database manager assumes that double-byte character data is contained in graphic data fields. The database manager *does* check that a graphic string value is an even number of bytes long.

NUL-terminated graphic strings that are found in C are handled differently, depending on the standards level of the precompile option. This data type cannot be created in a table. It can be used only to insert data into and retrieve data from the database.

Fixed-length graphic strings (GRAPHIC)

All values in a fixed-length graphic string column have the same length, which is determined by the length attribute of the column. The length attribute must be 1 - 127, inclusive, unless the string unit is CODEUNITS32 which has a range of 1 - 63, inclusive.

Varying-length graphic strings

There are two types of varying-length graphic string:

- A VARGRAPHIC value can be up to 16 336 double-byte code units long. If the string unit is CODEUNITS32, the length can be up to 8 168 string units.
- A DBCLOB (double-byte character large object) value can be up to 1 073 741 823 double-byte code units long. If the string unit is CODEUNITS32, the length can be up to 536 870 911 string units. A DBCLOB is used to store large DBCS character-based data (such as documents written with a single character set) and, therefore, has a DBCS code page that is associated with it.

Special restrictions apply to an expression that results in a DBCLOB data type. These restrictions are the same as the restrictions specified in “Varying-length character strings” on page 31.

Note: The LONG VARGRAPHIC data type continues to be supported but is deprecated, not recommended, and might be removed in a future release.

String units specification for graphic strings

The unit of length for a graphic string data type is double bytes, CODEUNITS16, or CODEUNITS32. The unit of length defines the counting method that is used to determine the length of the data.

Double bytes
Indicates that the units for the length attribute are double bytes. This unit of length applies to all graphic string data types in a non-Unicode database. In a Unicode database, CODEUNITS16 is used.

CODEUNITS16
Indicates that the units for the length attribute are Unicode UTF-16 code units which are the same as counting in double bytes. This unit of length does not affect the underlying code page of the data type. A string unit of CODEUNITS16 can be used only with graphic string data types in a Unicode database. CODEUNITS16 can be explicitly specified or determined based on an environment setting.

CODEUNITS32
Indicates that the units for the length attribute are Unicode UTF-32 code units which approximate counting in characters. This unit of length does not affect the underlying code page of the data type. The actual length of a data value is determined by counting the UTF-32 code units as if the data was converted to UTF-32. A string unit of CODEUNITS32 can be used only in a Unicode database. CODEUNITS32 can be explicitly specified or determined based on an environment setting.

In a non-Unicode database, the string unit is always double bytes and cannot be changed. In a Unicode database, the string units can be explicitly specified with the length attribute of a graphic string data type, or it can default based on an environment setting.

The environment setting for string units is based on the value for the NLS_STRING_UNITS global variable, or the **string_units** database configuration parameter. The database configuration parameter can be set to either SYSTEM or CODEUNITS32. The global variable can also be set to either SYSTEM or CODEUNITS32, but also can be set to NULL. The NULL value indicates that the SQL session should use the **string_units** database configuration parameter setting. If the value for the environment setting is SYSTEM, then CODEUNITS16 is used as the default string units setting in a Unicode database and double bytes is used in a non-Unicode database.

National character strings

A national character string is a sequence of bytes that represents character data in UTF-8 or UTF-16 BE encoding.

National character strings are only allowed in a Unicode database.

The length of the string is the number of code units in the sequence. If the length is zero, the value is called the empty string. This value should not be confused with the null value.

National character strings are synonyms for Unicode character strings or graphic strings. Based on the value that is set for **nchar_mapping** database configuration parameter, national character strings are mapped as follows:

nchar_mapping is CHAR_CU32:

- NCHAR is a synonym for CHARACTER with string units CODEUNITS32
- NVARCHAR is a synonym for VARCHAR with string units CODEUNITS32
- NCLOB is a synonym for CLOB with string units CODEUNITS32

nchar_mapping is GRAPHIC_CU32:

- NCHAR is a synonym for GRAPHIC with string units CODEUNITS32
- NVARCHAR is a synonym for VARGRAPHIC with string units CODEUNITS32
- NCLOB is a synonym for DBCLOB with string units CODEUNITS32

nchar_mapping is GRAPHIC_CU16:

- NCHAR is a synonym for GRAPHIC with string units CODEUNITS16
- NVARCHAR is a synonym for VARGRAPHIC with string units CODEUNITS16
- NCLOB is a synonym for DBCLOB with string units CODEUNITS16

For details, refer to the topics "Character strings" and "Graphic strings".

Binary strings

A *binary string* is a sequence of bytes. Unlike character strings, which usually contain text data, binary strings are used to hold data such as pictures, voice, or mixed media.

Binary strings are not associated with a code page; their code page value is 0. The length of a binary string is the number of bytes it contains. Only character strings of the FOR BIT DATA subtype are compatible with binary strings.

The unit of length for the binary string data type is OCTETS and cannot be explicitly specified.

Fixed-length binary string

A fixed-length binary string has the data type BINARY. All values in a fixed-length string column have the same length, which is determined by the length attribute of the data type. The length attribute must be in the range 1 - 255, inclusive.

Varying-length binary strings

There are two types of varying-length binary string:

VARBINARY
A VARBINARY value can be up to 32,672 bytes long.

BLOB
A binary large object (BLOB) value can be up to 2 gigabytes minus 1 byte (2,147,483,647 bytes) long. A BLOB can hold structured data for exploitation by user-defined types and user-defined functions.

Special restrictions apply to an expression that results in a BLOB data type. These restrictions are the same as the restrictions described in "Varying-length character strings" on page 31.

Large objects (LOBs)

The term *large object* and the generic acronym LOB refer to the BLOB, CLOB, or DBCLOB data type. In a Unicode database, NCLOB can be used as a synonym for DBCLOB.

LOB values are subject to restrictions, as described in "Varying-length character strings" on page 31. These restrictions apply even if the length attribute of the LOB string is 254 bytes or less.

LOB values can be very large, and the transfer of these values from the database server to client application program host variables can be time consuming. Because application programs typically process LOB values one piece at a time, rather than as a whole, applications can reference a LOB value by using a large object locator.

A *large object locator*, or LOB locator, is a host variable whose value represents a single LOB value on the database server.

An application program can select a LOB value into a LOB locator. Then, using the LOB locator, the application program can request database operations on the LOB value (such as applying the scalar functions SUBSTR, CONCAT, VALUE, or LENGTH; performing an assignment; searching the LOB with LIKE or POSSTR; or applying user-defined functions against the LOB) by supplying the locator value as input. The resulting output (data assigned to a client host variable) would typically be a small subset of the input LOB value.

LOB locators can represent more than just base values; they can also represent the value associated with a LOB expression. For example, a LOB locator might represent the value associated with:

```
SUBSTR( <lob 1> CONCAT <lob 2> CONCAT <lob 3>, <start>, <length> )
```

When a null value is selected into a normal host variable, the indicator variable is set to -1, signifying that the value is null. In the case of LOB locators, however, the meaning of indicator variables is slightly different. Because a locator host variable can itself never be null, a negative indicator variable value indicates that the LOB value represented by the LOB locator is null. The null information is kept local to the client by virtue of the indicator variable value - the server does not track null values with valid locators.

It is important to understand that a LOB locator represents a value, not a row or a location in the database. Once a value is selected into a locator, there is no operation that one can perform on the original row or table that will affect the value which is referenced by the locator. The value associated with a locator is valid until the transaction ends, or until the locator is explicitly freed, whichever comes first. Locators do not force extra copies of the data to provide this function. Instead, the locator mechanism stores a description of the base LOB value. The materialization of the LOB value (or expression, as shown previously) is deferred until it is actually assigned to some location - either a user buffer in the form of a host variable, or another record in the database.

A LOB locator is only a mechanism used to refer to a LOB value during a transaction; it does not persist beyond the transaction in which it was created. It is not a database type; it is never stored in the database and, as a result, cannot participate in views or check constraints. However, because a LOB locator is a client representation of a LOB type, there are SQLTYPEs for LOB locators so that they can be described within an SQLDA structure used by FETCH, OPEN, or EXECUTE statements.

Datetime values

The datetime data types are DATE, TIME, and TIMESTAMP. Although datetime values can be used in certain arithmetic and string operations and are compatible with certain strings, they are not strings or numbers.

Date

A *date* is a three-part value (year, month, and day):

- The range of the month part is 1 - 12.
- The range of the day part is 1 - *x*, where *x* is 28, 29, 30, or 31, and depends on the month.
- The range of the year part is 0001 - 9999 for local tables.

The internal representation of a date is a string of 4 bytes. Each byte consists of 2 packed decimal digits. The first 2 bytes represent the year, the third byte the month, and the last byte the day.

The length of a DATE column, as described in the SQLDA, is 10 bytes, which is the appropriate length for a character string representation of the value.

Time

A *time* is a three-part value (hour, minute, and second) designating a time of day under a 24-hour clock:

- The range of the hour part is 0 - 24.
- The range of the minute and second parts is 0 - 59.

If the hour is 24, the minute and second specifications are 0.

Important: Using the value 24 when representing hours in a time value might result in errors or unexpected data. To avoid this issue, use 00 instead of 24.

The internal representation of a time is a string of 3 bytes. Each byte consists of 2 packed decimal digits. The first byte represents the hour, the second byte the minute, and the last byte the second.

The length of a TIME column, as described in the SQLDA, is 8 bytes, which is the appropriate length for a character string representation of the value.

Timestamp

A *timestamp* is a six or seven-part value (year, month, day, hour, minute, second, and optional fractions of a second) designating a date and time as defined in the previous sections, except that the time can also include an additional part designating a fraction of a second. The number of digits in the fractional seconds is specified using an attribute in the range from 0 to 12; the default is 6.

Important: Using the value 24 when representing hours in a timestamp value might result in errors or unexpected data. To avoid this issue, use 00 instead of 24.

The internal representation of a timestamp is a string of 7 - 13 bytes. Each byte consists of 2 packed decimal digits. The first 4 bytes represent the date, the next 3 bytes the time, and the last 0 - 6 bytes the fractions of a second.

The length of a TIMESTAMP column, as described in the SQLDA, is 19 - 32 bytes, which is the appropriate length for the character string representation of the value.

String representations of datetime values

Values whose data types are DATE, TIME, or TIMESTAMP are represented in an internal form that is transparent to the user. Date, time, and timestamp values can, however, also be represented by strings. This is useful because there are no constants or variables whose data types are DATE, TIME, or TIMESTAMP. Before it can be retrieved, a datetime value must be assigned to a string variable. The CHAR function or the GRAPHIC function (for Unicode databases only) can be used to change a datetime value to a string representation. The string representation is normally the default format of datetime values associated with the territory code of the application, unless overridden by specification of the DATETIME option when the program is precompiled or bound to the database.

No matter what its length, a large object string cannot be used as a string representation of a datetime value (SQLSTATE 42884).

When a valid string representation of a datetime value is used in an operation with an internal datetime value, the string representation is converted to the internal form of the date, time, or timestamp value before the operation is performed.

Date, time and timestamp strings must contain only characters and digits.

Date strings

Table 3. Formats for String Representations of Dates

Format Name	Abbreviation	Date Format	Example
International Standards Organization (with separators)	ISO	yyyy-mm-dd	'2018-10-27'
International Standards Organization (without separators)[1]	-	yyyymmdd	'20181027'
IBM USA standard	USA	mm/dd/yyyy	'10/27/2018'
IBM European standard	EUR	dd.mm.yyyy	'27.10.2018'
Japanese Industrial Standard Christian Era	JIS	yyyy-mm-dd	'2018-10-27'
Netezza®[1]	-	dd-mon-yy	'12-FEB-16'
Site-defined	LOC	Depends on the territory code of the application	-

[1] This format can be used for input values only, not for output values or constants.

Note:

- Trailing blanks can be included.
- Leading zeros can be omitted from the month and day portions.

Time strings

Table 4. Formats for String Representations of Times			
Format Name	**Abbreviation**	**Time Format**	**Example**
International Standards Organization	ISO	hh.mm or hh.mm.ss	'13.30.05'
IBM USA standard	USA	hh or hh:mm AM or PM	'1:30 PM'
IBM European standard	EUR	hh.mm or hh.mm.ss	'13.30.05'
Japanese Industrial Standard Christian Era	JIS	hh:mm or hh:mm:ss	'13:30:05'
Site-defined	LOC	Depends on the territory code of the application	-

Note:

- Trailing blanks can be included.
- A leading zero can be omitted from the hour.
- Seconds can be omitted, in which case an implicit specification of 0 seconds is assumed (for example, 13:30 is equivalent to 13:30:00).
- The International Standards Organization changed the time format so that it is identical to the Japanese Industrial Standard Christian Era format. Therefore, use the JIS format if an application requires the current International Standards Organization format.
- In the USA format:
  - Minutes can be omitted, in which case an implicit specification of 00 minutes is assumed (for example, 1 PM is equivalent to 1:00 PM).
  - The hour must not be greater than 12 and cannot be 0, except in the special case of 00:00 AM.
  - There is a single space before 'AM' or 'PM'.
  - 'AM' or 'PM' can be represented in lowercase or uppercase characters.
- The following table shows how the USA format corresponds to the JIS format:

USA format	**JIS format**
12:01 AM through 12:59 AM	00:01:00 through 00:59:00
01:00 AM through 11:59 AM	01:00:00 through 11:59:00
12:00 PM (noon) through 11:59 PM	12:00:00 through 23:59:00
12:00 AM (midnight)	24:00:00
00:00 AM (midnight)	00:00:00

Timestamp strings

Table 5. Formats for String Representations of Timestamps		
Format Name	**Timestamp Format**	**Example**
IBM SQL	yyyy-mm-dd-hh.mm.ss.nnnnnnnnnnnn	'2018-03-22-12.00.00.000000000005'

Table 5. Formats for String Representations of Timestamps (continued)		
Format Name	**Timestamp Format**	**Example**
ODBC[1]	yyyy-mm-dd hh:mm:ss.nnnnnnnnnnnn	'2018-03-22 08:30:58.000000000005'
Netezza	yyyymmdd hh:mm:ss AM or PM	'20180101 12:00:59 PM'
Netezza[1]	mm-dd-yyyy hh:mm:ss.nnnnnnnnnnnn	'05-18-2020 18:10:16.123456123456'
No delimiters	yyyymmddhhmmss	'20180322120000'
[1] The date and time portions of the timestamp are shown here as being separated by a blank. They can also be separated by a hyphen or the letter T. For example, '2018-03-22 08:30:58.7', '2018-03-22-08:30:58.7', and '2018-03-22T08:30:58.7' are all equivalent.		

Note:

- IBM SQL is the only supported output format
- Seconds can be specified to up to 12 decimal places.
- Trailing zeros can be truncated or entirely omitted from the fractional seconds.
- Leading zeros can be omitted from the month, day, and hour part of the timestamp.
- A character string can contain any number of trailing blanks.
- The separator character that follows the seconds element can be omitted if fractional seconds are not included.
- If a string representation of a timestamp is implicitly cast to a value with a TIMESTAMP data type, the precision of the result of the cast is determined by the precision of the TIMESTAMP operand in an expression or the precision of the TIMESTAMP target in an assignment:
  - Digits for fractional seconds that exceed the precision of the cast are truncated from the string. For example, if the string '2018-3-2-8.30.00297' is cast to TIMESTAMP(3), the result is 2018-03-02-08.30.00.002.
  - If the precision of the cast exceeds the precision of the string, the result is padded with zeros. For example, if the string '2018-3-2-8.30.07' is cast to TIMESTAMP(12), the result is 2018-03-02-08.30.00.070000000000.
- A string representation of a timestamp can be given a different timestamp precision by explicitly casting the value to a timestamp with a specified precision. If the string is a constant, an alternative is to precede the string constant with the TIMESTAMP keyword. For example, TIMESTAMP '2018-03-28 14:50:35.123' has the TIMESTAMP(3) data type.

Boolean values

A Boolean value represents a truth value; that is, TRUE or FALSE. A Boolean expression or predicate can result in a value of unknown, which is represented by the null value.

The following data types can be cast to the BOOLEAN data type:

- CHAR or VARCHAR can be cast to a BOOLEAN value:
  - Cast to TRUE: 't' , 'true' , 'y', 'yes' , 'on' ,'1'
  - Cast to FALSE: 'f', 'false', 'n', 'no', 'off', '0'

  A string can use any combination of uppercase and lowercase characters ('yes', 'YES', 'Yes', 'yES', and so on).
- decimal floating point or binary integer

– When a value of data type DECFLOAT, SMALLINT, INTEGER, or BIGINT is cast to a BOOLEAN value, the result is TRUE if the value is not zero, and FALSE if the value is zero.

Cursor values

A cursor value is used to represent a reference to an underlying cursor.

The CURSOR type is a built-in data type that can only be used as the data type of:

- A local variable in a compound SQL (compiled) statement
- A parameter of an SQL routine
- The returns type of an SQL function
- A global variable

A variable or parameter defined with the CURSOR type can only be used in compound SQL (compiled) statements.

A cursor variable is an SQL variable, SQL parameter, or global variable of a cursor type. A cursor variable is said to have an underlying cursor that corresponds to the cursor created for a SELECT statement and assigned to that variable. More than one cursor variable may share the same underlying cursor.

Cursor variables can be used the same way as conventional SQL cursors to iterate through a result set of a SELECT statement with OPEN, FETCH, and CLOSE statements.

XML values

An XML value represents well-formed XML in the form of an XML document, XML content, or a sequence of XML nodes.

An XML value that is stored in a table as a value of a column defined with the XML data type must be a well-formed XML document. XML values are processed in an internal representation that is not comparable to any string value. An XML value can be transformed into a serialized string value representing the XML document using the XMLSERIALIZE function. Similarly, a string value that represents an XML document can be transformed into an XML value using the XMLPARSE function. An XML value can be implicitly parsed or serialized when exchanged with application string and binary data types.

Special restrictions apply to expressions that result in an XML data type value; such expressions and columns are not permitted in (SQLSTATE 42818):

- A SELECT list preceded by the DISTINCT clause
- A GROUP BY clause
- An ORDER BY clause
- A subselect of a set operator other than UNION ALL
- A basic, quantified, BETWEEN, IN, or LIKE predicate
- An aggregate function with DISTINCT

Array values

An *array* is a structure that contains an ordered collection of data elements in which each element can be referenced by its index value in the collection.The *cardinality* of an array is the number of elements in the array. All elements in an array have the same data type.

An *ordinary array* has a defined upper bound on the number of elements, known as the maximum cardinality. Each element in the array is referenced by its ordinal position as the index value. If *N* is the number of elements in an ordinary array, the ordinal position associated with each element is an integer value greater than or equal to 1 and less than or equal to *N*.

An *associative array* has no specific upper bound on the number of elements. Each element is referenced by its associated index value. The data type of the index value can be an integer or a character string but is the same data type for the entire array.

The maximum cardinality of an ordinary array is not related to its physical representation, unlike the maximum cardinality of arrays in programming languages such as C. Instead, the maximum cardinality is used by the system at run time to ensure that subscripts are within bounds. The amount of memory required to represent an ordinary array value is not proportional to the maximum cardinality of its type.

The amount of memory required to represent an array value is usually proportional to its cardinality. When an array is being referenced, all of the values in the array are stored in main memory. Therefore, arrays that contain a large amount of data will consume large amounts of main memory.

The Array type is not supported for multi-row insert, update, or delete.

Anchored types

An anchored type defines a data type based on another SQL object such as a column, global variable, SQL variable, SQL parameter, or the row of a table or view.

A data type defined using an anchored type definition maintains a dependency on the object to which it is anchored. Any change in the data type of the anchor object will impact the anchored data type. If anchored to the row of a table or view, the anchored data type is ROW with the fields defined by the columns of the anchor table or anchor view.

User-defined types

A user-defined data type (UDT) is a data type that derived from an existing data type. You can use UDTs to extend the built-in types already available and create your own customized data types.

There are six user-defined types:

- Distinct type
- Structured type
- Reference type
- Array type
- Row type
- Cursor type

Each of these types is described in the following sections.

Distinct type

A *distinct type* is a user-defined data type that shares its internal representation with an existing built-in data type (its "source" type).

Distinct types include qualified identifiers. If the schema name is not used to qualify the distinct type name when used in other than the CREATE TYPE (Distinct), DROP, or COMMENT statements, the SQL path is searched in sequence for the first schema with a distinct type that matches.

Distinct types that are sourced on LOB types are subject to the same restrictions as their source type.

A distinct type is defined to use either strong typing or weak typing rules. Strong typing rules are the default.

Strongly typed distinct type

A strongly typed distinct type is considered to be a separate and incompatible type for most operations. For example, you want to define a picture type, a text type, and an audio type. Each of these types has different semantics, but each uses the built-in data type BLOB for their internal representation.

The following example illustrates the creation of a distinct type named AUDIO:

```
CREATE TYPE AUDIO  AS BLOB (1M)
```

Although AUDIO has the same representation as the built-in data type BLOB, it is considered to be a separate type; this consideration allows the creation of functions that are written specifically for

AUDIO, and assures that these functions are not applied to values of any other data type (for example pictures or text).

Strongly typed distinct types support strong typing by ensuring that only those functions and operators that are explicitly defined on the distinct type can be applied to its instances. For this reason, a strongly typed distinct type does not automatically acquire the functions and operators of its source type, because these functions and operators might not be meaningful. For example, a LENGTH function could be defined to support a parameter with the data type AUDIO that returns length of the object in seconds instead of bytes.

Weakly typed distinct type
A weakly typed distinct type is considered to be the same as its source type for all operations, except when the weakly typed distinct type applies constraints on values during assignments or casts. This consideration also applies to function resolution.

The following example illustrates the creation of a distinct type named POSITIVEINTEGER:

```
CREATE TYPE POSITIVEINTEGER AS INTEGER
      WITH WEAK TYPE RULES CHECK (VALUE>=0)
```

Weak typing means that except for accepting only positive integer values, POSITIVEINTEGER operates in the same way as its underlying data type of INTEGER.

A weakly typed distinct type can be used as an alternative method of referring to a built-in data type within application code. The ability to define constraints on the values that are associated with the distinct type provides a method for checking values during assignments and casts.

Using distinct types can provide benefits in the following categories:

Extensibility
By defining new data types, you increase the set of data types available to you to support your applications.

Flexibility
You can specify any semantics and behavior for your new data type by using user-defined functions (UDFs) to augment the diversity of the data types available in the system.

Consistent and inherited behavior
Strong typing guarantees that only functions defined on your distinct type can be applied to instances of the distinct type. Weak typing ensures that the distinct type behaves the same way as its underlying data type and so can use all the same functions and methods available to that underlying type.

Encapsulation
Using a weakly typed distinct type makes it possible to define data type constraints in one location for all usages within application code for that distinct type.

Performance
Distinct types are highly integrated into the database manager. Because distinct types are internally represented the same way as built-in data types, they share the same efficient code that is used to implement components such as built-in functions, comparison operators, and indexes for built-in data types.

Not all built-in data types can be used to define distinct types. The source data type cannot be XML, array, row, or cursor. For more information, see "CREATE TYPE (distinct) statement" in *SQL Reference Volume 1*.

Structured type

A *structured type* is a user-defined data type that has a structure that is defined in the database. It contains a sequence of named *attributes*, each of which has a data type. A structured type also includes a set of method specifications.

A structured type can be used as the type of a table, view, or column. When used as a type for a table or view, that table or view is known as a *typed table* or *typed view*. For typed tables and typed views, the names and data types of the attributes of the structured type become the names and data types of the columns of this typed table or typed view. Rows of the typed table or typed view can be thought of as a

representation of instances of the structured type. When used as a data type for a column, the column contains values of that structured type (or values of any of the subtypes for that type, as defined later in this section). Methods are used to retrieve or manipulate attributes of a structured column object.

A *supertype* is a structured type for which other structured types, called *subtypes*, are defined. A subtype inherits all the attributes and methods of its supertype and can have additional attributes and methods defined. The set of structured types that is related to a common supertype is called a type *hierarchy* and the type that does not have any supertype is called the *root type* of the type hierarchy.

The term subtype applies to a user-defined structured type and all user-defined structured types that are below it in the type hierarchy. Therefore, a subtype of a structured type T is T and all structured types below T in the hierarchy. A *proper subtype* of a structured type T is a structured type below T in the type hierarchy.

There are restrictions on having recursive type definitions in a type hierarchy. For this reason, it is necessary to develop a shorthand way of referring to the specific type of recursive definitions that are allowed. The following definitions are used:

- *Directly uses*: A type **A** is said to directly use another type **B**, if and only if one of the following statements is true:
  1. Type **A** has an attribute of type **B**.
  2. Type **B** is a subtype of **A**or a supertype of **A**.
- *Indirectly uses*: A type **A** is said to indirectly use a type **B**, if one of the following statements is true:
  1. Type **A** directly uses type **B**.
  2. Type **A** directly uses some type **C**and type **C** indirectly uses type **B**.

A type cannot be defined so that one of its attribute types directly or indirectly uses itself. If it is necessary to have such a configuration, consider using a reference as the attribute. For example, with structured type attributes, there cannot be an instance of "employee" with an attribute of "manager" when "manager" is of type "employee". There can, however, be an attribute of "manager" with a type of REF(employee).

A type cannot be dropped if certain other objects use the type, either directly or indirectly. For example, a type cannot be dropped if a table or view column makes direct or indirect use of the type.

Reference type

A *reference type* is a companion type to a structured type. Similar to a distinct type, a reference type is a scalar type that shares a common representation with one of the built-in data types. This same representation is shared for all types in the type hierarchy. The reference type representation is defined when the root type of a type hierarchy is created. When using a reference type, a structured type is specified as a parameter of the type. This parameter is called the *target type* of the reference.

The target of a reference is always a row in a typed table or a typed view. When a reference type is used, it can have a *scope* defined. The scope identifies a table (called the *target table*) or view (called the *target view*) that contains the target row of a reference value. The target table or view must have the same type as the target type of the reference type. An instance of a scoped reference type uniquely identifies a row in a typed table or typed view, called the *target row*.

Array type

A user-defined *array type* is a data type that is defined as an array with elements of another data type. Every ordinary array type has an index with the data type of INTEGER and has a defined maximum cardinality. Every associative array has an index with the data type of INTEGER or VARCHAR and does not have a defined maximum cardinality.

Row type

A *row type* is a data type that is defined as an ordered sequence of named fields, each with an associated data type, which effectively represents a row. A row type can be used as the data type for variables and parameters in SQL PL to provide simple manipulation of a row of data.

Cursor type

A user-defined *cursor type* is a user-defined data type defined with the keyword CURSOR and optionally with an associated row type. A user-defined cursor type with an associated row type is a *strongly typed cursor type*; otherwise, it is a *weakly typed cursor type*. A value of a user-defined cursor type represents a reference to an underlying cursor.

Promotion of data types

Data types can be classified into groups of related data types. Within such groups, a precedence order exists where one data type is considered to precede another data type. This precedence is used to allow the *promotion* of one data type to a data type later in the precedence ordering.

For example, the data type CHAR can be promoted to VARCHAR; INTEGER can be promoted to DOUBLE-PRECISION; but CLOB is NOT promotable to VARCHAR.

Promotion of data types is used when:

- Performing function resolution
- Casting user-defined types
- Assigning user-defined types to built-in data types

Table 6 on page 46 shows the precedence list (in order) for each data type and can be used to determine the data types to which a given data type can be promoted. The table shows that the best choice is always the same data type instead of choosing to promote to another data type.

Table 6. Data Type Precedence Table

Data Type	Data Type Precedence List (in best-to-worst order)
SMALLINT	SMALLINT, INTEGER, BIGINT, decimal, real, double, DECFLOAT
INTEGER	INTEGER, BIGINT, decimal, real, double, DECFLOAT
BIGINT	BIGINT, decimal, real, double, DECFLOAT
decimal	decimal, real, double, DECFLOAT
real	real, double, DECFLOAT
double	double, DECFLOAT
DECFLOAT	DECFLOAT
CHAR	CHAR, VARCHAR, CLOB
VARCHAR	VARCHAR, CLOB
CLOB	CLOB
GRAPHIC	GRAPHIC, VARGRAPHIC, DBCLOB
VARGRAPHIC	VARGRAPHIC, DBCLOB
DBCLOB	DBCLOB
BINARY	BINARY, VARBINARY, BLOB
VARBINARY	VARBINARY, BLOB

Table 6. Data Type Precedence Table (continued)

Data Type	Data Type Precedence List (in best-to-worst order)
BLOB	BLOB
DATE	DATE, TIMESTAMP
TIME	TIME
TIMESTAMP	TIMESTAMP
BOOLEAN	BOOLEAN
CURSOR	CURSOR
ARRAY	ARRAY
udt	udt (same name) or a supertype of udt
REF(T)	REF(S) (provided that S is a supertype of T)
ROW	ROW

Note:

1. The lowercase types in the preceding table are defined as follows:
   - decimal = DECIMAL(p,s) or NUMERIC(p,s)
   - real = REAL or FLOAT(*n*), where *n* is not greater than 24
   - double = DOUBLE, DOUBLE-PRECISION, FLOAT or FLOAT(*n*), where *n* is greater than 24
   - udt = a user-defined type (except for a weakly typed distinct type which uses the source type to determine data type precedence)

   Shorter and longer form synonyms of the listed data types are considered to be the same as the listed form.
2. For a Unicode database, the following data types are considered to be equivalent:
   - CHAR and GRAPHIC
   - VARCHAR and VARGRAPHIC
   - CLOB and DBCLOB

   When resolving a function within a Unicode database, if a user-defined function and a built-in function are both applicable for a given function invocation, then generally the built-in function will be invoked. The UDF will be invoked only if its schema precedes SYSIBM in the **CURRENT PATH** special register and if its argument data types match all the function invocation argument data types, regardless of Unicode data type equivalence.

Casting between data types

There are many occasions where a value with a given data type needs to be *cast* to a different data type or to the same data type with a different length, precision, or scale.

Data type promotion is one example where the promotion of one data type to another data type requires that the value be cast to the new data type. A data type that can be cast to another data type is *castable* from the source data type to the target data type.

The casting of one data type to another can occur implicitly or explicitly. The cast functions, CAST specification, or XMLCAST specification can be used to explicitly change a data type, depending on the data types involved. In addition, when a sourced user-defined function is created, the data types of the parameters of the source function must be castable to the data types of the function that is being created.

The supported casts between built-in data types are shown in Table 7 on page 49. The first column represents the data type of the cast operand (source data type), and the data types across the header

row represent the target data type of the cast operation. A 'Y' indicates that the CAST specification can be used for the combination of source and target data types. Cases in which only the XMLCAST specification can be used are noted.

If truncation occurs when any data type is cast to a character or graphic data type, a warning is returned if any non-blank characters are truncated. This truncation behavior is unlike the assignment to a character or graphic data type, when an error occurs if any non-blank characters are truncated.

In a Unicode database, character and graphic string source values can be cast to between different string units. Any truncation is applied according to the string units of the target data type.

The following casts involving strongly typed distinct types are supported (using the CAST specification unless noted otherwise):

- Cast from distinct type *DT* to its source data type *S*
- Cast from the source data type *S* of distinct type *DT* to distinct type *DT*
- Cast from distinct type *DT* to the same distinct type *DT*
- Cast from a data type *A* to distinct type *DT* where *A* is promotable to the source data type *S* of distinct type *DT*
- Cast from an INTEGER to distinct type *DT* with a source data type SMALLINT
- Cast from a DOUBLE to distinct type *DT* with a source data type REAL
- Cast from a DECFLOAT to distinct type DT with a source data type of DECFLOAT
- Cast from a VARBINARY to distinct type *DT* with a source data type BINARY
- Cast from a VARCHAR to distinct type *DT* with a source data type CHAR
- Cast from a VARGRAPHIC to distinct type *DT* with a source data type GRAPHIC
- For a Unicode database, cast from a VARCHAR or a VARGRAPHIC to distinct type *DT* with a source data type CHAR or GRAPHIC
- Cast from a distinct type *DT* with a source data type *S* to XML using the XMLCAST specification
- Cast from an XML to a distinct type *DT* with a source data type of any built-in data type, using the XMLCAST specification depending on the XML schema data type of the XML value

For casts involving a weakly typed distinct type as a target, the cast from data type must be castable to the source type of the weakly typed distinct type and the data type constraints must evaluate to true or unknown for the value. The only context in which an operand is implicitly cast to a weakly typed distinct type is when a source operand is assigned to a target with a data type that is a weakly typed distinct type.

FOR BIT DATA character types cannot be cast to CLOB.

For casts that involve an array type as a target, the data type of the elements of the source array value must be castable to the data type of the elements of the target array data (SQLSTATE 42846). If the target array type is an ordinary array, the source array value must be an ordinary array (SQLSTATE 42821) and the cardinality of the source array value must be less than or equal to the maximum cardinality of the target array data type (SQLSTATE 2202F). If the target array type is an associative array, the data type of the index for the source array value must be castable to data type of the index for the target array type. A user-defined array type value can be cast only to the same user-defined array type (SQLSTATE 42846).

A cursor type cannot be either the source data type or the target data type of a CAST specification, except to cast a parameter marker to a cursor type.

For casts that involve a row type as a target, the degree of the source row value expression and degree of the target row type must match and each field in the source row value expression must be castable to the corresponding target field. A user-defined row type value can only be cast to another user-defined row-type with the same name (SQLSTATE 42846).

It is not possible to cast a structured type value to something else. A structured type *ST* should not need to be cast to one of its supertypes, because all methods on the supertypes of *ST* are applicable to *ST*. If the required operation is only applicable to a subtype of *ST*, use the subtype-treatment expression to treat *ST* as one of its subtypes.

When a user-defined data type involved in a cast is not qualified by a schema name, the *SQL path* is used to find the first schema that includes the user-defined data type by that name.

The following casts involving reference types are supported:

- cast from reference type *RT* to its representation data type *S*
- cast from the representation data type *S* of reference type *RT* to reference type *RT*
- cast from reference type *RT* with target type *T* to a reference type *RS* with target type *S* where *S* is a supertype of *T*.
- cast from a data type *A* to reference type *RT*, where *A* is promotable to the representation data type *S* of reference type *RT*.

When the target type of a reference data type involved in a cast is not qualified by a schema name, the *SQL path* is used to find the first schema that includes the user-defined data type by that name.

Table 7. Supported Casts between Built-in Data Types

Source Data Type	**Target Data Type**																						
	SMALLINT	**INTEGER**	**BIGINT**	**DECIMAL**	**REAL**	**DOUBLE**	**DECFLOAT**	**CHAR**	**CHAR FBD 2**	**VARCHAR**	**VARCHAR FBD 2**	**CLOB**	**GRAPHIC**	**VARGRAPHIC**	**DBCLOB**	**BINARY**	**VARBINARY**	**BLOB**	**DATE**	**TIME**	**TIMESTAMP**	**XML**	**BOOLEAN**
SMALLINT	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	Y [1]	Y [1]	-	-	-	-	-	-	-	Y [3]	Y
INTEGER	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	Y [1]	Y [1]	-	-	-	-	-	-	-	Y [3]	Y
BIGINT	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	Y [1]	Y [1]	-	-	-	-	-	-	-	Y [3]	Y
DECIMAL	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	Y [1]	Y [1]	-	-	-	-	-	-	-	Y [3]	-
REAL	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	Y [1]	Y [1]	-	-	-	-	-	-	-	Y [3]	-
DOUBLE	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	Y [1]	Y [1]	-	-	-	-	-	-	-	Y [3]	-
DECFLOAT	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	Y [1]	Y [1]	-	-	-	-	-	-	-	-	-
CHAR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y [1]	Y	Y	Y	Y	Y	Y	Y [4]	Y
CHAR FOR BIT DATA	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	Y	Y	Y	Y	Y	Y	Y [3]	-
VARCHAR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y [1]	Y	Y	Y	Y	Y	Y	Y [4]	Y
VARCHAR FOR BIT DATA	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	Y	Y	Y	Y	Y	Y	Y [3]	-
CLOB	-	-	-	-	-	-	-	Y	-	Y	-	Y	Y [1]	Y [1]	Y [1]	Y	Y	Y	-	-	-	Y [4]	-

Table 7. Supported Casts between Built-in Data Types (continued)

Source Data Type	Target Data Type																						
	SMALLINT	INTEGER	BIGINT	DECIMAL	REAL	DOUBLE	DECFLOAT	CHAR	CHAR FBD[2]	VARCHAR	VARCHAR FBD[2]	CLOB	GRAPHIC	VARGRAPHIC	DBCLOB	BINARY	VARBINARY	BLOB	DATE	TIME	TIMESTAMP	XML	BOOLEAN
GRAPHIC	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	-	Y[1]	-	Y[1]	Y	Y	Y	Y	Y	Y	Y[1]	Y[1]	Y[1]	Y[3]	Y[1]
VARGRAPHIC	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	Y[1]	-	Y[1]	-	Y[1]	Y	Y	Y	Y	Y	Y	Y[1]	Y[1]	Y[1]	Y[3]	Y[1]
DBCLOB	-	-	-	-	-	-	-	Y[1]	-	Y[1]	-	Y[1]	Y	Y	Y	Y	Y	Y	-	-	-	Y[3]	-
BINARY	-	-	-	-	-	-	-	-	Y	-	Y	-	-	-	-	Y	Y	Y	-	-	-	-	-
VARBINARY	-	-	-	-	-	-	-	-	Y	-	Y	-	-	-	-	Y	Y	Y	-	-	-	-	-
BLOB	-	-	-	-	-	-	-	-	Y	-	Y	-	-	-	-	Y	Y	Y	-	-	-	Y[4]	-
DATE	-	Y	Y	Y	-	-	-	Y	Y	Y	Y	-	Y[1]	Y[1]	-	-	-	-	Y	-	Y	Y[3]	-
TIME	-	Y	Y	Y	-	-	-	Y	Y	Y	Y	-	Y[1]	Y[1]	-	-	-	-	-	Y	-	Y[3]	-
TIMESTAMP	-	-	Y	Y	-	-	-	Y	Y	Y	Y	-	Y[1]	Y[1]	-	-	-	-	Y	Y	Y	Y[3]	-
XML	Y[5]	Y[5]	Y[5]	Y[5]	Y[5]	Y[5]	Y[5]	Y[5]	Y[5]	Y[5]	Y[5]	Y[5]	Y[5]	Y[5]	Y[5]	-	-	Y[5]	Y[5]	Y[5]	Y[5]	Y	-
BOOLEAN	Y	Y	Y	-	-	-	-	Y	-	Y	-	-	Y[1]	Y[1]	-	-	-	-	-	-	-	-	Y

Table 7. Supported Casts between Built-in Data Types (continued)

Source Data Type	Target Data Type																						
	SMALLINT	INTEGER	BIGINT	DECIMAL	REAL	DOUBLE	DECFLOAT	CHAR	CHAR FBD[2]	VARCHAR	VARCHAR FBD[2]	CLOB	GRAPHIC	VARGRAPHIC	DBCLOB	BINARY	VARBINARY	BLOB	DATE	TIME	TIMESTAMP	XML	BOOLEAN

Notes

- See the description preceding the table for information about supported casts involving user-defined types and reference types.
- It is not possible to cast a structured type value to anything else.
- The LONG VARCHAR and LONG VARGRAPHIC data types continue to be supported but are deprecated, not recommended, and might be removed in a future release.

[1] Cast is only supported for Unicode databases.

[2] FOR BIT DATA

[3] Cast can only be performed using XMLCAST.

[4] An XMLPARSE function is implicitly processed to convert a string to XML on assignment (INSERT or UPDATE) of a string to an XML column. The string must be a well-formed XML document for the assignment to succeed.

[5] Cast can only be performed using XMLCAST and depends on the underlying XML schema data type of the XML value. For details, see "XMLCAST".

[6] A cursor type cannot be either the source data type or the target data type of a CAST specification, except to cast a parameter marker to a cursor type.

Table 8 on page 51 shows where to find information about the rules that apply when casting to the identified target data types.

Table 8. Rules for Casting to a Data Type

Target Data Type	Rules
SMALLINT	“SMALLINT ” on page 502.
INTEGER	“INTEGER ” on page 379.
BIGINT	“BIGINT ” on page 290.
DECIMAL	“DECIMAL or DEC ” on page 331
NUMERIC	“DECIMAL or DEC ” on page 331
REAL	“REAL ” on page 455
DOUBLE	“DOUBLE_PRECISION or DOUBLE” on page 339
DECFLOAT	“DECFLOAT ” on page 327
CHAR	“CHAR ” on page 300

Table 8. Rules for Casting to a Data Type (continued)	
Target Data Type	**Rules**
VARCHAR	“VARCHAR ” on page 557
CLOB	“CLOB ” on page 308
GRAPHIC	“GRAPHIC ” on page 353
VARGRAPHIC	“VARGRAPHIC ” on page 573
DBCLOB	“DBCLOB ” on page 325
BINARY	“BINARY ” on page 292
VARBINARY	“VARBINARY ” on page 556
BLOB	“BLOB ” on page 295
DATE	“DATE ” on page 315
TIME	“TIME ” on page 525
TIMESTAMP	If the source type is a character string, see “TIMESTAMP ” on page 525, where one operand is specified. If the source data type is a DATE, the timestamp is composed of the specified date and a time of 00:00:00.
BOOLEAN	“BOOLEAN ” on page 295

Casting non-XML values to XML values

Table 9. Supported Casts from Non-XML Values to XML Values

Source Data Type	**Target Data Type**	
	XML	**Resulting XML Schema Type**
SMALLINT	Y	xs:short
INTEGER	Y	xs:int
BIGINT	Y	xs:long
DECIMAL or NUMERIC	Y	xs:decimal
REAL	Y	xs:float
DOUBLE	Y	xs:double
DECFLOAT	N	-
CHAR	Y	xs:string
VARCHAR	Y	xs:string
CLOB	Y	xs:string
GRAPHIC	Y	xs:string
VARGRAPHIC	Y	xs:string
DBCLOB	Y	xs:string
DATE	Y	xs:date
TIME	Y	xs:time

Table 9. Supported Casts from Non-XML Values to XML Values (continued)

Source Data Type	Target Data Type	
	XML	Resulting XML Schema Type
TIMESTAMP	Y	xs:dateTime[1]
BLOB	Y	xs:base64Binary
BOOLEAN	Y	xs:boolean
character type FOR BIT DATA	Y	xs:base64Binary
distinct type		use this chart with the source type of the distinct type

Notes

[1] The source data type TIMESTAMP supports timestamp precision of 0 to 12. The maximum fractional seconds precision of xs:dateTime is 6. If the timestamp precision of a TIMESTAMP source data type exceeds 6, the value is truncated when cast to xs:dateTime.

The LONG VARCHAR and LONG VARGRAPHIC data types continue to be supported but are deprecated, not recommended, and might be removed in a future release.

When character string values are cast to XML values, the resulting xs:string atomic value cannot contain illegal XML characters (SQLSTATE 0N002). If the input character string is not in Unicode, the input characters are converted to Unicode.

Casting to SQL binary types results in XQuery atomic values with the type xs:base64Binary.

Casting XML values to non-XML values

An XMLCAST from an XML value to a non-XML value can be described as two casts: an XQuery cast that converts the source XML value to an XQuery type corresponding to the SQL target type, followed by a cast from the corresponding XQuery type to the actual SQL type.

An XMLCAST is supported if the target type has a corresponding XQuery target type that is supported, and if there is a supported XQuery cast from the source value's type to the corresponding XQuery target type. The target type that is used in the XQuery cast is based on the corresponding XQuery target type and might contain some additional restrictions.

The following table lists the XQuery types that result from such conversion.

Table 10. Supported Casts from XML Values to Non-XML Values

Target Data Type	Source Data Type	
	XML	Corresponding XQuery Target Type
SMALLINT	Y	xs:short
INTEGER	Y	xs:int
BIGINT	Y	xs:long
DECIMAL or NUMERIC	Y	xs:decimal
REAL	Y	xs:float
DOUBLE	Y	xs:double
DECFLOAT	Y	no matching type[1]
CHAR	Y	xs:string

Table 10. Supported Casts from XML Values to Non-XML Values (continued)

Target Data Type	Source Data Type	
	XML	Corresponding XQuery Target Type
VARCHAR	Y	xs:string
CLOB	Y	xs:string
GRAPHIC	Y	xs:string
VARGRAPHIC	Y	xs:string
DBCLOB	Y	xs:string
DATE	Y	xs:date
TIME (without time zone)	Y	xs:time
TIMESTAMP (without time zone)	Y	xs:dateTime[2]
BLOB	Y	xs:base64Binary
BOOLEAN	Y	xs:boolean
CHAR FOR BIT DATA	N	not castable
VARCHAR FOR BIT DATA	Y	xs:base64Binary
distinct type		use this chart with the source type of the distinct type
row, reference, structured or abstract data type (ADT), other	N	not castable

Notes

[1] XML Schema 1.0 is supported, which does not provide a matching XML schema type for a DECFLOAT. Processing of the XQuery cast step of XMLCAST is handled as follows:

- If the source value is typed with an XML schema numeric type, use that numeric type.
- If the source value is typed with the XML schema type xs:boolean, use xs:double.
- Otherwise, use xs:string with additional checking for a valid numeric format.

[2] The maximum fractional seconds precision of xs:dateTime is 6. The source data type TIMESTAMP supports timestamp precision of 0 to 12. If the timestamp precision of a TIMESTAMP target data type is less than 6, the value is truncated when cast from xs:dateTime. If the timestamp precision of a TIMESTAMP target data type exceeds 6, the value is padded with zeros when cast from xs:dateTime.

In the following restriction cases, a derived by restriction XML schema data type is effectively used as the target data type for the XQuery cast.

- XML values that are to be converted to string types other than CHAR and VARCHAR must fit within the length limits of those data types without truncation of any characters or bytes. The name used for the derived XML schema type is the uppercase SQL type name followed by an underscore character and the maximum length of the string; for example, CLOB_1048576 if the XMLCAST target data type is CLOB(1M) .

  If an XML value is converted to a CHAR or VARCHAR type that is too small to contain all of the data, the data is truncated to fit the specified data type and no error is returned. If any non-blank characters are truncated, a warning (SQLSTATE 01004) is returned. If truncation of the value leads to the truncation of a multibyte character, the whole multibyte character is removed. Therefore, in some cases, truncation can produce a shorter string than expected. For example, the character, ñ, is represented in UTF-8 by 2 bytes, 'C3 B1'. When this character is cast as VARCHAR(1), the truncation of 'C3 B1' to 1 byte would

leave a partial character of 'C3'. This partial character, 'C3', is also removed, therefore the final result is an empty string.

- XML values that are to be converted to DECIMAL values must have no more than (*precision* - *scale*) digits before the decimal point; excess digits after the decimal point beyond the scale are truncated. The name used for the derived XML schema type is DECIMAL_*precision*_*scale*, where *precision* is the precision of the target SQL data type, and *scale* is the scale of the target SQL data type; for example, DECIMAL_9_2 if the XMLCAST target data type is DECIMAL(9,2).
- XML values that are to be converted to TIME values cannot contain a seconds component with nonzero digits after the decimal point. The name used for the derived XML schema type is TIME.

The derived XML schema type name only appears in a message if an XML value does not conform to one of these restrictions. This type name helps one to understand the error message, and does not correspond to any defined XQuery type. If the input value does not conform to the base type of the derived XML schema type (the corresponding XQuery target type), the error message might indicate that type instead. Because this derived XML schema type name format might change in the future, it should not be used as a programming interface.

Before an XML value is processed by the XQuery cast, any document node in the sequence is removed and each direct child of the removed document node becomes an item in the sequence. If the document node has multiple direct children nodes, the revised sequence will have more items than the original sequence. The XML value without any document nodes is then atomized using the XQuery fn:data function, with the resulting atomized sequence value used in the XQuery cast. If the atomized sequence value is an empty sequence, a null value is returned from the cast without any further processing. If there are multiple items in the atomized sequence value, an error is returned (SQLSTATE 10507).

If the target type of XMLCAST is the SQL data type DATE, TIME, or TIMESTAMP, the resulting XML value from the XQuery cast is also adjusted to UTC, and the time zone component of the value is removed.

When the corresponding XQuery target type value is converted to the SQL target type, binary XML data types, such as xs:base64Binary or xs:hexBinary, are converted from character form to actual binary data.

If an xs:double or xs:float value of INF, -INF, or NaN is cast (using XMLCAST) to an SQL data type DOUBLE or REAL value, an error is returned (SQLSTATE 22003). An xs:double or xs:float value of -0 is converted to +0.

The target type can be a user-defined distinct type if the source operand is not a user-defined distinct type. In this case, the source value is cast to the source type of the user-defined distinct type (that is, the target type) using the XMLCAST specification, and then this value is cast to the user-defined distinct type using the CAST specification.

In a non-Unicode database, casting from an XML value to a non-XML target type involves code page conversion from an internal UTF-8 format to the database code page. This conversion will result in the introduction of substitution characters if any code point in the XML value is not present in the database code page.

Assignments and comparisons

The basic operations of SQL are assignment and comparison.

Assignment operations are performed during the execution of INSERT, UPDATE, FETCH, SELECT INTO, VALUES INTO and SET transition-variable statements. Arguments of functions are also assigned when invoking a function. Comparison operations are performed during the execution of statements that include predicates and other language elements such as MAX, MIN, DISTINCT, GROUP BY, and ORDER BY.

One basic rule for both operations is that the data type of the operands involved must be compatible. The compatibility rule also applies to set operations.

Another basic rule for assignment operations is that a null value cannot be assigned to a column that cannot contain null values, nor to a host variable that does not have an associated indicator variable.

Following is a compatibility matrix showing the built-in data type compatibilities for assignment and comparison operations.

Table 11. Data type compatibility for assignments and comparisons

Operands	Binary Integer	Decimal Number	Floating-point	Decimal Floating-point	Character String	Graphic String	Binary String	Date	Time	Time-stamp	Boolean	UDT
Binary Integer	Yes	Yes	Yes	Yes	Yes	Yes [5]	No	No	No	No	Yes	2
Decimal Number	Yes	Yes	Yes	Yes	Yes	Yes [5]	No	No	No	No	No	2
Floating point	Yes	Yes	Yes	Yes	Yes	Yes [5]	No	No	No	No	No	2
Decimal Floating-point	Yes	Yes	Yes	Yes	Yes	Yes [5]	No	No	No	No	No	2
Character String	Yes	Yes	Yes	Yes	Yes	Yes [5,6]	Yes [3]	Yes	Yes	Yes	Yes	2
Graphic String	Yes [5]	Yes [5]	Yes [5]	Yes [5]	Yes [5,6]	Yes	No	Yes [5]	Yes [5]	Yes [5]	Yes	2
Binary String	No	No	No	No	Yes[3]	No	Yes	No	No	No	No	2
Date	No	No	No	No	Yes	Yes [5]	No	Yes	No	Yes	No	2
Time	No	No	No	No	Yes	Yes [5]	No	No	Yes	[1]	No	2
Time-stamp	No	No	No	No	Yes	Yes [5]	No	Yes	[1]	Yes	No	2
Boolean	Yes	No	No	No	Yes	Yes	No	No	No	No	Yes	2
UDT	2	2	2	2	2	2	2	2	2	2	2	Yes

[1] A TIMESTAMP value can be assigned to a TIME value; however, a TIME value cannot be assigned to a TIMESTAMP value and a TIMESTAMP value cannot be compared with a TIME value.

[2] For detailed user-defined type information see "User-defined type assignments" on page 62 and "User-defined type comparisons" on page 69.

[3] Character strings, except those with FOR BIT DATA, are not compatible with binary strings. FOR BIT DATA character strings and binary strings are considered compatible and any padding is performed based on the data type of the target. For example, when assigning a FOR BIT DATA column value to a fixed-length binary host variable, any necessary padding uses a pad byte of X'00'.

[4] For information about assignment and comparison of reference types, see "Reference type assignments" on page 65 and "Reference type comparisons" on page 71.

[5] Only supported for Unicode databases.

[6] Bit data and graphic strings are not compatible.

Numeric assignments

For numeric assignments, overflow is not allowed.

- When assigning to an exact numeric data type, overflow occurs if any digit of the whole part of the number would be eliminated. If necessary, the fractional part of a number is truncated.
- When assigning to an approximate numeric data type or decimal floating-point, overflow occurs if the most significant digit of the whole part of the number is eliminated. For floating-point and decimal floating-point numbers, the whole part of the number is the number that would result if the floating-point or decimal floating-point number were converted to a decimal number with unlimited precision. If necessary, rounding might cause the least significant digits of the number to be eliminated.

  For decimal floating-point numbers, truncation of the whole part of the number is allowed and results in infinity with a warning.

  For floating-point numbers, underflow is also not allowed. Underflow occurs for numbers between 1 and -1 if the most significant digit other than zero would be eliminated. For decimal floating-point, underflow is allowed and depending on the rounding mode, results in zero or the smallest positive number or the largest negative number that can be represented along with a warning.

  An overflow or underflow warning is returned instead of an error if an overflow or underflow occurs on assignment to a host variable with an indicator variable. In this case, the number is not assigned to the host variable and the indicator variable is set to negative 2.

For decimal floating-point numbers, the CURRENT DECFLOAT ROUNDING MODE special register indicates the rounding mode in effect.

Assignments to integer

When a decimal, floating-point, or decimal floating-point number is assigned to an integer column or variable, the fractional part of the number is eliminated. As a result, a number between 1 and -1 is reduced to 0.

Assignments to decimal

When an integer is assigned to a decimal column or variable, the number is first converted to a temporary decimal number and then, if necessary, to the precision and scale of the target. The precision and scale of the temporary decimal number is 5,0 for a small integer, 11,0 for a large integer, or 19,0 for a big integer.

When a decimal number is assigned to a decimal column or variable, the number is converted, if necessary, to the precision and the scale of the target. The necessary number of leading zeros is added, and in the fractional part of the decimal number the necessary number of trailing zeros is added, or the necessary number of trailing digits is eliminated.

When a floating-point number is assigned to a decimal column or variable, the number is first converted to a temporary decimal number of precision 31, and then, if necessary, truncated to the precision and scale of the target. In this conversion, the number is rounded (using floating-point arithmetic) to a precision of 31 decimal digits. As a result, a number between 1 and -1 that is less than the smallest positive number or greater than the largest negative number that can be represented in the decimal column or variable is reduced to 0. The scale is given the largest possible value that allows the whole part of the number to be represented without loss of significance.

When a decimal floating-point number is assigned to a decimal column or variable, the number is rounded to the precision and scale of the decimal column or variable. As a result, a number between 1 and -1 that is less than the smallest positive number or greater than the largest negative number that can be represented in the decimal column or variable is reduced to 0 or rounded to the smallest positive or largest negative value that can be represented in the decimal column or variable, depending on the rounding mode.

Assignments to floating-point

Floating-point numbers are approximations of real numbers. Hence, when an integer, decimal, floating-point, or decimal floating-point number is assigned to a floating-point column or variable, the result might not be identical to the original number. The number is rounded to the precision of the floating-point column or variable using floating-point arithmetic. A decimal floating-point value is first converted to a string representation, and is then converted to a floating-point number.

Assignments to decimal floating-point

When an integer number is assigned to a decimal floating-point column or variable, the number is first converted to a temporary decimal number and then to a decimal floating-point number. The precision and scale of the temporary decimal number is 5,0 for a small integer, 11,0 for a large integer, or 19,0 for a big integer. Rounding can occur when assigning a BIGINT to a DECFLOAT(16) column or variable.

When a decimal number is assigned to a decimal floating-point column or variable, the number is converted to the precision (16 or 34) of the target. Leading zeros are eliminated. Depending on the precision and scale of the decimal number and the precision of the target, the value might be rounded.

When a floating-point number is assigned to a decimal floating-point column or variable, the number is first converted to a temporary string representation of the floating-point number. The string representation of the number is then converted to decimal floating-point.

When a DECFLOAT(16) number is assigned to a DECFLOAT(34) column or variable, the resulting value is identical to the DECFLOAT(16) number.

When a DECFLOAT(34) number is assigned to a DECFLOAT(16) column or variable, the exponent of the source is converted to the corresponding exponent in the result format. The mantissa of the DECFLOAT(34) number is rounded to the precision of the target.

Assignments from strings to numeric

When a string is assigned to a numeric data type, it is converted to the target numeric data type using the rules for a CAST specification. For more information, see "CAST specification" in the *SQL Reference Volume 1*.

String assignments

There are two types of assignments:

- In *storage assignment*, a value is assigned and truncation of significant data is not desirable; for example, when assigning a value to a column
- In *retrieval assignment*, a value is assigned and truncation is allowed; for example, when retrieving data from the database

The rules for string assignment differ based on the assignment type and the type of string.

Binary string assignments

Storage assignment

The length of a string that is assigned to a binary string target must not be greater than the length attribute of the target. If the length of the string is greater than the length attribute of the target, the following actions might occur:

- The string is assigned with trailing hexadecimal zeros (X'00') truncated (except BLOB strings) to fit the length attribute of the target
- An error is returned (SQLSTATE 22001) in the following truncation scenarios:
  - Characters other than a hexadecimal zero (X'00) would be truncated from a binary string other than a BLOB string.
  - Any character (or byte) would be truncated from a BLOB string

When the string is assigned to a fixed-length binary target, and the length of the string is less than the length attribute of the target, the string is padded to the right with the necessary number of hexadecimal zeros.

Retrieval Assignment

The length of a string that is assigned to a target can be longer than the length attribute of the target. When a string is assigned to a target, and the length of the string is longer than the length attribute of the target, the string is truncated on the right by the necessary number of bytes. When this truncation occurs, a warning is returned (SQLSTATE 01004).

Furthermore, if the data is truncated when it is being assigned to a host variable, the value 'W' is assigned to the SQLWARN1 field of the SQLCA. Also, if an indicator variable is provided, and the source of the value is not a LOB, the indicator variable is set to the original length of the string.

If a string is assigned to a fixed-length target, and the length of the string is less than the length attribute of the target, the string is padded to the right with the necessary number of hexadecimal zeroes.

Character and graphic string assignments

Storage assignment

The basic rule is that the length of the string assigned to the target must not be greater than the length attribute of the target. If the length of the string is greater than the length attribute of the target, the following actions might occur:

- The string is assigned with trailing blanks truncated (from all string types except LOB strings) to fit the length attribute of the target
- An error is returned (SQLSTATE 22001) in the following truncation scenarios:
  - Non-blank characters would be truncated from strings other than a LOB string
  - Any character (or byte) would be truncated from a LOB string

In a Unicode database, length is defined as the number of code units in the string units of the target.

- If the target string units are OCTETS, the string assigned to the target must not have a greater byte length than the target.
- If the target string units are CODEUNITS16, the string assigned to the target must not have a greater number of Unicode UTF-16 code units than the length attribute of the target.
- If the target string units are CODEUNITS32, the string assigned to the target must not have a greater number of Unicode UTF-32 code units than the length attribute of the target.

If a string is assigned to a fixed-length target, and the length of the string is less than the length attribute of the target, the string is padded to the right with the necessary number of single-byte, double-byte, or UCS-2 blanks. The pad character is always a blank, even for columns defined with the FOR BIT DATA attribute. (UCS-2 defines several SPACE characters with different properties. For a Unicode database, the database manager always uses the ASCII SPACE at position x'0020' as UCS-2 blank. For an EUC database, the IDEOGRAPHIC SPACE at position x'3000' is used for padding GRAPHIC strings.)

Retrieval Assignment

The length of a string that is assigned to a target can be longer than the length attribute of the target. When a string is assigned to a target, and the length of the string is longer than the length attribute of the target, the string is truncated on the right by the necessary number of characters (or bytes). When this truncation occurs, a warning is returned (SQLSTATE 01004).

Furthermore, if the data is truncated when it is being assigned to a host variable, the value 'W' is assigned to the SQLWARN1 field of the SQLCA. Also, if an indicator variable is provided, and the source of the value is not a LOB, the indicator variable is set to the original length of the string.

Length is defined as the number of code units in the string units of the target.

- If the target string units are OCTETS and the source string has a greater byte length than the target, the string is truncated on the right by the necessary number of bytes.
- If the target string units are CODEUNITS16 and the source string has a greater number of Unicode UTF-16 code units than the length attribute of the target, the string is truncated on the right by the necessary number of Unicode UTF-16 code units.
- If the target string units are CODEUNITS32 and the source string has a greater number of Unicode UTF-32 code units than the length attribute of the target, the string is truncated on the right by the necessary number of Unicode UTF-32 code units.

If a character string is assigned to a fixed-length target, and the length of the string is less than the length attribute of the target, the string is padded to the right with the necessary number of single-byte, double-byte, or UCS-2 blanks. The pad character is always a blank, even for strings defined with the FOR BIT DATA attribute. (UCS-2 defines several SPACE characters with different properties. For a Unicode database, the database manager always uses the ASCII SPACE at position x'0020' as UCS-2 blank. For an EUC database, the IDEOGRAPHIC SPACE at position x'3000' is used for padding GRAPHIC strings.)

Retrieval assignment of C NUL-terminated host variables is handled based on options that are specified with the PREP or BIND command.

Conversion rules for string assignments

A character string or graphic string assigned to a column, variable, or parameter is first converted, if necessary, to the code page of the target. Character conversion is necessary only if all of the following are true:

- The code pages are different.
- The string is neither null nor empty.
- Neither string has a code page value of 0 (FOR BIT DATA).

For Unicode databases, character strings can be assigned to a graphic column, and graphic strings can be assigned to a character column.

MBCS considerations for character string assignments

There are several considerations when assigning character strings that could contain both single and multi-byte characters. These considerations apply to all character strings, including those defined as FOR BIT DATA.

- Blank padding is always done using the single-byte blank character (X'20').
- Blank truncation is always done based on the single-byte blank character (X'20'). The double-byte blank character is treated like any other character with respect to truncation.
- Assignment of a character string to a host variable might result in fragmentation of MBCS characters if the target host variable is not large enough to contain the entire source string. If an MBCS character is fragmented, each byte of the MBCS character fragment in the target is set to a single-byte blank character (X'20'), no further bytes are moved from the source, and SQLWARN1 is set to 'W' to indicate truncation. Note that the same MBCS character fragment handling applies even when the character string is defined as FOR BIT DATA.

DBCS considerations for graphic string assignments

Graphic string assignments are processed in a manner analogous to that for character strings. For non-Unicode databases, graphic string data types are compatible only with other graphic string data types, and never with numeric, character string, or datetime data types. For Unicode databases, graphic string data types are compatible with character string data types. However, graphic and character string data types cannot be used interchangeably in the SELECT INTO or the VALUES INTO statement.

If a graphic string value is assigned to a graphic string column, the length of the value must not be greater than the length of the column.

If a graphic string value (the 'source' string) is assigned to a fixed-length graphic string data type (the 'target', which can be a column, variable, or parameter), and the length of the source string is less than that of the target, the target will contain a copy of the source string which has been padded on the right with the necessary number of double-byte blank characters to create a value whose length equals that of the target.

If a graphic string value is assigned to a graphic string host variable and the length of the source string is greater than the length of the host variable, the host variable will contain a copy of the source string which has been truncated on the right by the necessary number of double-byte characters to create a value whose length equals that of the host variable. (Note that for this scenario, truncation need not be concerned with bisection of a double-byte character; if bisection were to occur, either the source value or target host variable would be an ill-defined graphic string data type.) The warning flag SQLWARN1 in the SQLCA will be set to 'W'. The indicator variable, if specified, will contain the original length (in double-byte characters) of the source string. In the case of DBCLOB, however, the indicator variable does not contain the original length.

Retrieval assignment of C NUL-terminated host variables (declared using wchar_t) is handled based on options specified with the PREP or BIND command.

Assignments from numeric to strings

When a number is assigned to a string data type, it is converted to the target string data type using the rules for a CAST specification. For more information, see "CAST specification" in the *SQL Reference Volume 1*.

If a nonblank character is truncated during the cast of a numeric value to a character or graphic data type, a warning is returned. This truncation behavior is unlike the assignment to a character or graphic data type that follows storage assignment rules, where if a nonblank character is truncated during assignment, an error is returned.

Datetime assignments

A TIME value can be assigned only to a TIME column or to a string variable or string column.

A DATE can be assigned to a DATE, TIMESTAMP or string data type. When a DATE value is assigned to a TIMESTAMP data type, the missing time information is assumed to be all zeros.

A TIMESTAMP value can be assigned to a DATE, TIME, TIMESTAMP or string data type. When a TIMESTAMP value is assigned to a DATE data type, the date portion is extracted and the time portion is truncated. When a TIMESTAMP value is assigned to a TIME data type, the date portion is ignored and the time portion is extracted, but with the fractional seconds truncated. When a TIMESTAMP value is assigned to a TIMESTAMP with lower precision, the excess fractional seconds are truncated. When a TIMESTAMP value is assigned to a TIMESTAMP with higher precision, missing digits are assumed to be zeros.

The assignment must not be to a CLOB, DBCLOB, or BLOB variable or column.

When a datetime value is assigned to a string variable or string column, conversion to a string representation is automatic. Leading zeros are not omitted from any part of the date, time, or timestamp. The required length of the target will vary, depending on the format of the string representation. If the length of the target is greater than required, and the target is a fixed-length string, it is padded on the right with blanks. If the length of the target is less than required, the result depends on the type of datetime value involved, and on the type of target.

When the target is not a host variable and has a character data type, truncation is not allowed. The length attribute of the column must be at least 10 for a date, 8 for a time, 19 for a TIMESTAMP(0), and 20+*p* for TIMESTAMP(*p*).

When the target is a string host variable, the following rules apply:

- **For a DATE:** If the length of the host variable is less than 10 characters, an error is returned.
- **For a TIME:** If the USA format is used, the length of the host variable must not be less than 8 characters; in other formats the length must not be less than 5 characters.

  If ISO or JIS formats are used, and if the length of the host variable is less than 8 characters, the seconds part of the time is omitted from the result and assigned to the indicator variable, if provided. The SQLWARN1 field of the SQLCA is set to indicate the omission.
- **For a TIMESTAMP:** If the length of the host variable is less than 19 characters, an error is returned. If the length is less than 32 characters, but greater than or equal to 19 characters, trailing digits of the fractional seconds part of the value are omitted. The SQLWARN1 field of the SQLCA is set to indicate the omission.

When a DATE is assigned to a TIMESTAMP, the time and fractional components of the timestamp are set to midnight and 0, respectively. When a TIMESTAMP is assigned to a DATE, the date portion is extracted and the time and fractional components are truncated.

When a TIMESTAMP is assigned to a TIME, the DATE portion is ignored and the fractional components are truncated.

XML assignments

The general rule for XML assignments is that only an XML value can be assigned to XML columns or to XML variables. There are exceptions to this rule, as follows.

- ***Processing of input XML host variables:*** This is a special case of the XML assignment rule, because the host variable is based on a string value. To make the assignment to XML within SQL, the string value is implicitly parsed into an XML value using the setting of the CURRENT IMPLICIT XMLPARSE OPTION

special register. This determines whether to preserve or to strip whitespace, unless the host variable is an argument of the XMLVALIDATE function, which always strips unnecessary whitespace.

- ***Assigning strings to input parameter markers of data type XML:*** If an input parameter marker has an implicit or explicit data type of XML, the value bound (assigned) to that parameter marker could be a character string variable, graphic string variable, or binary string variable. In this case, the string value is implicitly parsed into an XML value using the setting of the CURRENT IMPLICIT XMLPARSE OPTION special register to determine whether to preserve or to strip whitespace, unless the parameter marker is an argument of the XMLVALIDATE function, which always strips unnecessary whitespace.
- ***Assigning strings directly to XML columns in data change statements:*** If assigning directly to a column of type XML in a data change statement, the assigned expression can also be a character string or a binary string. In this case, the result of `XMLPARSE (DOCUMENT` *`expression`* `STRIP WHITESPACE)` is assigned to the target column. The supported string data types are defined by the supported arguments for the XMLPARSE function. Note that this XML assignment exception does not allow character or binary string values to be assigned to SQL variables or to SQL parameters of data type XML.
- ***Assigning XML to strings on retrieval:*** If retrieving XML values into host variables using a FETCH INTO statement or an EXECUTE INTO statement in embedded SQL, the data type of the host variable can be CLOB, DBCLOB, or BLOB. If using other application programming interfaces (such as CLI, JDBC, or .NET), XML values can be retrieved into the character, graphic, or binary string types that are supported by the application programming interface. In all of these cases, the XML value is implicitly serialized to a string encoded in UTF-8 and, for character or graphic string variables, converted into the client code page.

Character string or binary string values cannot be retrieved into XML host variables. Values in XML host variables cannot be assigned to columns, SQL variables, or SQL parameters of a character string data type or a binary string data type.

Assignment to XML parameters and variables in inlined SQL bodied UDFs and SQL procedures is done by reference. Passing parameters of data type XML to invoke an inlined SQL UDF or SQL procedure is also done by reference. When XML values are passed by reference, any input node trees are used directly. This direct usage preserves all properties, including document order, the original node identities, and all parent properties.

User-defined type assignments

Assignments involving user-defined type values generally allow assignment to the same user-defined type name with some additional rules for the different kinds of user-defined types. Additional information about specific user-defined types is in the sections that follow.

Strongly typed distinct type assignments

The rules that apply to the assignments of strongly typed distinct type values to host variables are different than the rules for all other assignments that involve strongly typed distinct type values

Assignments to host variables

The assignment of a strongly typed distinct type value to a host variable (or parameter marker) is based on the source data type of the distinct type. Therefore, the value of a strongly typed distinct type is assignable to a host variable (or parameter marker) only if the source data type of the distinct type is assignable to the variable

For example, the distinct type AGE is created by the following SQL statement:

```
CREATE TYPE AGE AS SMALLINT
```

When the statement is executed, the following cast functions are also generated:

```
AGE (SMALLINT) RETURNS AGE
AGE (INTEGER) RETURNS AGE
SMALLINT (AGE) RETURNS SMALLINT
```

Next, assume that column STU_AGE was defined in table STUDENTS with distinct type AGE. Now, consider this valid assignment of a student's age to host variable HV_AGE, which has an INTEGER data type:

```
SELECT STU_AGE INTO :HV_AGE FROM STUDENTS WHERE STU_NUMBER = 200
```

The strongly typed distinct type value is assignable to host variable HV_AGE because the source data type of the distinct type (SMALLINT) is assignable to the host variable (INTEGER). If strongly typed distinct type AGE had been sourced on a datetime data type such as DATE, the preceding assignment would be invalid because a datetime data type cannot be assigned to an integer type.

Assignments other than to host variables

A strongly typed distinct type can be either the source or target of an assignment. Assignment is based on whether the data type of the value to be assigned is castable to the data type of the target (see "Casting between data types" on page 47 for the casts supported when a distinct type is involved). A strongly typed distinct type value can be assigned to any target other than host a variable in the following cases:

- The target of the assignment has the same distinct type.
- The distinct type is castable to the data type of the target.

Any value can be assigned to a strongly typed distinct type when:

- The value to be assigned has the same distinct type as the target.
- The data type of the assigned value is castable to the target distinct type.

For example, the source data type for strongly typed distinct type AGE is SMALLINT:

```
CREATE TYPE AGE AS SMALLINT
```

Next, assume that the tables TABLE1 and TABLE2 are created with four identical column descriptions:

```
AGECOL AGE
SMINTCOL SMALLINT
INTCOL INTEGER
DECCOL DECIMAL(6,2)
```

Using the following SQL statement and substituting various values for X and Y to insert values into various columns of TABLE1 from TABLE2. The database manager uses assignment rules in this INSERT statement to determine if X can be assigned to Y. Table 12 on page 63 shows whether the assignments are valid.

```
INSERT INTO TABLE1(Y)
   SELECT X FROM TABLE2;
```

Table 12. Assessment of various assignments

TABLE2.X	TABLE1.Y	Valid	Reason
AGECOL	AGECOL	Yes	Source and target are the same distinct type
SMINTCOL	AGECOL	Yes	SMALLINT can be cast to AGE
INTCOL	AGECOL	Yes	INTEGER can be cast to AGE because the AGE source type is SMALLINT
DECCOL	AGECOL	No	DECIMAL cannot be cast to AGE
AGECOL	SMINTCOL	Yes	AGECOL can be cast to its source type of SMALLINT
AGECOL	INTCOL	No	AGE cannot be cast to INTEGER
AGECOL	DECCOL	No	AGE cannot be cast to DECIMAL

Weakly typed distinct type assignments

The value of a weakly typed distinct type can be assigned according to the assignment rules for the source type of the weakly typed distinct type. When the assignment target has a distinct type defined with data type constraints, the data type constraints are applied to the source value and must evaluate to true or unknown.

Structured type assignments

The value of a structured type can be assigned when the target of the assignment has the same structured type or one of its supertypes except in cases that involve host variables.

Assignment to and from host variables is based on the declared type of the host variable; that is, it follows the rule:

- A source value of a structured type is assignable to a target host variable if and only if the declared type of the host variable is the structured type or a supertype of the structured type.

If the target of the assignment is a column of a structured type, the source data type must be the target data type or a subtype of the target data type.

Array type assignments

The value for an element of an array must be assignable to the data type of the array elements. The assignment rules for that data type apply to the value assignment. The value specified for an index in the array must be assignable to the data type of the index for the array. The assignment rules for that data type apply to the value assignment. For an ordinary array, the index data type is INTEGER and for an associative array the data type is either INTEGER or VARCHAR(n), where n is any valid length attribute for the VARCHAR data type. If the index value for an assignment to an ordinary array is larger than the current cardinality of the array, then the cardinality of the array is increased to the new index value, provided the value does not exceed the maximum value for an INTEGER data type. An assignment of one new element to an associative array increases the cardinality by exactly 1 since the index values can be sparse.

The validity of an assignment to an SQL variable or parameter is determined according to the following rules:

- If the right side of the assignment is an SQL variable or parameter, an invocation of the TRIM_ARRAY function, an invocation of the ARRAY_DELETE function, or a CAST expression, then its type must be the same as the type of the SQL variable or parameter on the left side of the assignment.
- If the right side of the assignment is an array constructor or an invocation of the ARRAY_AGG function, then it is implicitly cast to the type of the SQL variable or parameter on the left side.

For example, assuming that the type of variable *V* is MYARRAY, the statement:

```
   SET V = ARRAY[1,2,3];
```

is equivalent to:

```
SET V = CAST(ARRAY[1,2,3] AS MYARRAY);
```

And the statement:

```
SELECT ARRAY_AGG(C1) INTO V FROM T
```

is equivalent to:

```
SELECT CAST(ARRAY_AGG(C1) AS MYARRAY) INTO V FROM T
```

The following are valid assignments that involve array type values:

- Array variable to another array variable with the same array type as the source variable.
- An expression of type array to an array variable, where the array element type in the source expression is assignable to the array element type in the target array variable.

Row type assignments

Assignments to fields within a row variable must conform to the same rules as if the field itself was a variable of the same data type as the field. A row variable can be assigned only to a row variable with the same user-defined row type. When using FETCH, SELECT, or VALUES INTO to assign values to a row variable, the source value types must be assignable to the target row fields. If the source or the target variable (or both) of an assignment is anchored to the row of a table or view, the number of fields must be the same and the field types of the source value must be assignable to the field types of the target value.

Cursor type assignments

Assignments to cursors depend on the type of cursor. The following values are assignable to a variable or parameter of built-in type CURSOR:

- A cursor value constructor
- A value of built-in type CURSOR
- A value of any user-defined cursor type

The following values are assignable to a variable or parameter of a weakly typed user-defined cursor type:

- A cursor value constructor
- A value of built-in type CURSOR
- A value of a user-defined cursor type with the same type name

The following values are assignable to a variable or parameter of strongly typed user-defined cursor type:

- A cursor value constructor
- A value of a user-defined cursor type with the same type name

Boolean type assignments

The following keywords represent values that are assignable to a variable, parameter, or return type of built-in type BOOLEAN:

- TRUE
- FALSE
- NULL

The result of the evaluation of a search condition can also be assigned. If the search condition evaluates to unknown, the value of NULL is assigned.

Reference type assignments

A reference type with a target type of *T* can be assigned to a reference type column that is also a reference type with target type of *S* where *S* is a supertype of *T*. If an assignment is made to a scoped reference column or variable, no check is performed to ensure that the actual value being assigned exists in the target table or view defined by the scope.

Assignment to host variables is done based on the representation type of the reference type. That is, it follows the rule:

- A value of a reference type on the right hand side of an assignment is assignable to a host variable on the left side if and only if the representation type of this reference type is assignable to this host variable.

If the target of the assignment is a column, and the right side of the assignment is a host variable, the host variable must be explicitly cast to the reference type of the target column.

Numeric comparisons

Numbers are compared algebraically; that is, with regard to sign. For example, -2 is less than +1.

If one number is an integer and the other is decimal, the comparison is made with a temporary copy of the integer, which has been converted to decimal.

When decimal numbers with different scales are compared, the comparison is made with a temporary copy of one of the numbers that has been extended with trailing zeros so that its fractional part has the same number of digits as the other number.

If one number is floating-point and the other is integer or decimal, the comparison is made with a temporary copy of the other number, which has been converted to double-precision floating-point.

Two floating-point numbers are equal only if the bit configurations of their normalized forms are identical.

If one number is decimal floating-point and the other number is integer, decimal, single precision floating-point, or double precision floating-point, the comparison is made with a temporary copy of the other number, which has been converted to decimal floating-point.

If one number is DECFLOAT(16) and the other number is DECFLOAT(34), the DECFLOAT(16) value is converted to DECFLOAT(34) before the comparison is made.

The decimal floating-point data type supports both positive and negative zero. Positive and negative zero have different binary representations, but the = (equal) predicate will return true for comparisons of negative and positive zero.

The COMPARE_DECFLOAT and TOTALORDER scalar functions can be used to perform comparisons at a binary level if, for example, a comparison of 2.0 <> 2.00 is required.

The decimal floating-point data type supports the specification of negative and positive NaN (quiet and signalling), and negative and positive infinity. From an SQL perspective, INFINITY = INFINITY, NAN = NAN, SNAN = SNAN, and -0 = 0.

The comparison and ordering rules for special values are as follows:

- (+/-) INFINITY compares equal only to (+/-) INFINITY of the same sign.
- (+/-) NAN compares equal only to (+/-) NAN of the same sign.
- (+/-) SNAN compares equal only to (+/-) SNAN of the same sign.

The ordering among different special values is as follows:

- -NAN < -SNAN < -INFINITY < 0 < INFINITY < SNAN < NAN

When string and numeric data types are compared, the string is cast to DECFLOAT(34) using the rules for a CAST specification. For more information, see "CAST specification" in the *SQL Reference Volume 1*. The string must contain a valid string representation of a number.

String comparisons

Binary string comparisons

Binary string comparisons are always performed by comparing the binary values for the corresponding bytes of each string. Additionally, two binary strings are equal only if the actual length of the two strings is identical. The shorter string is considered less than the longer string when otherwise equal to the length of the shorter string. Binary strings cannot be compared with character strings unless the character string is cast to a binary string or the character string has a subtype of FOR BIT DATA. In this case, the FOR BIT DATA character string is treated as if it were a binary string.

BLOB strings that have an actual length less than 32673 bytes are supported as operands in basic predicates, IN, BETWEEN and the simple CASE expression. In comparisons that use the LIKE predicate, NULL predicate, and the POSSTR function, BLOB strings of any length continue to be supported.

Character and graphic string comparisons

Character strings are compared according to the collating sequence specified when the database was created with the following exceptions:

- Character strings with a FOR BIT DATA attribute are always compared according to their bit values.

- In a non-Unicode database, string comparisons that involve a Unicode string data type use the alternate collating sequence.

When comparing character strings of unequal lengths, the comparison is made using a logical copy of the shorter string, which is padded on the right with blanks sufficient to extend its length to that of the longer string. This logical extension is done for all character strings, including those tagged as FOR BIT DATA.

Character strings (except character strings tagged as FOR BIT DATA) are compared either according to the collating sequence specified when the database was created, or according to the alternate collating sequence. For example, the default collating sequence supplied by the database manager may give lowercase and uppercase versions of the same character the same weight. The database manager performs a two-pass comparison to ensure that only identical strings are considered equal to each other. In the first pass, strings are compared according to the database collating sequence. If the weights of the characters in the strings are equal, a second "tie-breaker" pass is performed to compare the strings on the basis of their actual code point values.

Two strings are equal if they are both empty or if all corresponding bytes are equal. If either operand is null, the result is unknown.

LOB strings that have an actual length less than 32673 bytes are now supported as operands in basic predicates, IN, BETWEEN and the simple CASE expression. In comparisons using the LIKE predicate, NULL predicate, and the POSSTR function, LOB strings of any length continue to be supported.

LOB strings are not supported in any other comparison operations such as MAX, MIN, DISTINCT, GROUP BY, and ORDER BY.

Portions of strings can be compared using the SUBSTR and VARCHAR scalar functions. For example, given the columns:

```
MY_SHORT_CLOB   CLOB(300)
MY_LONG_VAR     VARCHAR(8000)
```

then the following is valid:

```
WHERE VARCHAR(MY_SHORT_CLOB) > VARCHAR(SUBSTR(MY_LONG_VAR,1,300))
```

Examples:

For these examples, 'A', 'Á', 'a', and 'á', have the code point values X'41', X'C1', X'61', and X'E1' respectively.

Consider a collating sequence where the characters 'A', 'Á', 'a', 'á' have weights 136, 139, 135, and 138. Then the characters sort in the order of their weights as follows:

```
'a' < 'A' < 'á' < 'Á'
```

Now consider four DBCS characters D1, D2, D3, and D4 with code points 0xC141, 0xC161, 0xE141, and 0xE161, respectively. If these DBCS characters are in CHAR columns, they sort as a sequence of bytes according to the collation weights of those bytes. First bytes have weights of 138 and 139, therefore D3 and D4 come before D2 and D1; second bytes have weights of 135 and 136. Hence, the order is as follows:

```
D4 < D3 < D2 < D1
```

However, if the values being compared have the FOR BIT DATA attribute, or if these DBCS characters were stored in a GRAPHIC column, the collation weights are ignored, and characters are compared according to their code points as follows:

```
 'A' < 'a' < 'Á' < 'á'
```

The DBCS characters sort as sequence of bytes, in the order of code points as follows:

```
D1 < D2 < D3 < D4
```

Now consider a collating sequence where the characters 'A', 'Á', 'a', 'á' have (non-unique) weights 74, 75, 74, and 75. Considering collation weights alone (first pass), 'a' is equal to 'A', and 'á' is equal to 'Á'. The code points of the characters are used to break the tie (second pass) as follows:

```
'A' < 'a' < 'Á' < 'á'
```

DBCS characters in CHAR columns sort a sequence of bytes, according to their weights (first pass) and then according to their code points to break the tie (second pass). First bytes have equal weights, so the code points (0xC1 and 0xE1) break the tie. Therefore, characters D1 and D2 sort before characters D3 and D4. Then the second bytes are compared in similar way, and the final result is as follows:

```
D1 < D2 < D3 < D4
```

Once again, if the data in CHAR columns have the FOR BIT DATA attribute, or if the DBCS characters are stored in a GRAPHIC column, the collation weights are ignored, and characters are compared according to their code points:

```
D1 < D2 < D3 < D4
```

For this particular example, the result happens to be the same as when collation weights were used, but obviously this is not always the case.

Conversion rules for comparison

When two strings are compared, one of the strings is first converted, if necessary, to the encoding scheme, code page, and collating sequence of the other string.

Ordering of results

Results that require sorting are ordered based on the string comparison rules discussed in “String comparisons” on page 66. The comparison is performed at the database server. On returning results to the client application, code page conversion may be performed. This subsequent code page conversion does not affect the order of the server-determined result set.

MBCS considerations for string comparisons

Mixed SBCS/MBCS character strings are compared according to the collating sequence specified when the database was created. For databases created with default (SYSTEM) collation sequence, all single-byte ASCII characters are sorted in correct order, but double-byte characters are not necessarily in code point sequence. For databases created with IDENTITY sequence, all double-byte characters are correctly sorted in their code point order, but single-byte ASCII characters are sorted in their code point order as well. For databases created with COMPATIBILITY sequence, a compromise order is used that sorts properly for most double-byte characters, and is almost correct for ASCII.

Mixed character strings are compared byte-by-byte. This may result in unusual results for multi-byte characters that occur in mixed strings, because each byte is considered independently.

Example:

For this example, 'A', 'B', 'a', and 'b' double-byte characters have the code point values X'8260', X'8261', X'8281', and X'8282', respectively.

Consider a collating sequence where the code points X'8260', X'8261', X'8281', and X'8282' have weights 96, 65, 193, and 194. Then:

```
    'B' < 'A' < 'a' < 'b'
```

and

```
    'AB' < 'AA' < 'Aa' < 'Ab' < 'aB' < 'aA' < 'aa' < 'ab'
```

Graphic string comparisons are processed in a manner analogous to that for character strings.

Graphic string comparisons are valid between all graphic string data types except DBCLOB.

For graphic strings, the collating sequence of the database is not used. Instead, graphic strings are always compared based on the numeric (binary) values of their corresponding bytes.

Using the previous example, if the literals were graphic strings, then:

```
'A' < 'B' < 'a' < 'b'
```

and

```
'AA' < 'AB' < 'Aa' < 'Ab' < 'aA' < 'aB' < 'aa' < 'ab'
```

When comparing graphic strings of unequal lengths, the comparison is made using a logical copy of the shorter string which is padded on the right with double-byte blank characters sufficient to extend its length to that of the longer string.

Two graphic values are equal if they are both empty or if all corresponding graphics are equal. If either operand is null, the result is unknown. If two values are not equal, their relation is determined by a simple binary string comparison.

As indicated in this section, comparing strings on a byte by byte basis can produce unusual results; that is, a result that differs from what would be expected in a character by character comparison. The examples shown here assume the same MBCS code page, however, the situation can be further complicated when using different multi-byte code pages with the same national language. For example, consider the case of comparing a string from a Japanese DBCS code page and a Japanese EUC code page.

Datetime comparisons

A date, time, or timestamp value can be compared with another value of the same data type, a datetime constant of the same data type, or with a string representation of a value of that data type. A date value or a string representation of a date can also be compared with a TIMESTAMP, where the missing time information for the date value is assumed to be all zeros. All comparisons are chronological, which means the further a point in time is from January 1, 0001, the greater the value of that point in time. The time 24:00:00 is greater than the time 00:00:00.

Comparisons that involve time values and string representations of time values always include seconds. If the string representation omits seconds, zero seconds are implied.

Comparisons that involve timestamp values are evaluated according to the following rules:

- When comparing timestamp values with different precisions, the higher precision is used for the comparison and any missing digits for fractional seconds are assumed to be zero.
- When comparing a timestamp value with a string representation of a timestamp, the string representation is first converted to TIMESTAMP(12).
- Timestamp comparisons are chronological without regard to representations that might be considered equivalent. Thus, the following predicate is true:

```
TIMESTAMP('1990-02-23-00.00.00') > '1990-02-22-24.00.00'
```

User-defined type comparisons

Information about comparisons involving user-defined types is in the sections that follow.

Strongly typed distinct type comparisons

Values with a strongly typed distinct type only can be compared with values of exactly the same strongly typed distinct type.

For example, given the following YOUTH distinct type and CAMP_DB_ROSTER table:

```
CREATE TYPE YOUTH AS INTEGER

CREATE TABLE CAMP_DB_ROSTER
   ( NAME                VARCHAR(20),
     ATTENDEE_NUMBER     INTEGER NOT NULL,
     AGE                 YOUTH,
     HIGH_SCHOOL_LEVEL   YOUTH)
```

The following comparison is valid:

```
SELECT * FROM CAMP_DB_ROSTER
   WHERE AGE > HIGH_SCHOOL_LEVEL
```

The following comparison is not valid:

```
SELECT * FROM CAMP_DB_ROSTER
   WHERE AGE > ATTENDEE_NUMBER
```

However, AGE can be compared to ATTENDEE_NUMBER by using a function or CAST specification to cast between the distinct type and the source type. The following comparisons are all valid:

```
SELECT * FROM CAMP_DB_ROSTER
   WHERE INTEGER(AGE) > ATTENDEE_NUMBER

SELECT * FROM CAMP_DB_ROSTER
   WHERE CAST( AGE AS INTEGER) > ATTENDEE_NUMBER

SELECT * FROM CAMP_DB_ROSTER
   WHERE AGE > YOUTH(ATTENDEE_NUMBER)

SELECT * FROM CAMP_DB_ROSTER
   WHERE AGE > CAST(ATTENDEE_NUMBER AS YOUTH)
```

Weakly typed distinct type comparisons

Values with a weakly typed distinct type are compared according to the comparison rules for the source type of the distinct type. Data type constraints have no impact on the comparison. It is valid to compare different weakly typed distinct types if the underlying source types are comparable.

Structured type comparisons

Values with a user-defined structured type cannot be compared with any other value (the NULL predicate and the TYPE predicate can be used).

Array type comparisons

Comparisons of array type values are not supported. Elements of arrays can be compared based on the comparison rules for the data type of the elements.

Row type comparisons

A row variable cannot be compared to another row variable even if the row type name is the same. Individual fields within a row type can be compared to other values and the comparison rules for the data type of the field apply.

Cursor type comparisons

A cursor variable cannot be compared to another cursor variable even if the cursor type name is the same.

Boolean type comparisons

A Boolean value can be compared to another Boolean value or to a value that can be cast to a Boolean value. A value of TRUE is greater than a value of FALSE.

Reference type comparisons

Reference type values can be compared only if their target types have a common supertype. The appropriate comparison function will only be found if the schema name of the common supertype is included in the SQL path. The comparison is performed using the representation type of the reference types. The scope of the reference is not considered in the comparison.

XML comparisons in a non-Unicode database

When performed in a non-Unicode database, comparisons between XML data and character or graphic string values require a code page conversion of one of the two sets of data being compared. Character or graphic values used in an SQL or XQuery statement, either as a query predicate or as a host variable with a character or graphic string data type, are converted to the database code page before comparison. If any characters included in this data have code points that are not part of the database code page, substitution characters are added in their place, potentially causing unexpected results for the query.

For example, a client with a UTF-8 code page is used to connect to a database server created with the Greek encoding ISO8859-7. The expression $\Sigma_G\Sigma_M$ is sent as the predicate of an XQuery statement, where Σ_G represents the Greek sigma character in Unicode (U+03A3) and Σ_M represents the mathematical symbol sigma in Unicode (U+2211). This expression is first converted to the database code page, so that both "Σ" characters are converted to the equivalent code point for sigma in the Greek database code page, 0xD3. We may denote this code point as Σ_A. The newly converted expression $\Sigma_A\Sigma_A$ is then converted again to UTF-8 for comparison with the target XML data. Since the distinction between these two code points was lost as a result of the code page conversion required to pass the predicate expression into the database, the two initially distinct values Σ_G and Σ_M are passed to the XML parser as the expression $\Sigma_G\Sigma_G$. This expression then fails to match when compared to the value $\Sigma_G\Sigma_M$ in an XML document.

One way to avoid the unexpected query results that may be caused by code page conversion issues is to ensure that all characters used in a query expression have matching code points in the database code page. Characters that do not have matching code points can be included through the use of a Unicode character entity reference. A character entity reference will always bypass code page conversion. For example, using the character entity reference ࢣ in place of the Σ_M character ensures that the correct Unicode code point is used for the comparison, regardless of the database code page.

Rules for result data types

The data types of a result are determined by rules which are applied to the operands in an operation. This topic explains those rules.

These rules apply to:

- Corresponding columns in fullselects of set operations (UNION, INTERSECT and EXCEPT)
- Result expressions of a CASE expression and the DECODE and NVL2 scalar functions
- Arguments of the scalar function COALESCE (also NVL and VALUE)
- Arguments of the scalar functions GREATEST, LEAST, MAX, and MIN
- Expression values of the in list of an IN predicate
- Corresponding expressions of a multiple row VALUES clause
- Expression values for the elements in an array constructor
- Arguments of a BETWEEN predicate (except if the data types of all operands are numeric)
- Arguments for the aggregation group ranges in OLAP specifications

These rules are applied subject to other restrictions on long strings for the various operations.

The rules involving various data types follow. In some cases, a table is used to show the possible result data types. The LONG VARCHAR and LONG VARGRAPHIC data types continue to be supported but are deprecated and not recommended.

These tables identify the data type of the result, including the applicable length or precision and scale. The result type is determined by considering the operands. If there is more than one pair of operands,

start by considering the first pair. This gives a result type which is considered with the next operand to determine the next result type, and so on. The last intermediate result type and the last operand determine the result type for the operation. Processing of operations is done from left to right so that the intermediate result types are important when operations are repeated. For example, consider a situation involving:

```
CHAR(2) UNION CHAR(4) UNION VARCHAR(3)
```

The first pair results in a type of CHAR(4). The result values always have 4 bytes. The final result type is VARCHAR(4). Values in the result from the first UNION operation will always have a length of 4.

Character strings

A character string value is compatible with another character string value. Character strings include data types CHAR, VARCHAR, and CLOB.

If one operand is...	And the other operand is...	The data type of the result is...
CHAR(x)	CHAR(y)	CHAR(z) where z = max(x,y)
CHAR(x)	VARCHAR(y)	VARCHAR(z) where z = max(x,y)
VARCHAR(x)	CHAR(y) or VARCHAR(y)	VARCHAR(z) where z = max(x,y)
CLOB(x)	CHAR(y), VARCHAR(y), or CLOB(y)	CLOB(z) where z = max(x,y)

The code page of the result character string will be derived based on the rules for string conversions.

In a Unicode database, if either operand has string units CODEUNITS32 and the derived code page is not 0, the string units of the result character string is CODEUNITS32. If an operand is defined with CODEUNITS32, the other operand cannot be defined as FOR BIT DATA.Otherwise, the string units of the result character string is OCTETS. Special cases apply when the string units of one operand is CODEUNITS32 and the string units of the other operand is OCTETS with a length attribute that exceeds the data type maximum in CODEUNITS32.

If one operand is...	And the other operand is...	The data type of the result is...
CHAR(x OCTETS) with x>63	CHAR(y CODEUNITS32)	VARCHAR(z CODEUNITS32) where z = max(x,y)
VARCHAR(x OCTETS) with x>8168	CHAR(y CODEUNITS32) or VARCHAR(y CODEUNITS32)	Error
CLOB(x OCTETS) with x>536870911	CHAR(y CODEUNITS32), VARCHAR(y CODEUNITS32), or CLOB(y CODEUNITS32)	CLOB(536870911 CODEUNITS32)

Graphic strings

A graphic string value is compatible with another graphic string value. Graphic strings include data types GRAPHIC, VARGRAPHIC, and DBCLOB.

If one operand is...	And the other operand is...	The data type of the result is...
GRAPHIC(x)	GRAPHIC(y)	GRAPHIC(z) where z = max(x,y)
VARGRAPHIC(x)	GRAPHIC(y) OR VARGRAPHIC(y)	VARGRAPHIC(z) where z = max(x,y)
DBCLOB(x)	GRAPHIC(y), VARGRAPHIC(y), or DBCLOB(y)	DBCLOB(z) where z = max (x,y)

The code page of the result graphic string will be derived based on the rules for string conversions.

In a Unicode database, if either operand has string units CODEUNITS32 and the derived code page is not 0, the string units of the result character string is CODEUNITS32. If an operand is defined with CODEUNITS32, the other operand cannot be defined as FOR BIT DATA.Otherwise, the string units of the result character string is OCTETS. Special cases apply when the string units of one operand is CODEUNITS32 and the string units of the other operand is OCTETS with a length attribute that exceeds the data type maximum in CODEUNITS32. In a Unicode database, if either operand has string units CODEUNITS32, or CODEUNITS16, the string units of the result graphic string is CODEUNITS32. Special cases apply when the string units of one operand is CODEUNITS32 and the string units of the other operand is CODEUNITS16 with a length attribute that exceeds the data type maximum in CODEUNITS32.

If one operand is...	And the other operand is...	The data type of the result is...
GRAPHIC(x CODEUNITS16) with x>63	GRAPHIC(y CODEUNITS32)	VARGRAPHIC(z CODEUNITS32) where z = max(x,y)
VARGRAPHIC(x CODEUNITS16) with x>8168	GRAPHIC(y CODEUNITS32) or VARGRAPHIC(y CODEUNITS32)	Error
DBCLOB(x CODEUNITS16) with x>536870911	GRAPHIC(y CODEUNITS32), VARGRAPHIC(y CODEUNITS32) or DBCLOB(y CODEUNITS32)	DBCLOB(536870911 CODEUNITS32)

Character and graphic strings in a Unicode database

In a Unicode database, a character string value is compatible with a graphic string value.

If one operand is...	And the other operand is...	The data type of the result is...
GRAPHIC(x)	CHAR(y) or GRAPHIC(y)	GRAPHIC(z) where z = max(x,y)
VARGRAPHIC(x)	CHAR(y) or VARCHAR(y)	VARGRAPHIC(z) where z = max(x,y)
VARCHAR(x)	GRAPHIC(y) or VARGRAPHIC	VARGRAPHIC(z) where z = max(x,y)
DBCLOB(x)	CHAR(y) or VARCHAR(y) or CLOB(y)	DBCLOB(z) where z = max(x,y)
CLOB(x)	GRAPHIC(y) or VARGRAPHIC(y)	DBCLOB(z) where z = max(x,y)

The string units of the result graphic string will be CODEUNITS32 if either operand has string units CODEUNITS32, or CODEUNITS16. Special cases apply when the string units of one operand is CODEUNITS32 and the string units of the other operand is OCTETS or CODEUNITS16 with a length attribute that exceeds the data type maximum in CODEUNITS32.

If one operand is...	And the other operand is...	The data type of the result is...
CHAR(x OCTETS) with x>63	GRAPHIC(y CODEUNITS32)	VARGRAPHIC(z CODEUNITS32) where z = max(x,y)
GRAPHIC(x CODEUNITS16) with x>63	CHAR(y CODEUNITS32)	VARGRAPHIC(z CODEUNITS32) where z = max(x,y)
VARCHAR(x OCTETS) with x>8168	GRAPHIC(y CODEUNITS32) or VARGRAPHIC(y CODEUNITS32)	Error

If one operand is...	And the other operand is...	The data type of the result is...
VARGRAPHIC(x CODEUNITS16) with x>8168	CHAR(y CODEUNITS32) or VARCHAR(y CODEUNITS32)	Error
CLOB(x OCTETS) with x>536870911	GRAPHIC(y CODEUNITS32), VARGRAPHIC(y CODEUNITS32), or DBCLOB(y CODEUNITS32)	DBCLOB(536870911 CODEUNITS32)
DBCLOB(x CODEUNITS16) with x>536870911	CHAR(y CODEUNITS32), VARCHAR(y CODEUNITS32), or CLOB(y CODEUNITS32)	DBCLOB(536 870 911 CODEUNITS32)

Binary strings

Binary strings are compatible with other binary strings and FOR BIT DATA character strings. Binary strings include BINARY, VARBINARY and BLOB.

Table 13. Operands and the resulting data type

If one operand is...	And the other operand is...	The data type of the result is...
BINARY(x)	BINARY(y) or CHAR(y) FOR BIT DATA	BINARY(z) where z=max(x,y)
VARBINARY(x)	BINARY(y), VARBINARY(y), CHAR(y) FOR BIT DATA, or VARCHAR(y) FOR BIT DATA	VARBINARY(z) where z=max(x,y)
VARCHAR(x) FOR BIT DATA	BINARY(y) or VARBINARY(y)	VARBINARY(z) where z=max(x,y)
BLOB(x)	BINARY(y), VARBINARY(y), BLOB(y), CHAR(y) FOR BIT DATA, or VARCHAR(y) FOR BIT DATA	BLOB(z) where z=max(x,y)

Numeric

Numeric types are compatible with other numeric data types, character-string data types (except CLOB), and in a Unicode database, graphic-string data types (except DBCLOB). Numeric types include SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, DOUBLE, and DECFLOAT.

Table 14. Operands and the resulting data type

If one operand is...	And the other operand is...	The data type of the result is...
SMALLINT	SMALLINT	SMALLINT
SMALLINT	String	DECFLOAT(34)
INTEGER	SMALLINT or INTEGER	INTEGER
INTEGER	String	DECFLOAT(34)
BIGINT	SMALLINT, INTEGER, or BIGINT	BIGINT
BIGINT	String	DECFLOAT(34)
DECIMAL(w,x)	SMALLINT	DECIMAL(p,x) where p = x+max(w-x,5)[1]

Table 14. Operands and the resulting data type (continued)

If one operand is...	And the other operand is...	The data type of the result is...
DECIMAL(w,x)	INTEGER	DECIMAL(p,x) where p = x+max(w-x,11)[1]
DECIMAL(w,x)	BIGINT	DECIMAL(p,x) where p = x+max(w-x,19)[1]
DECIMAL(w,x)	DECIMAL(y,z)	DECIMAL(p,s) where p = max(x,z)+max(w-x,y-z)[1] s = max(x,z)
DECIMAL(w,x)	String	DECFLOAT(34)
REAL	REAL	REAL
REAL	SMALLINT, INTEGER, BIGINT, or DECIMAL	DOUBLE
REAL	String	DECFLOAT(34)
DOUBLE	SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, or DOUBLE	DOUBLE
DOUBLE	String	DECFLOAT(34)
DECFLOAT(n)	SMALLINT, INTEGER, DECIMAL (<=16,s), REAL, or DOUBLE	DECFLOAT(n)
DECFLOAT(n)	BIGINT or DECIMAL (>16,s)	DECFLOAT(34)
DECFLOAT(n)	DECFLOAT(m)	DECFLOAT(MAX(n,m))
DECFLOAT(n)	String	DECFLOAT(34)

[1] Precision cannot exceed 31.

Datetime

Datetime data types are compatible with other operands of the same data type or any CHAR or VARCHAR expression that contains a valid string representation of the same data type. In addition, DATE is compatible with TIMESTAMP and the other operand of a TIMESTAMP can be the string representation of a timestamp or a date. In a Unicode database, character and graphic strings are compatible which implies that GRAPHIC or VARGRAPHIC string representations of datetime values are compatible with other datetime operands.

Table 15. Result data types with datetime operands

If one operand is...	And the other operand is...	The data type of the result is...
DATE	DATE, CHAR(*y*), or VARCHAR(*y*)	DATE
TIME	TIME, CHAR(*y*), or VARCHAR(*y*)	TIME
TIMESTAMP(*x*)	TIMESTAMP(*y*)	TIMESTAMP(*max(x,y)*)
TIMESTAMP(*x*)	DATE, CHAR(*y*), or VARCHAR(*y*)	TIMESTAMP(*x*)

XML

An XML value is compatible with another XML value. The data type of the result is XML.

Boolean

A Boolean value is compatible with the following values or types:

- Another Boolean value.
- Binary integer data types, such as SMALLINT, INTEGER, and BIGINT.
- Character-string data types, except for CLOB.
- Graphic-string data types, except for DBCLOB, in a Unicode database.

The data type of the result is BOOLEAN.

A Boolean value is compatible with another Boolean value. The data type of the result is BOOLEAN.

User-defined types

Distinct types
A strongly typed distinct type value is compatible only with another value of the same distinct type. The data type of the result is the distinct type.

If both operands have the same weakly typed distinct type, the result is the distinct type. Otherwise, if any operand is a weakly typed distinct type then the data type of the operand is considered to be the source data type and the result data type is determined based on the combination of built-in data type operands.

Array data types
A user-defined array data type value is compatible only with another value of the same user-defined array data type. The data type of the result is the user-defined array data type.

Cursor data types
A CURSOR value is compatible with another CURSOR value. The result data type is CURSOR. A user-defined cursor data type value is compatible only with another value of the same user-defined cursor data type. The data type of the result is the user-defined cursor data type.

Row data types
A user-defined row data type value is compatible only with another value of the same user-defined row data type. The data type of the result is the user-defined row data type.

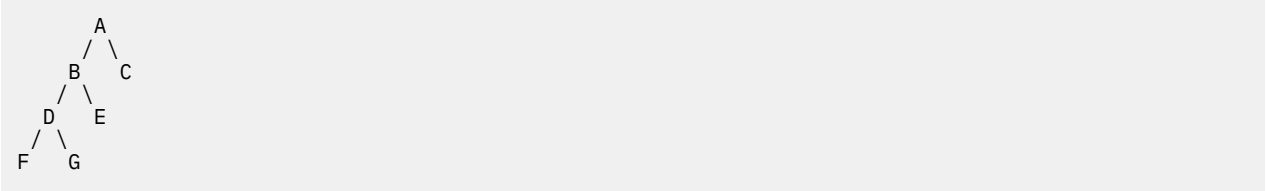
Reference types

A reference type value is compatible with another value of the same reference type provided that their target types have a common supertype. The data type of the result is a reference type having the common supertype as the target type. If all operands have the identical scope table, the result has that scope table. Otherwise the result is unscoped.

Structured types

A structured type value is compatible with another value of the same structured type provided that they have a common supertype. The static data type of the resulting structured type column is the structured type that is the least common supertype of either column.

For example, consider the following structured type hierarchy,



Structured types of the static type E and F are compatible with the resulting static type of B, which is the least common super type of E and F.

Nullable attribute of result

With the exception of INTERSECT and EXCEPT, the result allows nulls unless both operands do not allow nulls.

- For INTERSECT, if either operand does not allow nulls the result does not allow nulls (the intersection would never be null).
- For EXCEPT, if the first operand does not allow nulls the result does not allow nulls (the result can only be values from the first operand).

Rules for string conversions

The code page used to perform an operation is determined by rules which are applied to the operands in that operation. This topic explains those rules.

These rules apply to:

- Corresponding string columns in fullselects with set operations (UNION, INTERSECT and EXCEPT)
- Operands of concatenation
- Operands of predicates (with the exception of LIKE)
- Result expressions of a CASE expression and the DECODE scalar function
- Arguments of the scalar function COALESCE (also NVL and VALUE)
- Arguments of the scalar functions GREATEST, LEAST, MAX, and MIN
- The *source-string* and *insert-string* arguments of the scalar function OVERLAY (and INSERT)
- Expression values of the in list of an IN predicate
- Corresponding expressions of a multiple row VALUES clause.

In each case, the code page of the result is determined at bind time, and the execution of the operation may involve conversion of strings to the code page identified by that code page. A character that has no valid conversion is mapped to the substitution character for the character set and SQLWARN10 is set to 'W' in the SQLCA.

The code page of the result is determined by the code pages of the operands. The code pages of the first two operands determine an intermediate result code page, this code page and the code page of the next operand determine a new intermediate result code page (if applicable), and so on. The last intermediate result code page and the code page of the last operand determine the code page of the result string or column. For each pair of code pages, the result is determined by the sequential application of the following rules:

- If the code pages are equal, the result is that code page.
- If either code page is BIT DATA (code page 0), the result code page is BIT DATA.
- If both code pages are Unicode, and if one code page denotes data in an encoding scheme that is different from the other code page, the result is UTF-16 BE over UTF-8 (that is, the graphic data type over the character data type).
- If one code page is Unicode and the other is neither Unicode nor BIT DATA, the result is Unicode with the same encoding as the Unicode operand (either UTF-16 BE or UTF-8).
- For operands that are host variables (whose code page is not BIT DATA), the result code page is the database code page. Input data from such host variables is converted from the application code page to the database code page before being used.

In a non-Unicode database, if the result code page is Unicode, then the result collation is the alternate collating sequence as defined by the **ALT_COLLATE** database configuration parameter.

Conversions to the code page of the result are performed, if necessary, for:

- An operand of the concatenation operator
- The selected argument of the COALESCE (also NVL and VALUE) scalar function
- The selected argument of the scalar functions GREATEST, LEAST, MAX, and MIN

- The *source-string* and *insert-string* arguments of the scalar function OVERLAY (and INSERT)
- The selected result expression of the CASE expression and the DECODE scalar function
- The expressions of the in list of the IN predicate
- The corresponding expressions of a multiple row VALUES clause
- The corresponding columns involved in set operations.

Character conversion is necessary if all of the following are true:

- The code pages are different
- Neither string is BIT DATA
- The string is neither null nor empty

Examples

Example 1: Given the following in a database created with code page 850:

Expression	Type	Code Page
COL_1	column	850
HV_2	host variable	437

When evaluating the predicate:

```
COL_1 CONCAT :HV_2
```

the result code page of the two operands is 850, because the host variable data will be converted to the database code page before being used.

Example 2: Using information from the previous example when evaluating the predicate:

```
COALESCE(COL_1, :HV_2:NULLIND,)
```

the result code page is 850; therefore, the result code page for the COALESCE scalar function will be code page 850.

String comparisons in a Unicode database

Pattern matching is one area where the behavior of existing MBCS databases is slightly different from the behavior of a Unicode database.

For MBCS databases in Db2®, the current behavior is as follows: If the match-expression contains MBCS data, the pattern can include both SBCS and non-SBCS characters. The special characters in the pattern are interpreted as follows:

- An SBCS halfwidth underscore refers to one SBCS character.
- A non-SBCS fullwidth underscore refers to one non-SBCS character.
- A percent (either SBCS halfwidth or non-SBCS fullwidth) refers to zero or more SBCS or non-SBCS characters.

In a Unicode database, there is really no distinction between "single-byte" and "non-single-byte" characters. Although the UTF-8 format is a "mixed-byte" encoding of Unicode characters, there is no real distinction between SBCS and non-SBCS characters in UTF-8. Every character is a Unicode character, regardless of the number of bytes in UTF-8 format. In a Unicode graphic string, every non-supplementary character, including the halfwidth underscore (U+005F) and halfwidth percent (U+0025), is two bytes in width. For Unicode databases, the special characters in the pattern are interpreted as follows:

- For character strings, a halfwidth underscore (X'5F') or a fullwidth underscore (X'EFBCBF') refers to one Unicode character. A halfwidth percent (X'25') or a fullwidth percent (X'EFBC85') refers to zero or more Unicode characters.

- For graphic strings, a halfwidth underscore (U+005F) or a fullwidth underscore (U+FF3F) refers to one Unicode character. A halfwidth percent (U+0025) or a fullwidth percent (U+FF05) refers to zero or more Unicode characters.

Note: Two underscores are needed to match a Unicode supplementary graphic character because such a character is represented by two UCS-2 characters in a graphic string. Only one underscore is needed to match a Unicode supplementary character in a character string.

For the optional "escape expression", which specifies a character to be used to modify the special meaning of the underscore and percent sign characters, the expression can be specified by any one of:

- A constant
- A special register
- A host variable
- A scalar function whose operands are any of the previously mentioned operands
- An expression concatenating any of the previously mentioned operands or functions

with the restrictions that:

- No element in the expression can be of type CLOB or DBCLOB. In addition, it cannot be a BLOB file reference variable.
- For character strings, the result of the expression must be one character or a FOR BIT DATA string containing exactly one (1) byte (SQLSTATE 22019). For graphic strings, the result of the expression must be one character (SQLSTATE 22019).

Resolving the anchor object for an anchored type

The anchor object of an anchored type that does not specify ROW is specified with a name that could represent an SQL variable, an SQL parameter, a global variable, a module variable, a column of a table, or a column of a view.

The way the anchor object is resolved depends on the number of identifiers in the anchor object name and the context of the statement using the ANCHOR clause.

- If the anchor object name is specified with 1 identifier, the name could represent an SQL variable, an SQL parameter, a module variable, or a global variable.
- If the anchor object name is specified with 2 identifiers, the name could represent a label-qualified SQL variable, a routine-qualified SQL parameter, a schema-qualified global variable, a module variable, the column of a table, or the column of a view.
- If the anchor object name is specified with 3 identifiers, the name represents a column of a schema-qualified table, a column of a schema-qualified view, a global variable qualified by the current server name and a schema, or a schema-qualified module variable.

An SQL variable is a candidate for an anchor object name only if the ANCHOR clause is used in an SQL variable declaration within a compound statement. An SQL parameter is a candidate for an anchor object name only if the ANCHOR clause is used in an SQL variable declaration within a compound statement used in an SQL routine body.

Resolving the anchor object name that has 1 identifier is done using the following steps:

1. If the ANCHOR clause is in an SQL variable declaration of a compound statement, search for a matching SQL variable name starting from the innermost nested compound to the outermost compound.
2. If the ANCHOR clause is in an SQL variable declaration of a compound statement within a routine body, search for a matching SQL parameter name for the routine.
3. If the ANCHOR clause is used in defining a module object, then search for a matching module variable name within the module.
4. If not yet found, then search for a table or view using the first identifier as the schema name and the second identifier as the table or view name.

5. If not yet found, then search for a global variable with a matching global variable name on the SQL path.

Resolving the anchor object name that has 2 identifiers is done using the following steps:

1. If the ANCHOR clause is in an SQL variable declaration of a compound statement, search for a matching qualified SQL variable name starting from the innermost nested compound to the outermost compound.
2. If the ANCHOR clause is in an SQL variable declaration of a compound statement within a routine body, search for a matching SQL parameter name for the routine if the first identifier of the anchor object name matches the name of the routine.
3. If the ANCHOR clause is used in defining a module object and if the first identifier matches the module name of that module, then search for a module variable name within the module that matches the second identifier.
4. If not yet found, then search for a table or view column in the current schema using the first identifier as a table or view name and the second identifier as a column name.
5. If not yet found, then search for a global variable using the first identifier as a schema name and the second identifier as a global variable name.
6. If not found and a module was not searched in step 3, then search for a module on the SQL path with a name that matches the first identifier. If found, then use the second identifier to search for a matching published module variable name in the module.
7. If a module is not found using the SQL path in step 6, check for a module public alias that matches the name of the first identifier. If found, then use the second identifier to search for a matching published module variable name in the module identified by the module public alias.

Resolving the anchor object name that has 3 identifiers is done using the following steps:

1. If the ANCHOR clause is used in defining a module object and if the first 2 identifiers match the schema name and the module name of that module, then search for a module variable with a name that matches the last identifier.
2. If not found in the previous step or the step is not applicable, then search for a table or view column using the first identifier as a schema name, the second identifier as a table or view name and the third identifier as a column name.
3. If not found in the previous step and the first identifier is the same as the current server name, then search for a global variable using the second identifier as a schema name, and the third identifier as a global variable name.
4. If not found and a module was not searched in step 1, then search for a published module variable using the first identifier as a schema name, the second identifier as a module name and, if such a module exists, use the third identifier to search for a matching published module variable name in the module.

Resolving the anchor object for an anchored row type

The anchor object of an anchored type that includes the ROW keyword is specified with a name that could represent a variety of objects, depending on the context and the number of identifiers in the name and the context of the ANCHOR clause.

The objects include the following:

- An SQL variable
- An SQL parameter
- A global variable
- A module variable
- A table
- A view

The way the anchor object is resolved depends on the number of identifiers in the anchor object name and the context of the statement using the ANCHOR clause.

- If the anchor object name is specified with 1 identifier, the name could represent an SQL variable, an SQL parameter, a module variable, a global variable, a table, or a view.
- If the anchor object name is specified with 2 identifiers, the name could represent a label-qualified SQL variable, a routine-qualified SQL parameter, a schema-qualified global variable, a module variable, a schema-qualified table, or a schema-qualified view.
- If the anchor object name is specified with 3 identifiers, the name could represent a global variable qualified by the current server name and a schema, a table qualified by the current server name and a schema, a view qualified by the current server name and a schema, or a schema-qualified module variable.

An SQL variable is a candidate for an anchor object name only if the ANCHOR clause is used in an SQL variable declaration within a compound statement. An SQL parameter is a candidate for an anchor object name only if the ANCHOR clause is used in an SQL variable declaration within a compound statement used in an SQL routine body. Resolving the anchor object name that has 1 identifier is done using the following steps:

1. If the ANCHOR clause is in an SQL variable declaration of a compound statement, search for a matching SQL variable name starting from the innermost nested compound to the outermost compound.
2. If the ANCHOR clause is in an SQL variable declaration of a compound statement within a routine body, search for a matching SQL parameter name for the routine.
3. If the ANCHOR clause is used in defining a module object, then search for a matching module variable name within the module.
4. If not yet found, then search for a table or view with a matching name in the current schema.
5. If not yet found, then search for a schema global variable with a matching global variable name on the SQL path.

Resolving the anchor object name that has 2 identifiers is done using the following steps:

1. If the ANCHOR clause is in an SQL variable declaration of a compound statement, search for a matching qualified SQL variable name starting from the innermost nested compound to the outermost compound.
2. If the ANCHOR clause is in an SQL variable declaration of a compound statement within a routine body, search for a matching SQL parameter name for the routine if the first identifier of the anchor object name matches the name of the routine.
3. If the ANCHOR clause is used in defining a module object and if the first identifier matches the module name of that module, then search for a module variable name within the module that matches the second identifier.
4. If not yet found, then search for a table or view using the first identifier as the schema name and the second identifier as the table or view name.
5. If not yet found, then search for a global variable using the first identifier as a schema name and the second identifier as a global variable name.
6. If not found and a module was not searched in step 3, then search for a module on the SQL path with a name that matches the first identifier. If found, then use the second identifier to search for a matching published module variable name in the module.
7. If a module is not found using the SQL path in step 6, check for a module public alias that matches the name of the first identifier. If found, then use the second identifier to search for a matching published module variable name in the module identified by the module public alias.

Resolving the anchor object name that has 3 identifiers is done using the following steps:

1. If the ANCHOR clause is used in defining a module object and if the first 2 identifiers match the schema name and the module name of that module, then search for a module variable with a name that matches the last identifier.

2. If not found and the first identifier is the same as the current server name, then search for a table or view using the second identifier as the schema name and the third identifier as the table or view name.
3. If not found and the first identifier is the same as the current server name, then search for a global variable using the second identifier as the schema name and the third identifier as the global variable name.
4. If not found and a module was not searched in step 1, then search for a published module variable using the first identifier as a schema name, the second identifier as a module name and, if such a module exists, use the third identifier to search for a matching published variable name in the module.

Database partition-compatible data types

Database partition compatibility is defined between the base data types of corresponding columns of distribution keys. Database partition-compatible data types have the property that two variables, one of each type, with the same value, are mapped to the same distribution map index by the same database partitioning function.

Table 16 on page 83 shows the compatibility of data types in database partitions.

Database partition compatibility has the following characteristics:

- Internal formats are used for DATE, TIME, and TIMESTAMP. They are not compatible with each other, and none are compatible with character or graphic data types.
- Partition compatibility is not affected by the nullability of a column.
- Partition compatibility is affected by collation. Locale-sensitive UCA-based collations require an exact match in collation, except that the strength (S) attribute of the collation is ignored. All other collations are considered equivalent for the purposes of determining partition compatibility.
- Character columns defined with FOR BIT DATA are only compatible with character columns without FOR BIT DATA when a collation other than a locale-sensitive UCA-based collation is used.
- Null values of compatible data types are treated identically. Different results might be produced for null values of non-compatible data types.
- Base data type of the UDT is used to analyze database partition compatibility.
- Timestamps of the same value in the distribution key are treated identically, even if their timestamp precisions differ.
- Decimals of the same value in the distribution key are treated identically, even if their scale and precision differ.
- Trailing blanks in character strings (CHAR, VARCHAR, GRAPHIC or VARGRAPHIC) are ignored by the system-provided hashing function.
- When a locale-sensitive UCA-based collation is used, CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are compatible data types. When other collations are used, CHAR and VARCHAR are compatible types and GRAPHIC and VARGRAPHIC are compatible types, but CHAR and VARCHAR are not compatible types with GRAPHIC and VARGRAPHIC. CHAR or VARCHAR of different lengths are compatible data types.
- DECFLOAT values that are equal are treated identically even if their precision differs. DECFLOAT values that are numerically equal are treated identically even if they have a different number of significant digits.
- Data types that are not supported as part of a distribution key are not applicable for database partition compatibility. Examples of such data types are:
  - BLOB
  - CLOB
  - DBCLOB
  - XML
  - A distinct type based on BLOB, CLOB, DBCLOB, or XML
  - A structured type

Table 16. Database Partition Compatibilities

Operands	Binary Integer	Decimal Number	Floating-point	Decimal Floating-point	Character String	Graphic String	Binary String	Date	Time	Time-stamp	Distinct Type	Boolean
Binary Integer	Yes	No	No	No	No	No	No	No	No	No	1	No
Decimal Number	No	Yes	No	No	No	No	No	No	No	No	1	No
Floating-point	No	No	Yes	No	No	No	No	No	No	No	1	No
Decimal Floating-point	No	No	No	Yes	No	No	No	No	No	No	1	No
Character String	No	No	No	No	Yes[2]	2, 3	No	No	No	No	1	No
Graphic String	No	No	No	No	2, 3	Yes[2]	No	No	No	No	1	No
Binary String							Yes					No
Date	No	No	No	No	No	No	No	Yes	No	No	1	No
Time	No	No	No	No	No	No	No	No	Yes	No	1	No
Timestamp	No	No	No	No	No	No	No	No	No	Yes	1	No
Distinct Type	1	1	1	1	1	1	1	1	1	1	1	1
Boolean	No	No	No	No	No	No	No	No	No	No	1	Yes

Note:

1
A distinct type value is database partition compatible with the source data type of the distinct type or with any other distinct type with the same source data type. The source data type of the distinct type must be a data type that is supported as part of a distribution key. A user-defined distinct type (UDT) value is database partition compatible with the source type of the UDT or any other UDT with a database partition compatible source type. A distinct type cannot be based on BLOB, CLOB, DBCLOB, or XML.

2
Character and graphic string types are compatible when they have compatible collations.

3
Character and graphic string types are compatible when a locale-sensitive UCA-based collation is in effect. Otherwise, they are not compatible types.

Constants

A *constant* (sometimes called a *literal*) specifies a value. Constants are classified as string constants or numeric constants. Numeric constants are further classified as integer, floating-point, or decimal.

All constants have the NOT NULL attribute.

A negative zero value in a numeric constant (-0) is the same value as a zero without the sign (0).

User-defined types have strong typing, except for weakly typed distinct types.. This means that a strongly typed user-defined type is only compatible with its own type. A constant, however, has a built-in type. Therefore, an operation involving a strongly typed user-defined type and a constant is only possible if the user-defined type has been cast to the constant's built-in type, or if the constant has been cast to the user-defined type. For example, using the table and distinct type in “User-defined type comparisons” on page 69, the following comparisons with the constant 14 are valid:

```
SELECT * FROM CAMP_DB_ROSTER
  WHERE AGE > CAST(14 AS YOUTH)
```

```
SELECT * FROM CAMP_DB_ROSTER
  WHERE CAST(AGE AS INTEGER) > 14
```

The following comparison is not valid:

```
SELECT * FROM CAMP_DB_ROSTER
  WHERE AGE > 14
```

Integer constants

An *integer constant* specifies an integer as a signed or unsigned number with a maximum of 19 digits that does not include a decimal point. The data type of an integer constant is large integer if its value is within the range of a large integer. The data type of an integer constant is big integer if its value is outside the range of large integer but within the range of a big integer. A constant that is defined outside the range of big integer values is considered a decimal constant.

Note that the smallest literal representation of a large integer constant is -2,147,483,647 and not -2,147,483,648, which is the limit for integer values. Similarly, the smallest literal representation of a big integer constant is -9,223,372,036,854,775,807 and not -9,223,372,036,854,775,808, which is the limit for big integer values.

Examples:

```
64     -15     +100     32767     720176      12345678901
```

In syntax diagrams, the term "integer" is used for a large integer constant that must not include a sign.

Floating-point constants

A *floating-point constant* specifies a floating-point number as two numbers separated by an E. The first number may include a sign and a decimal point; the second number may include a sign but not a decimal point. The data type of a floating-point constant is double-precision. The value of the constant is the product of the first number and the power of 10 specified by the second number; it must be within the range of floating-point numbers. The number of bytes in the constant must not exceed 30.

Examples:

```
15E1     2.E5     2.2E-1     +5.E+2
```

Decimal constants

A *decimal constant* is a signed or unsigned number that consists of no more than 31 digits and either includes a decimal point or is not within the range of binary integers. It must be within the range of decimal numbers. The precision is the total number of digits (including leading and trailing zeros); the scale is the number of digits to the right of the decimal point (including trailing zeros).

Examples:

```
25.5     1000.     -15.     +37589.3333333333
```

Decimal floating-point constants

There are no decimal floating-point constants except for the decimal floating-point special values, which are interpreted as DECFLOAT(34).

These special values are: INFINITY, NAN, and SNAN. INFINITY represents infinity, a number whose magnitude is infinitely large. INFINITY can be preceded by an optional sign. INF can be specified in place of INFINITY. NAN represents Not a Number (NaN) and is sometimes called quiet NaN. It is a value that represents undefined results which does not cause a warning or exception. SNAN represents signaling NaN (sNaN). It is a value that represents undefined results which will cause a warning or exception if used in any operation that is defined in any numeric operation. Both NAN and SNAN can be preceded by an

optional sign, but the sign is not significant for arithmetic operations.. SNAN can be used in non-numeric operations without causing a warning or exception, for example in the VALUES list of an INSERT or as a constant compared in a predicate.

```
SNAN   -INFINITY
```

When one of the special values (INFINITY, INF, NAN, or SNAN) is used in a context where it could be interpreted as an identifier, such as a column name, cast a string representation of the special value to decimal floating-point. Examples:

```
CAST ('snan' AS DECFLOAT)
CAST ('INF' AS DECFLOAT)
CAST ('Nan' AS DECFLOAT)
```

All non-special values are interpreted as integer, floating-point or decimal constants, in accordance with the rules specified previously. To obtain a numeric decimal floating-point value, use the DECFLOAT cast function with a character string constant. It is not recommended to use floating-point constants as arguments to the DECFLOAT function, because floating-point is not exact and the resulting decimal floating-point value might be different than the decimal digit characters that make up the argument. Instead, use character constants as arguments to the DECFLOAT function.

For example, DECFLOAT('6.0221415E23', 34) returns the decimal floating-point value 6.0221415E+23, but DECFLOAT(6.0221415E23, 34) returns the decimal floating-point value 6.0221415000000003E+23.

Character string constants

A *character string constant* specifies a varying-length character string of type VARCHAR. The constant value string units are determined by the environment default string units. There are three forms of a character string constant:

- A sequence of characters that starts and ends with a string delimiter, which is an apostrophe ('). The number of bytes between the string delimiters cannot be greater than 32672. When the environment string unit is CODEUNITS32, the number of code units cannot be greater than 8168. Two consecutive string delimiters are used to represent one string delimiter within the character string. Two consecutive string delimiters that are not contained within a string represent the empty string.
- X followed by a sequence of characters that starts and ends with a string delimiter. This form of a character string constant is also called a *hexadecimal constant*. The characters between the string delimiters must be an even number of hexadecimal digits. Blanks between the string delimiters are ignored. The number of hexadecimal digits must not exceed 32672. When the environment string unit is CODEUNITS32, the number of code units that the hexadecimal constant represents cannot be greater than 8168. Two consecutive string delimiters are used to represent one string delimiter within the character string. A hexadecimal digit is a digit 0 through 9 or any of the letters A through F (uppercase or lowercase). Under the conventions of hexadecimal notation, each pair of hexadecimal digits represents a byte. The constant is interpreted in the section code page. This form of a character string constant allows you to specify characters that do not have a keyboard representation.
- U& followed by a sequence of characters that starts and ends with a string delimiter and that is optionally followed by the UESCAPE clause. This form of a character string constant is also called a *Unicode string constant*. The number of bytes between the string delimiters cannot be greater than 32672. When the environment string unit is CODEUNITS32, the number of code units that the Unicode string constant represents cannot be greater than 8168. The Unicode string constant is converted from UTF-8 to the section code page during statement compilation. Two consecutive string delimiters are used to represent one string delimiter within the character string. Two consecutive Unicode escape characters are used to represent one Unicode escape character within the character string, but these characters count as one character when calculating the lengths of character constants. Two consecutive string delimiters that are not contained within a string represent the empty string. Because a character in UTF-8 can range from 1 to 4 bytes, a Unicode string constant of the maximum length might actually represent fewer than 32672 characters.

  A character can be expressed by either its typographical character (*glyph*) or its Unicode code point. The code point of a Unicode character ranges from X'000000' to X'10FFFF'. To express a Unicode

character through its code point, use the Unicode escape character followed by 4 hexadecimal digits, or the Unicode escape character followed by a plus sign (+) and 6 hexadecimal digits. The default Unicode escape character is the reverse solidus (\), but a different character can be specified with the UESCAPE clause. The UESCAPE clause is specified as the UESCAPE keyword followed by a single character between string delimiters. The Unicode escape character cannot be a plus sign (+), a double quotation mark ("), a single quotation mark ('), a blank, or any of the characters 0 through 9 or A through F, in either uppercase or lowercase (SQLSTATE 42604). An example of the two ways in which the Latin capital letter A can be specified as a Unicode code point is \0041 and \+000041.

The constant value is always converted to the database code page when it is bound to the database. It is considered to be in the database code page. Therefore, if used in an expression that combines a constant with a FOR BIT DATA column, and whose result is FOR BIT DATA, the constant value will not be converted from its database code page representation when used.

Examples:

```
'12/14/1985'    '32'    'DON''T CHANGE'    ''
X'FFFF'    X'46 72 61 6E 6B'
U&'\0141ód\017A is a city in Poland'   U&'c:\\temp'   U&'@+01D11E' UESCAPE '@'
```

The rightmost string on the second line in the example represents the VARCHAR pattern of the ASCII string "Frank". The last line corresponds to: "Łódź is a city in Poland", c:\temp, and a single character representing the musical symbol G clef.

Graphic string constants

A *graphic string constant* specifies a varying-length graphic string of type VARGRAPHIC.

Non-Unicode databases

In a non-Unicode database, a graphic string constant consists of a sequence of double-byte characters that starts and ends with a single-byte apostrophe ('), and that is preceded by a single-byte G or N. The characters between the apostrophes must represent an even number of bytes, and the length of the graphic string must not exceed 16336 double bytes. The apostrophe must not appear as part of an MBCS character to be considered a delimiter. For example:

```
G'double-byte character string'
N'double-byte character string'
```

Unicode databases

In a Unicode database, a graphic string constant consists of a sequence of characters that starts and ends with an apostrophe ('), and that is preceded by a G or N character. The constant value string units are determined by the environment default string units. The characters between the apostrophes are converted to code page 1200 and the length of the graphic string must not exceed 16336 double bytes. When the environment string unit is CODEUNITS32, the number of code units must not exceed 8168.

In a Unicode or DBCS database, a hexadecimal graphic string constant that specifies a varying-length graphic string is also supported. The format of a hexadecimal graphic string constant is: GX followed by a sequence of characters that starts and ends with an apostrophe. The characters between the apostrophes must be an even multiple of four hexadecimal digits. The number of hexadecimal digits must not exceed 32 672. When the environment string unit is CODEUNITS32, the number of code units that the hexadecimal graphic string constant represents must not exceed 8 168; otherwise, an error is returned (SQLSTATE 54002). If a hexadecimal graphic string constant is improperly formed, an error is returned (SQLSTATE 42606). Each group of four digits represents a single graphic character in the section DBCS code page. In a Unicode database, this would be a single UTF-16 BE graphic character.

Examples:

```
GX'FFFF'
```

represents the bit pattern '1111111111111111' in a Unicode database.

```
GX'005200690063006B'
```

represents the VARGRAPHIC pattern of the ASCII string "Rick" in a Unicode database.

Binary string constants

A binary string constant specifies a varying-length binary string of type VARBINARY.

A binary string constant is formed by specifying a BX followed by a sequence of characters that starts and ends with a string delimiter. The characters between the string delimiters must be an even number of hexadecimal digits. The number of hexadecimal digits must not exceed 32672.

A hexadecimal digit is a digit 0 - 9 or any of the letters A through F (uppercase or lowercase).
Under the conventions of hexadecimal notation, each pair of hexadecimal digits represents 1 byte. This representation is similar to the representation of the character-constant that uses the X'' form. However, binary string constant and character-string constant are not compatible and the X'' form cannot be used to specify a binary string constant, just as the BX'' form cannot be used to specify a character-string constant.

Examples of binary string constants:

```
BX'0000'
BX'C141C242'
BX'FF00FF01FF'
```

Datetime constants

A *datetime constant* specifies a date, time, or timestamp.

Typically, character-string constants are used to represent constant datetime values in assignments and comparisons. However, the associated data type name can be used preceding specific formats of the character-string constant to specifically denote the constant as a datetime constant instead of a character-string constant. The format for the three datetime constants are:

DATE '*yyyy-mm-dd*'
The data type of the value is DATE.

TIME '*hh:mm:ss*'
or
TIME '*hh:mm*'
The data type of the value is TIME.

TIMESTAMP '*yyyy-mm-dd hh:mm:ss.nnnnnnnnnnn*'
or
TIMESTAMP '*yyyy-mm-dd-hh.mm.ss.nnnnnnnnnnn*'
where the number of digits of fractional seconds can vary from 0 to 12 and the period character can be omitted if there are no fractional seconds. The data type of the value is TIMESTAMP(*p*), where *p* is the number of digits of fractional seconds.

Leading zeros can be omitted from the month, day, and hour part of the character-string constant portion, where applicable, in each of these datetime constants. Leading zero characters must be included for minutes and seconds elements of TIME or TIMESTAMP constants. Trailing blanks can be included and are ignored.

UTF-16 BE graphic string constants

A hexadecimal UTF-16 BE graphic string that specifies a varying-length UTF-16 BE graphic string constant is supported. The format of a hexadecimal UTF-16 BE graphic string constant is: UX followed by a sequence of characters that starts and ends with an apostrophe. The characters between the apostrophes must be an even multiple of four hexadecimal digits. The number of hexadecimal digits must not exceed 16336; otherwise, an error is returned (SQLSTATE 54002). If a hexadecimal UTF-16 BE graphic string constant is improperly formed, an error is returned (SQLSTATE 42606). Each group of four digits represents a single UTF-16 BE graphic character.

Example:

```
UX'0042006F006200620079'
```

represents the VARGRAPHIC pattern of the ASCII string "Bobby".

Boolean constants

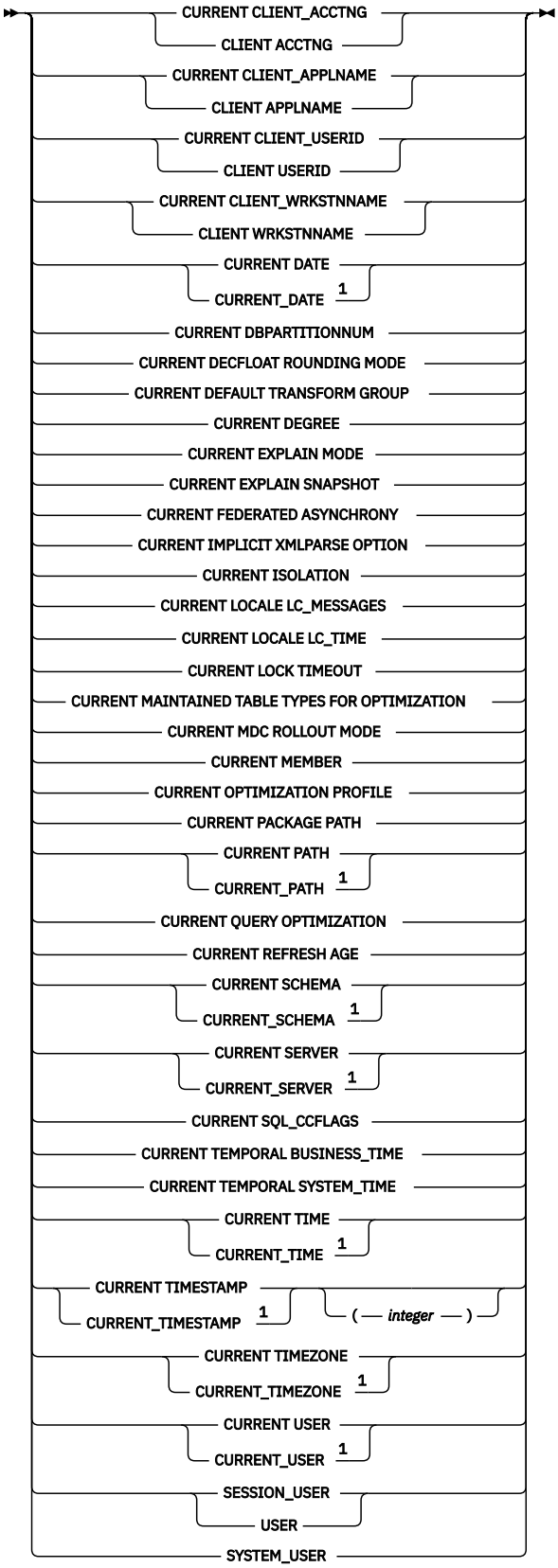
A Boolean constant specifies the keyword TRUE or FALSE, which represents the corresponding truth value. An unknown truth value can be specified using CAST(NULL AS BOOLEAN).

Special registers

A *special register* is a storage area that is defined for an application process by the database manager. It is used to store information that can be referenced in SQL statements.

A reference to a special register is a reference to a value provided by the current server. If the value is a string, its CCSID is a default CCSID of the current server.

The special registers can be referenced as follows:



Notes:

[1] The SQL2008 Core standard uses the form with the underscore.

Some special registers can be updated using the SET statement. The following table shows which of the special registers can be updated as well as indicating which special register can be the null value.

Table 17. Updatable and nullable special registers

Special Register	Updatable	Nullable
CURRENT CLIENT_ACCTNG	No	No
CURRENT CLIENT_APPLNAME	No	No
CURRENT CLIENT_USERID	No	No
CURRENT CLIENT_WRKSTNNAME	No	No
CURRENT DATE	No	No
CURRENT DBPARTITIONNUM	No	No
CURRENT DECFLOAT ROUNDING MODE	No	No
CURRENT DEFAULT TRANSFORM GROUP	Yes	No
CURRENT DEGREE	Yes	No
CURRENT EXPLAIN MODE	Yes	No
CURRENT EXPLAIN SNAPSHOT	Yes	No
CURRENT FEDERATED ASYNCHRONY	Yes	No
CURRENT IMPLICIT XMLPARSE OPTION	Yes	No
CURRENT ISOLATION	Yes	No
“CURRENT LOCALE LC_MESSAGES ” on page 98	Yes	No
CURRENT LOCALE LC_TIME	Yes	No
CURRENT LOCK TIMEOUT	Yes	Yes
CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	Yes	No
CURRENT MDC ROLLOUT MODE	Yes	No
CURRENT MEMBER	No	No
CURRENT OPTIMIZATION PROFILE	Yes	Yes
CURRENT PACKAGE PATH	Yes	No
CURRENT PATH	Yes	No
CURRENT QUERY OPTIMIZATION	Yes	No
CURRENT REFRESH AGE	Yes	No
CURRENT SCHEMA	Yes	No
CURRENT SERVER	No	No
CURRENT SQL_CCFLAGS	Yes	No
CURRENT TEMPORAL BUSINESS_TIME	Yes	Yes
CURRENT TEMPORAL SYSTEM_TIME	Yes	Yes
CURRENT TIME	No	No
CURRENT TIMESTAMP	No	No
CURRENT TIMEZONE	No	No
CURRENT USER	No	No

Table 17. Updatable and nullable special registers (continued)

Special Register	Updatable	Nullable
SESSION_USER	Yes	No
SYSTEM_USER	No	No
USER	Yes	No

When a special register is referenced in a routine, the value of the special register in the routine depends on whether the special register is updatable or not. For non-updatable special registers, the value is set to the default value for the special register. For updatable special registers, the initial value is inherited from the invoker of the routine and can be changed with a subsequent SET statement inside the routine.

CURRENT CLIENT_ACCTNG

The CURRENT CLIENT_ACCTNG (or CLIENT ACCTNG) special register contains the value of the accounting string from the client information specified for this connection.

The data type of the register is VARCHAR(255). The default value of this register is an empty string.

The value of the accounting string can be changed by using the Set Client Information (sqleseti) API or the wlm_set_client_info procedure.

Note that the value provided via the sqleseti API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

Example: Get the current value of the accounting string for this connection.

```
VALUES (CURRENT CLIENT_ACCTNG)
  INTO :ACCT_STRING
```

CURRENT CLIENT_APPLNAME

The CURRENT CLIENT_APPLNAME (or CLIENT APPLNAME) special register contains the value of the application name from the client information specified for this connection.

The data type of the register is VARCHAR(255). The default value of this register is an empty string.

The value of the application name can be changed by using the `sqleseti` API or the **`wlm_set_client_info`** procedure.

Note that the value provided via the `sqleseti` API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

A client sends the default client information register values to the Db2 for z/OS® server when they are not explicitly set by the user. The default **`CURRENT CLIENT_APPLNAME`** special register value is the current process name. The `sqleqryi` API can return the default value only when the **`enableDefaultClientInfo`** keyword is set to `True` in the IBM data server driver configuration file.

Example: Select which departments are allowed to use the application being used in this connection.

```
SELECT DEPT
  FROM DEPT_APPL_MAP
  WHERE APPL_NAME = CURRENT CLIENT_APPLNAME
```

CURRENT CLIENT_USERID

The CURRENT CLIENT_USERID (or CLIENT USERID) special register contains the value of the client user ID from the client information specified for this connection.

The data type of the register is VARCHAR(255). The default value of this register is an empty string.

The value of the client user ID can be changed by using the `sqleseti` API or the **wlm_set_client_info** procedure.

Note that the value provided via the `sqleseti` API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

A client sends the default client information register values to the Db2 for z/OS server when they are not explicitly set by the user. The default **CURRENT CLIENT_USERID** special register value is the user ID that is specified for a connection. The `sqleqryi` API can return the default value only when the **enableDefaultClientInfo** keyword is set to `True` in the IBM data server driver configuration file.

Example: Find out in which department the current client user ID works.

```
SELECT DEPT
  FROM DEPT_USERID_MAP
  WHERE USER_ID = CURRENT CLIENT_USERID
```

CURRENT CLIENT_WRKSTNNAME

The CURRENT CLIENT_WRKSTNNAME (or CLIENT WRKSTNNAME) special register contains the value of the workstation name from the client information specified for this connection.

The data type of the register is VARCHAR(255). The default value of this register is an empty string.

The value of the workstation name can be changed by using the `sqleseti` API or the **wlm_set_client_info** procedure.

Note that the value provided via the `sqleseti` API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

A client sends the default client information register values to the Db2 for z/OS server when they are not explicitly set by the user. The default **CURRENT CLIENT_WRKSTNNAME** special register value is the host name of the client. The `sqleqryi` API can return the default value only when the **enableDefaultClientInfo** keyword is set to `True` in the IBM data server driver configuration file.

Example: Get the workstation name being used for this connection.

```
VALUES (CURRENT CLIENT_WRKSTNNAME)
  INTO :WS_NAME
```

CURRENT DATE

The CURRENT DATE (or CURRENT_DATE) special register specifies a date that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server.

If this special register is used more than once within a single SQL statement, or used with CURRENT TIME or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.

When used in an SQL statement inside a routine, CURRENT DATE is not inherited from the invoking statement.

In a federated system, CURRENT DATE can be used in a query intended for data sources. When the query is processed, the date returned will be obtained from the CURRENT DATE register at the federated server, not from the data sources.

Examples

1. Run the following command from the Db2 CLP to obtain the current date.

```
db2 values CURRENT DATE
```

2. Using the PROJECT table, set the project end date (PRENDATE) of the MA2111 project (PROJNO) to the current date.

```
UPDATE PROJECT
  SET PRENDATE = CURRENT DATE
  WHERE PROJNO = 'MA2111'
```

CURRENT DBPARTITIONNUM

The CURRENT DBPARTITIONNUM special register specifies an INTEGER value that identifies the coordinator database partition number for the statement.

For statements issued from an application, the coordinator is the database partition number to which the application connects.

For statements issued from a routine, the coordinator is the database partition number from which the routine is invoked.

When used in an SQL statement inside a routine, CURRENT DBPARTITIONNUM is never inherited from the invoking statement.

CURRENT DBPARTITIONNUM returns 0 if the database instance is not defined to support database partitioning. For a partitioned database, the db2nodes.cfg file exists and contains database partition and database partition number definitions.

In a database partitioning environment, the CURRENT DBPARTITIONNUM special register can be changed through the CONNECT statement, but only under certain conditions.

Examples

Example 1: Set the host variable APPL_DBPNUM (integer) to the number of the database partition to which the application is connected.

```
VALUES CURRENT DBPARTITIONNUM
  INTO :APPL_DBPNUM
```

Example 2: The following command is issued on member 0 and on a 4 member system in a partitioned database environment. This query will retrieve the currently connected database partition number.

```
VALUES CURRENT DBPARTITIONNUM

1
-----------
          0
```

CURRENT DECFLOAT ROUNDING MODE

The CURRENT DECFLOAT ROUNDING MODE special register specifies the rounding mode that is used for DECFLOAT values.

The data type is VARCHAR(128). The following rounding modes are supported:

- ROUND_CEILING rounds the value toward positive infinity. If all of the discarded digits are zero or if the sign is negative, the result is unchanged (except for the removal of the discarded digits). Otherwise, the result coefficient is incremented by 1.
- ROUND_DOWN rounds the value toward 0 (truncation). The discarded digits are ignored.

- ROUND_FLOOR rounds the value toward negative infinity. If all of the discarded digits are zero or if the sign is positive, the result is unchanged (except for the removal of the discarded digits). Otherwise, the sign is negative and the result coefficient is incremented by 1.
- ROUND_HALF_EVEN rounds the value to the nearest value. If the values are equidistant, rounds the value so that the final digit is even. If the discarded digits represent more than half of the value of a number in the next left position, the result coefficient is incremented by 1. If they represent less than half, the result coefficient is not adjusted (that is, the discarded digits are ignored). Otherwise, the result coefficient is unaltered if its rightmost digit is even, or incremented by 1 if its rightmost digit is odd (to make an even digit).
- ROUND_HALF_UP rounds the value to the nearest value. If the values are equidistant, rounds the value up. If the discarded digits represent half or more than half of the value of a number in the next left position, the result coefficient is incremented by 1. Otherwise, the discarded digits are ignored.

The value of the DECFLOAT rounding mode on a client can be confirmed to match that of the server by invoking the SET CURRENT DECFLOAT ROUNDING MODE statement. However, this statement cannot be used to change the rounding mode of the server. The initial value of CURRENT DECFLOAT ROUNDING MODE is determined by the **decflt_rounding** database configuration parameter and can only be changed by changing the value of this database configuration parameter.

CURRENT DEFAULT TRANSFORM GROUP

The CURRENT DEFAULT TRANSFORM GROUP special register specifies a VARCHAR(18) value that identifies the name of the transform group used by dynamic SQL statements for exchanging user-defined structured type values with host programs.

This special register does not specify the transform groups used in static SQL statements, or in the exchange of parameters and results with external functions or methods.

Its value can be set by the SET CURRENT DEFAULT TRANSFORM GROUP statement. If no value is set, the initial value of the special register is the empty string (a VARCHAR with a length of zero).

In a dynamic SQL statement (that is, one which interacts with host variables), the name of the transform group used for exchanging values is the same as the value of this special register, unless this register contains the empty string. If the register contains the empty string (no value was set by using the SET CURRENT DEFAULT TRANSFORM GROUP statement), the DB2_PROGRAM transform group is used for the transform. If the DB2_PROGRAM transform group is not defined for the structured type subject, an error is raised at run time (SQLSTATE 42741).

Examples

- Set the default transform group to MYSTRUCT1. The TO SQL and FROM SQL functions defined in the MYSTRUCT1 transform are used to exchange user-defined structured type variables with the host program.

```
SET CURRENT DEFAULT TRANSFORM GROUP = MYSTRUCT1
```

- Retrieve the name of the default transform group assigned to this special register.

```
VALUES (CURRENT DEFAULT TRANSFORM GROUP)
```

CURRENT DEGREE

The CURRENT DEGREE special register specifies the degree of intrapartition parallelism for the execution of dynamic SQL statements. (For static SQL, the DEGREE bind option provides the same control.)

The data type of the register is CHAR(5). Valid values are ANY or the string representation of an integer between 1 and 32 767, inclusive.

If the value of CURRENT DEGREE represented as an integer is 1 when an SQL statement is dynamically prepared, the execution of that statement will not use intrapartition parallelism.

If the value of CURRENT DEGREE represented as an integer is greater than 1 and less than or equal to 32 767 when an SQL statement is dynamically prepared, the execution of that statement can involve intrapartition parallelism with the specified degree.

If the value of CURRENT DEGREE is ANY when an SQL statement is dynamically prepared, the execution of that statement can involve intrapartition parallelism using a degree determined by the database manager.

The actual runtime degree of parallelism will be the lower of:

- The value of the maximum query degree (**max_querydegree**) configuration parameter
- The application runtime degree
- The SQL statement compilation degree
- MAXIMUM DEGREE service class option
- MAXIMUM DEGREE workload option

If the **intra_parallel** database manager configuration parameter is set to NO, the value of the CURRENT DEGREE special register will be ignored for the purpose of optimization, and the statement will not use intrapartition parallelism.

If DB2_WORKLOAD=ANALYTICS and MAXIMUM DEGREE for the workload is DEFAULT, the value of the **intra_parallel** setting for the workload is overridden to ON.

The value can be changed by invoking the SET CURRENT DEGREE statement.

The initial value of CURRENT DEGREE is determined by the **dft_degree** database configuration parameter.

The value in the CURRENT DEGREE special register and the **intra_parallel** setting can be overridden in a workload by setting the MAXIMUM DEGREE workload attribute.

CURRENT EXPLAIN MODE

The CURRENT EXPLAIN MODE special register holds a VARCHAR(254) value which controls the behavior of the Explain facility with respect to eligible dynamic SQL statements.

The CURRENT EXPLAIN MODE special register holds a VARCHAR(254) value which controls the behavior of the Explain facility with respect to eligible dynamic SQL statements. This facility generates and inserts Explain information into the Explain tables. This information does not include the Explain snapshot. Possible values are YES, EXPLAIN, NO, REOPT, RECOMMEND INDEXES, and EVALUATE INDEXES. (For static SQL, the **EXPLAIN** bind option provides the same control. In the case of the **PREP** and **BIND** commands, the **EXPLAIN** option values are: YES, NO, and ALL).

YES
Enables the Explain facility and causes Explain information for a dynamic SQL statement to be captured when the statement is compiled.

EXPLAIN
Enables the facility, but dynamic statements are not executed.

NO
Disables the Explain facility.

REOPT
Enables the Explain facility and causes Explain information for a dynamic (or incremental-bind) SQL statement to be captured only when the statement is reoptimized using real values for the input variables (host variables, special registers, global variables, or parameter markers).

RECOMMEND INDEXES
Recommends a set of indexes for each dynamic query. Populates the ADVISE_INDEX table with the set of indexes.

EVALUATE INDEXES
Enables the SQL compiler to evaluate virtual recommended indexes for dynamic queries. Queries executed in this explain mode will be compiled and optimized using fabricated statistics based on

the virtual indexes. The statements are not executed. The indexes to be evaluated are read from the ADVISE_INDEX table if the USE_INDEX column contains "Y". Existing non-unique indexes can also be ignored by setting the USE_INDEX column to "I" and the EXISTS column to "Y". If a combination of USE_INDEX="I" and EXISTS="N" is given then index evaluation for the query will continue normally but the index in question will not be ignored.

The initial value is NO. The value can be changed by invoking the SET CURRENT EXPLAIN MODE statement.

The CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values interact when the Explain facility is invoked. The CURRENT EXPLAIN MODE special register also interacts with the EXPLAIN bind option. RECOMMEND INDEXES and EVALUATE INDEXES can only be set for the CURRENT EXPLAIN MODE register, and must be set using the SET CURRENT EXPLAIN MODE statement.

Example: Set the host variable EXPL_MODE (VARCHAR(254)) to the value currently in the CURRENT EXPLAIN MODE special register.

```
VALUES CURRENT EXPLAIN MODE
  INTO :EXPL_MODE
```

CURRENT EXPLAIN SNAPSHOT

The CURRENT EXPLAIN SNAPSHOT special register holds a CHAR(8) value that controls the behavior of the Explain snapshot facility. This facility generates compressed information, including access plan information, operator costs, and bind-time statistics.

Only the following statements consider the value of this register: CALL, Compound SQL (Dynamic), DELETE, INSERT, MERGE, REFRESH, SELECT, SELECT INTO, SET INTEGRITY, UPDATE, VALUES, or VALUES INTO. Possible values are YES, EXPLAIN, NO, and REOPT. (For static SQL, the EXPLSNAP bind option provides the same control. In the case of the PREP and BIND commands, the EXPLSNAP option values are: YES, NO, and ALL.)

YES
Enables the Explain snapshot facility and takes a snapshot of the internal representation of a dynamic SQL statement as the statement is compiled.

EXPLAIN
Enables the Explain snapshot facility, but dynamic statements are not executed.

NO
Disables the Explain snapshot facility.

REOPT
Enables the Explain facility and causes Explain information for a dynamic (or incremental-bind) SQL statement to be captured only when the statement is reoptimized using real values for the input variables (host variables, special registers, global variables, or parameter markers).

The initial value is NO. The value can be changed by invoking the SET CURRENT EXPLAIN SNAPSHOT statement.

The CURRENT EXPLAIN SNAPSHOT and CURRENT EXPLAIN MODE special register values interact when the Explain facility is invoked. The CURRENT EXPLAIN SNAPSHOT special register also interacts with the EXPLSNAP bind option.

Example: Set the host variable EXPL_SNAP (char(8)) to the value currently in the CURRENT EXPLAIN SNAPSHOT special register.

```
VALUES CURRENT EXPLAIN SNAPSHOT
  INTO :EXPL_SNAP
```

CURRENT FEDERATED ASYNCHRONY

The CURRENT FEDERATED ASYNCHRONY special register specifies the degree of asynchrony for the execution of dynamic SQL statements. The FEDERATED_ASYNCHRONY bind option provides the same control for static SQL.

The data type of the register is INTEGER. Valid values are ANY (representing -1) or an integer between 0 and 32 767, inclusive. If, when an SQL statement is dynamically prepared, the value of CURRENT FEDERATED ASYNCHRONY is:

- 0, the execution of that statement will not use asynchrony
- Greater than 0 and less than or equal to 32 767, the execution of that statement can involve asynchrony using the specified degree
- ANY (representing -1), the execution of that statement can involve asynchrony using a degree that is determined by the database manager

The value of the CURRENT FEDERATED ASYNCHRONY special register can be changed by invoking the SET CURRENT FEDERATED ASYNCHRONY statement.

The initial value of the CURRENT FEDERATED ASYNCHRONY special register is determined by the **federated_async** database manager configuration parameter if the dynamic statement is issued through the command line processor (CLP). The initial value is determined by the FEDERATED_ASYNCHRONY bind option if the dynamic statement is part of an application that is being bound.

Example: Set the host variable FEDASYNC (INTEGER) to the value of the CURRENT FEDERATED ASYNCHRONY special register.

```
VALUES CURRENT FEDERATED ASYNCHRONY INTO :FEDASYNC
```

CURRENT IMPLICIT XMLPARSE OPTION

The CURRENT IMPLICIT XMLPARSE OPTION special register specifies the whitespace handling options that are to be used when serialized XML data is implicitly parsed by the database server, without validation.

An implicit non-validating parse operation occurs when an SQL statement is processing an XML host variable or an implicitly or explicitly typed XML parameter marker that is not an argument of the XMLVALIDATE function. The data type of the register is VARCHAR(19).

The value of the CURRENT IMPLICIT XMLPARSE OPTION special register can be changed by invoking the SET CURRENT IMPLICIT XMLPARSE OPTION statement. Its initial value is 'STRIP WHITESPACE'.

Examples

- Retrieve the value of the CURRENT IMPLICIT XMLPARSE OPTION special register into a host variable named CURXMLPARSEOPT:

  ```
  EXEC SQL VALUES (CURRENT IMPLICIT XMLPARSE OPTION) INTO :CURXMLPARSEOPT;
  ```

- Set the CURRENT IMPLICIT XMLPARSE OPTION special register to 'PRESERVE WHITESPACE'.

  ```
  SET CURRENT IMPLICIT XMLPARSE OPTION = 'PRESERVE WHITESPACE'
  ```

  Whitespace is then preserved when the following SQL statement executes:

  ```
  INSERT INTO T1 (XMLCOL1) VALUES (?)
  ```

CURRENT ISOLATION

The CURRENT ISOLATION special register holds a CHAR(2) value that identifies the isolation level (in relation to other concurrent sessions) for any dynamic SQL statements issued within the current session.

The possible values are:

(blanks)
Not set; use the isolation attribute of the package.

UR
Uncommitted Read

CS
Cursor Stability

RR
Repeatable Read

RS
Read Stability

The value of the CURRENT ISOLATION special register can be changed by the SET CURRENT ISOLATION statement.

Until a SET CURRENT ISOLATION statement is issued within a session, or after RESET has been specified for SET CURRENT ISOLATION, the CURRENT ISOLATION special register is set to blanks and is not applied to dynamic SQL statements; the isolation level used is taken from the isolation attribute of the package which issued the dynamic SQL statement. Once a SET CURRENT ISOLATION statement has been issued, the CURRENT ISOLATION special register provides the isolation level for any subsequent dynamic SQL statement compiled within the session, regardless of the settings for the package issuing the statement. This will remain in effect until the session ends or until a SET CURRENT ISOLATION statement is issued with the RESET option.

Example: Set the host variable ISOLATION_MODE (CHAR(2)) to the value currently stored in the CURRENT ISOLATION special register.

```
VALUES CURRENT ISOLATION
  INTO :ISOLATION_MODE
```

CURRENT LOCALE LC_MESSAGES

The CURRENT LOCALE LC_MESSAGES special register identifies the locale that is used by EVMON_UPGRADE_TABLES as well as monitoring routines in the **monreport** module.

EVMON_UPGRADE_TABLES and the monitoring routines use the value of CURRENT LOCALE LC_MESSAGES to determine in which language the result set text output should be returned. User-defined routines that are coded to return messages could also use the value of CURRENT LOCALE LC_MESSAGES to determine what language to use for message text.

The data type is VARCHAR(128).

The initial value of CURRENT LOCALE LC_MESSAGES is "en_US" for English (United States). The value can be changed by invoking the SET CURRENT LOCALE LC_MESSAGES statement.

CURRENT LOCALE LC_TIME

The CURRENT LOCALE LC_TIME special register identifies the locale that is used for SQL statements that involve the datetime related built-in functions.

These functions include DAYNAME, MONTHNAME, NEXT_DAY, ROUND, ROUND_TIMESTAMP, TIMESTAMP_FORMAT, TRUNCATE, TRUNC_TIMESTAMP and VARCHAR_FORMAT. These functions use the value of CURRENT LOCALE LC_TIME if the *locale-name* argument is not explicitly specified.

The data type is VARCHAR(128).

The initial value of CURRENT LOCALE LC_TIME is "en_US" for English (United States). The value can be changed by invoking the SET CURRENT LOCALE LC_TIME statement.

CURRENT LOCK TIMEOUT

The CURRENT LOCK TIMEOUT special register specifies the number of seconds to wait for a lock before returning an error indicating that a lock cannot be obtained. This special register impacts row, table, alter table, online backup, keyvalue, MDC block, reorg, XML path, extent movement, plan, variation, catalog, insert range, dictionary and table serialize locks.

The data type of the register is INTEGER.

Valid values for the CURRENT LOCK TIMEOUT special register are integers between -1 and 32767, inclusive. This special register can also be set to the null value. A value of -1 specifies that timeouts are not to take place, and that the application is to wait until the lock is released or a deadlock is detected. A value of 0 specifies that the application is not to wait for a lock; if a lock cannot be obtained, an error is to be returned immediately.

The value of the CURRENT LOCK TIMEOUT special register can be changed by invoking the SET CURRENT LOCK TIMEOUT statement. The initial value is null; in this case, the current value of the **locktimeout** database configuration parameter is used when waiting for a lock, and this value is returned for the special register.

CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION

The CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register specifies a VARCHAR(254) value that identifies the types of tables that can be considered when optimizing the processing of dynamic SQL queries. Materialized query tables are never considered by static embedded SQL queries.

The initial value of CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION is set to the value of the **dft_mttb_types** database configuration parameter. The default value of **dft_mttb_types** parameter is SYSTEM.

CURRENT MDC ROLLOUT MODE

The CURRENT MDC ROLLOUT MODE special register specifies the behavior on multidimensional clustering (MDC) tables of DELETE statements that qualify for rollout processing.

The default value of this register is determined by the DB2_MDC_ROLLOUT registry variable. The value can be changed by invoking the SET CURRENT MDC ROLLOUT MODE statement. When the CURRENT MDC ROLLOUT MODE special register is set to a particular value, the execution behavior of subsequent DELETE statements that qualify for rollout is impacted. The DELETE statement does not need to be recompiled for the behavior to change.

CURRENT MEMBER

The CURRENT MEMBER special register specifies an INTEGER value that identifies the coordinator member for the statement.

For statements issued from an application, the coordinator is the member to which the application connects. For statements issued from a routine, the coordinator is the member from which the routine is invoked.

When used in an SQL statement inside a routine, CURRENT MEMBER is never inherited from the invoking statement.

CURRENT MEMBER returns 0 if the database instance is not defined to support database partitioning or the IBM Db2 pureScale® Feature. The database instance is not defined for such support if there is no db2nodes.cfg file. For a partitioned database or a Db2 pureScale environment, the db2nodes.cfg file exists and contains database partition and member definitions.

CURRENT MEMBER can be changed through the CONNECT statement, but only under certain conditions.

For compatibility with previous versions of Db2 and with other database products, NODE can be specified in place of MEMBER.

Examples

Example 1: Set the host variable APPL_NODE (integer) to the number of the member to which the application is connected.

```
VALUES CURRENT MEMBER
  INTO :APPL_NODE
```

Example 2: The following command is issued on member 0 and on a 4 member system in a partitioned database environment. This query will retrieve the currently connected database member number.

```
VALUES CURRENT MEMBER

1
-----------
          0
```

CURRENT OPTIMIZATION PROFILE

The CURRENT OPTIMIZATION PROFILE special register specifies the qualified name of the optimization profile to be used by DML statements that are dynamically prepared for optimization.

The initial value is the null value. The value can be changed by invoking the SET CURRENT OPTIMIZATION PROFILE statement. An optimization profile that is not qualified with a schema name will be implicitly qualified with the value of the CURRENT DEFAULT SCHEMA special register.

Example 1: Set the optimization profile to 'JON.SALES'.

```
SET CURRENT OPTIMIZATION PROFILE = JON.SALES
```

Example 2: Get the current value of the optimization profile name for this connection.

```
VALUES (CURRENT OPTIMIZATION PROFILE) INTO :PROFILE
```

CURRENT PACKAGE PATH

The CURRENT PACKAGE PATH special register specifies a VARCHAR(4096) value that identifies the path to be used when resolving references to packages that are needed when executing SQL statements.

The value can be an empty or a blank string, or a list of one or more schema names that are delimited with double quotation marks and separated by commas. Any double quotation marks appearing as part of the string will need to be represented as two double quotation marks, as is common practice with delimited identifiers. The delimiters and commas contribute to the length of the special register.

This special register applies to both static and dynamic statements.

The initial value of CURRENT PACKAGE PATH in a user-defined function, method, or procedure is inherited from the invoking application. In other contexts, the initial value of CURRENT PACKAGE PATH is an empty string. The value is a list of schemas only if the application process has explicitly specified a list of schemas by means of the SET CURRENT PACKAGE PATH statement.

Examples

- An application will be using multiple SQLJ packages (in schemas SQLJ1 and SQLJ2) and a JDBC package (in schema DB2JAVA). Set the CURRENT PACKAGE PATH special register to check SQLJ1, SQLJ2, and DB2JAVA, in that order.

  ```
  SET CURRENT PACKAGE PATH = "SQLJ1", "SQLJ2", "DB2JAVA"
  ```

- Set the host variable HVPKLIST to the value currently stored in the CURRENT PACKAGE PATH special register.

```
VALUES CURRENT PACKAGE PATH INTO :HVPKLIST
```

CURRENT PATH

The CURRENT PATH (or CURRENT_PATH) special register specifies a VARCHAR(2048) value that identifies the SQL path used when resolving unqualified function names, procedure names, data type names, global variable names, and module object names in dynamically prepared SQL statements. CURRENT FUNCTION PATH is a synonym for CURRENT PATH.

The initial value is the default value specified in a following paragraph. For static SQL, the FUNCPATH bind option provides an SQL path that is used for function and data type resolution.

The CURRENT PATH special register contains a list of one or more schema names that are enclosed by double quotation marks and separated by commas. For example, an SQL path specifying that the database manager is to look first in the FERMAT schema, then in the XGRAPHIC schema, and finally in the SYSIBM schema, is returned in the CURRENT PATH special register as:

```
"FERMAT","XGRAPHIC","SYSIBM"
```

The default value is "SYSIBM","SYSFUN","SYSPROC","SYSIBMADM",*X*, where *X* is the value of the USER special register, delimited by double quotation marks. The value can be changed by invoking the SET CURRENT PATH statement. The schema SYSIBM does not need to be specified. If it is not included in the SQL path, it is implicitly assumed to be the first schema. SYSIBM does not take up any of the 2048 bytes if it is implicitly assumed.

A data type that is not qualified with a schema name will be implicitly qualified with the first schema in the SQL path that contains a data type with the same unqualified name. There are exceptions to this rule, as outlined in the descriptions of the following statements: CREATE TYPE (Distinct), CREATE FUNCTION, COMMENT, and DROP.

Example: Using the SYSCAT.ROUTINES catalog view, find all user-defined routines that can be invoked without qualifying the routine name, because the CURRENT PATH special register contains the schema name.

```
SELECT ROUTINENAME, ROUTINESCHEMA FROM SYSCAT.ROUTINES
  WHERE POSITION (ROUTINESCHEMA, CURRENT PATH, CODEUNITS16) <> 0
```

CURRENT QUERY OPTIMIZATION

The CURRENT QUERY OPTIMIZATION special register specifies an INTEGER value that controls the class of query optimization performed by the database manager when binding dynamic SQL statements.

The QUERYOPT bind option controls the class of query optimization for static SQL statements. The possible values range from 0 to 9. For example, if the query optimization class is set to 0 (minimal optimization), then the value in the special register is 0. The default value is determined by the **dft_queryopt** database configuration parameter. The value can be changed by invoking the SET CURRENT QUERY OPTIMIZATION statement.

Example: Using the SYSCAT.PACKAGES catalog view, find all plans that were bound with the same setting as the current value of the CURRENT QUERY OPTIMIZATION special register.

```
SELECT PKGNAME, PKGSCHEMA FROM SYSCAT.PACKAGES
  WHERE QUERYOPT = CURRENT QUERY OPTIMIZATION
```

CURRENT REFRESH AGE

The CURRENT REFRESH AGE special register specifies a time stamp duration value with a data type of DECIMAL(20,6).

The value of the CURRENT REFRESH AGE special register must be 0 - 99999999999999. The default value is determined by the **dft_refresh_age** database configuration parameter. You can change the value by issuing the SET CURRENT REFRESH AGE statement.

The value represents the maximum duration since a particular time-stamped event occurred to a cached data object. The cached data object can be used during this period to help optimize the processing of a query. An example of a time-stamped event is processing a REFRESH TABLE statement on a system-maintained REFRESH DEFERRED materialized query table.

For example, if the CURRENT REFRESH AGE special register has a value of 99999999999999 and the query optimization class is 2 or is greater than or equal to 5, the types of refresh deferred materialized query tables that you specify for the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register are considered during optimization of the processing of a dynamic SQL query.

CURRENT SCHEMA

The CURRENT SCHEMA (or CURRENT_SCHEMA) special register specifies a VARCHAR(128) value that identifies the schema name used to qualify database object references, where applicable, in dynamically prepared SQL statements.

For compatibility with Db2 for z/OS, CURRENT SQLID (or CURRENT_SQLID) can be specified in place of CURRENT SCHEMA.

The initial value of CURRENT SCHEMA is the authorization ID of the current session user. The value can be changed by invoking the SET SCHEMA statement.

The setting of CURRENT SCHEMA does not affect the Explain facility's selection of explain tables.

The QUALIFIER bind option controls the schema name used to qualify database object references, where applicable, for static SQL statements.

Example: Set the schema for object qualification to 'D123'.

```
SET CURRENT SCHEMA =  'D123'
```

CURRENT SERVER

The CURRENT SERVER (or CURRENT_SERVER) special register specifies a VARCHAR(18) value that identifies the current database server (sometimes referred to as the application server). The register contains the actual name of the database, not an alias.

CURRENT SERVER can be changed through the CONNECT statement, but only under certain conditions.

When used in an SQL statement inside a routine, CURRENT SERVER is not inherited from the invoking statement.

Example: Set the host variable APPL_SERVE (VARCHAR(18)) to the name of the database server to which the application is connected.

```
VALUES CURRENT SERVER INTO :APPL_SERVE
```

CURRENT SQL_CCFLAGS

The CURRENT SQL_CCFLAGS special register specifies the conditional compilation named constants that are defined for use during compilation of SQL statements.

The data type of the special register is VARCHAR(1024).

The CURRENT SQL_CCFLAGS special register contains a list of name and value pairs separated by a comma and a blank. The name is separated from the value in a pair using the colon character. The values in the list are a BOOLEAN constant, an INTEGER constant, or the keyword NULL. The names can be specified using any combination of uppercase or lowercase characters which are folded to all uppercase characters. For example, conditional compilation values defined for debug and tracing could appear in the special register as the string value:

```
    CC_DEBUG:TRUE, CC_TRACE_LEVEL:2
```

The initial value of the special register is the value of the **sql_ccflags** database configuration parameter when the special register is first used. The first use can occur as a result of processing a statement with an inquiry directive or as a direct reference to the special register. If the value assigned to the **sql_ccflags** database configuration parameter is not valid, an error is returned on the first use (SQLSTATE 42815 or 428HV).

The value of the special register can be changed by executing the SET CURRENT SQL_CCFLAGS statement.

CURRENT TEMPORAL BUSINESS_TIME

The CURRENT TEMPORAL BUSINESS_TIME special register specifies a TIMESTAMP(12) value that is used in the default BUSINESS_TIME period specification for references to application-period temporal tables.

When an application-period temporal table is referenced and the value in effect for the CURRENT TEMPORAL BUSINESS_TIME special register is represented by *CTBT*, which is the non-null value, the following period specification is implicit:

```
FOR BUSINESS_TIME AS OF CTBT
```

When an application-period temporal table is the target of an UPDATE or DELETE statement and the value in effect for the CURRENT TEMPORAL BUSINESS_TIME special register is not the null value, the following additional predicate is implicit:

```
bt_begin <= CURRENT TEMPORAL BUSINESS_TIME
    AND bt_end > CURRENT TEMPORAL BUSINESS_TIME
```

where `bt_begin` and `bt_end` are the begin and end columns of the BUSINESS_TIME period of the target table of the UPDATE statement.

The initial value of the special register in a user-defined function or procedure is inherited from the invoking application. In other contexts the initial value of the special register is the null value.

The value of this special register can be changed by executing the SET CURRENT TEMPORAL BUSINESS_TIME statement.

The setting of the CURRENT TEMPORAL BUSINESS_TIME special register impacts the following compiled SQL objects when the associated package is bound with the BUSTIMESENSITIVE bind option set to YES:

- SQL procedures
- Compiled functions
- Compiled triggers
- Compound SQL (compiled) statements
- External UDFs

The setting for the BUSTIMESENSITIVE bind option determines whether references to application-period temporal tables and bitemporal tables in both static SQL statements and dynamic SQL statements in a package are affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register. The bind option can be set to YES or NO.

For the following examples, assume the table IN_TRAY is an application-period temporal table.

Example 1: Based on the state of the messages in IN_TRAY as of the date specified by the CURRENT TEMPORAL BUSINESS_TIME special register, list the user IDs and subject lines.

```
SELECT SOURCE, SUBJECT FROM IN_TRAY
```

Assuming that the CURRENT TEMPORAL BUSINESS_TIME special register was previously set to the value CURRENT TIMESTAMP-4 DAYS and is currently set to the null value, the following statement returns the same result.

```
SELECT SOURCE, SUBJECT
    FROM IN_TRAY
    FOR BUSINESS_TIME AS OF CURRENT TIMESTAMP-4 DAYS
```

Example 2: List the user ID and subject line for the messages in IN_TRAY sent before the date specified by the CURRENT TEMPORAL BUSINESS_TIME special register.

```
SELECT SOURCE, SUBJECT
    FROM IN_TRAY
    WHERE DATE(RECEIVED) < DATE(CURRENT TEMPORAL BUSINESS_TIME)
```

Assuming that the CURRENT TEMPORAL BUSINESS_TIME special register was previously set to '2011-01-01-00.00.00' and is currently set to the null value, the following statement returns the same result.

```
SELECT SOURCE, SUBJECT
   FROM IN_TRAY
   FOR BUSINESS_TIME AS OF '2011-01-01-00.00.00'
   WHERE DATE(RECEIVED) < DATE('2011-01-01-00.00.00')
```

CURRENT TEMPORAL SYSTEM_TIME

The CURRENT TEMPORAL SYSTEM_TIME special register specifies a TIMESTAMP(12) value that is used in the default SYSTEM_TIME period specification for references to system-period temporal tables.

When a system-period temporal table is referenced and the value in effect for the CURRENT TEMPORAL SYSTEM_TIME special register is represented by *CTST*, which is the non-null value, , the following period specification is implicit:

```
FOR SYSTEM_TIME AS OF CTST
```

The initial value of the special register in a user-defined function or procedure is inherited from the invoking application. In other contexts the initial value of the special register is the null value.

The value of this special register can be changed by executing the SET CURRENT TEMPORAL SYSTEM_TIME statement.

The setting of the CURRENT TEMPORAL SYSTEM_TIME special register impacts the following compiled SQL objects when they have been bound with the SYSTIMESENSITIVE bind option set to YES:

- SQL procedures
- Compiled functions
- Compiled triggers
- Compound SQL (compiled) statements
- External UDFs)

The setting for the SYSTIMESENSITIVE bind option determines whether references to system-period temporal tables in both static SQL statements and dynamic SQL statements in a package are affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register. The bind option can be set to YES or NO.

For the following examples, assume the table IN_TRAY is a system-period temporal table.

Example 1: Based on the state of the messages in IN_TRAY as of the date specified by the CURRENT TEMPORAL SYSTEM_TIME special register, list the user IDs and subject lines.

```
SELECT SOURCE, SUBJECT
    FROM IN_TRAY
```

Assuming that the CURRENT TEMPORAL SYSTEM_TIME special register was previously set to the value CURRENT TIMESTAMP-7 DAYS and is currently set to the null value, the following statement returns the same result.

```
SELECT SOURCE, SUBJECT
    FROM IN_TRAY
    FOR SYSTEM_TIME AS OF CURRENT TEMPORAL SYSTEM_TIME
```

Example 2: List the user ID and subject line for the messages in IN_TRAY sent before the value specified by the CURRENT TEMPORAL SYSTEM_TIME special register.

```
SELECT SOURCE, SUBJECT
    FROM IN_TRAY
    WHERE RECEIVED < CURRENT TEMPORAL SYSTEM_TIME
```

Assuming that the CURRENT TEMPORAL SYSTEM_TIME special register was previously set to '2011-01-01-00.00.00' and is currently set to the null value, the following statement returns the same result.

```
SELECT SOURCE, SUBJECT
    FROM IN_TRAY
    FOR SYSTEM_TIME AS OF '2011-01-01-00.00.00'
    WHERE DATE(RECEIVED) < DATE('2011-01-01-00.00.00')
```

CURRENT TIME

The CURRENT TIME (or CURRENT_TIME) special register specifies a time that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server.

If this special register is used more than once within a single SQL statement, or used with CURRENT DATE or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.

When used in an SQL statement inside a routine, CURRENT TIME is not inherited from the invoking statement.

In a federated system, CURRENT TIME can be used in a query intended for data sources. When the query is processed, the time returned will be obtained from the CURRENT TIME register at the federated server, not from the data sources.

Examples

1. Run the following command from the Db2 CLP to obtain the current time.

   ```
   db2 values CURRENT TIME
   ```

2. Using the CL_SCHED table, select all the classes (CLASS_CODE) that start (STARTING) later today. Today's classes have a value of 3 in the DAY column.

   ```
   SELECT CLASS_CODE FROM CL_SCHED
     WHERE STARTING > CURRENT TIME AND DAY = 3
   ```

CURRENT TIMESTAMP

The CURRENT TIMESTAMP (or CURRENT_TIMESTAMP) special register specifies a timestamp that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server.

If this special register is used more than once within a single SQL statement, or used with CURRENT DATE or CURRENT TIME within a single statement, all values are based on a single clock reading. It is possible

for separate CURRENT TIMESTAMP special register requests to return the same value; if unique values are required, consider using the GENERATE_UNIQUE function, a sequence, or an identity column.

If a timestamp with a specific precision is desired, the special register can be referenced as CURRENT TIMESTAMP(*integer*), where *integer* can range from 0 to 12. The default precision is 6. The precision of the clock reading varies by platform and the resulting value is padded with zeros where the precision of the retrieved clock reading is less than the precision of the request.

For example:

- CURRENT TIMESTAMP()
  - Output in Windows: 2015-03-23-09.41.24.684000
  - Output in Linux: 2015-03-23-09.41.24.684842
  - Output in UNIX: 2015-03-23-09.41.24.684842
- CURRENT TIMESTAMP(12)
  - Output in Windows: 2015-03-23-09.41.24.684000000000
  - Output in Linux: 2015-03-23-09.41.24.684842000000
  - Output in UNIX: 2015-03-23-09.41.24.684842000000
- CURRENT TIMESTAMP(6)
  - Output in Windows: 2015-03-23-09.41.24.684000
  - Output in Linux: 2015-03-23-09.41.24.684842
  - Output in UNIX: 2015-03-23-09.41.24.684842
- CURRENT TIMESTAMP(3)
  - Output in Windows: 2015-03-23-09.41.24.684
  - Output in Linux: 2015-03-23-09.41.24.684
  - Output in UNIX: 2015-03-23-09.41.24.684

When used in an SQL statement inside a routine, CURRENT TIMESTAMP is not inherited from the invoking statement.

In a federated system, CURRENT TIMESTAMP can be used in a query intended for data sources. When the query is processed, the timestamp returned will be obtained from the CURRENT TIMESTAMP register at the federated server, not from the data sources.

On a Db2 pureScale instance, with transaction workload balancing enabled, the CURRENT TIMESTAMP special register does not necessarily return increasing values across transactions if those transactions are executed on different members.

SYSDATE can also be specified as a synonym for CURRENT TIMESTAMP(0).

LOCALTIMESTAMP can also be specified as a synonym for CURRENT TIMESTAMP.

LOCALTIMESTAMP(*integer*) can also be specified as a synonym for CURRENT TIMESTAMP(*integer*).

Example: Insert a row into the IN_TRAY table. The value of the RECEIVED column should be a timestamp that indicates when the row was inserted. The values for the other three columns come from the host variables SRC (char(8)), SUB (char(64)), and TXT (VARCHAR(200)).

```
INSERT INTO IN_TRAY
  VALUES (CURRENT TIMESTAMP, :SRC, :SUB, :TXT)
```

CURRENT TIMEZONE

The CURRENT TIMEZONE (or CURRENT_TIMEZONE) special register specifies the difference between UTC (Coordinated Universal Time, formerly known as GMT) and local time at the application server.

The difference is represented by a time duration (a decimal number in which the first two digits are the number of hours, the next two digits are the number of minutes, and the last two digits are the number of seconds). The number of hours is between -24 and 24 exclusive. Subtracting CURRENT TIMEZONE from a local time converts that local time to UTC. The time is calculated from the operating system time at the moment the SQL statement is executed. (The CURRENT TIMEZONE value is determined from C runtime functions.)

The CURRENT TIMEZONE special register can be used wherever an expression of the DECIMAL(6,0) data type is used; for example, in time and timestamp arithmetic.

When used in an SQL statement inside a routine, CURRENT TIMEZONE is not inherited from the invoking statement.

Example: Insert a record into the IN_TRAY table, using a UTC timestamp for the RECEIVED column.

```
INSERT INTO IN_TRAY VALUES (
  CURRENT TIMESTAMP - CURRENT TIMEZONE,
  :source,
  :subject,
  :notetext )
```

CURRENT USER

The CURRENT USER (or CURRENT_USER) special register specifies the authorization ID that is used for statement authorization for the statement in which it was referenced.

For dynamic SQL statements, the value depends on the dynamic SQL statement behavior in effect for the package issuing the dynamic SQL statement in which this special register is referenced. See "Effect of DYNAMICRULES bind option on dynamic SQL" for details. The data type of the register is VARCHAR(128). If the length of the authorization ID is less than 8 bytes, the special register value is padded with trailing blanks such that the length is 8 bytes.

Example: Select table names whose schema matches the value of the CURRENT USER special register.

```
SELECT TABNAME FROM SYSCAT.TABLES
  WHERE TABSCHEMA = CURRENT USER AND TYPE = 'T'
```

If this statement is executed as a static SQL statement, it returns the tables whose schema name matches the binder of the package that includes the statement. If this statement is executed as a dynamic SQL statement using dynamic SQL statement run behavior, it returns the tables whose schema name matches the current value of the SESSION_USER special register.

SESSION_USER

The SESSION_USER special register specifies the current runtime authorization ID that is being used for the current session.

The data type of the register is VARCHAR(128). If the length of the authorization ID is less than 8 bytes, the special register value is padded with trailing blanks such that the length is 8 bytes.

The initial value of SESSION_USER for a new connection is the same as the value of the SYSTEM_USER special register. Its value can be changed by invoking the SET SESSION AUTHORIZATION statement.

SESSION_USER is a synonym for the USER special register.

Example: Determine what routines can be executed by current runtime authorization ID if it were to issue invocations through dynamic SQL.

```
SELECT SCHEMA, SPECIFICNAME FROM SYSCAT.ROUTINEAUTH
  WHERE GRANTEE = SESSION_USER
  AND EXECUTEAUTH IN ('Y', 'G')
```

SYSTEM_USER

The SYSTEM_USER special register specifies the authorization ID of the user that connected to the database.

The value of this register can only be changed by connecting as a user with a different authorization ID. The data type of the register is VARCHAR(128). If the length of the authorization ID is less than 8 bytes, the special register value is padded with trailing blanks such that the length is 8 bytes.

See "Example" in the description of the SET SESSION AUTHORIZATION statement.

USER

The USER special register specifies the runtime authorization ID that is used for the current session.

The data type of the register is VARCHAR(128). If the length of the authorization ID is less than 8 bytes, the special register value is padded with trailing blanks such that the length is 8 bytes.

USER is a synonym for the SESSION_USER special register. SESSION_USER is the preferred spelling.

Example: Select all notes from the IN_TRAY table that were placed there by the user.

```
SELECT * FROM IN_TRAY
  WHERE SOURCE = USER
```

Global variables

A global variable is a named memory variable that is retrieved or modified through SQL statements.

Global variables enable applications to share relational data among SQL statements, without the need for additional application logic to support this data transfer.

A global variable is defined within a schema. A global variable defined in a module within a schema is referred to as a *module global variable*. All other global variables are referred to as *schema global variables*.

The definitions for global variables are recorded in the system catalogs.

Types of global variables

There are three different ways to classify global variables: by the ownership of the variable, by the scope of the value, and by the method used to maintain the value.

Ownership of the variable

A global variable can be classified based on whether the variable is owned by the database manager, or if the variable is user-defined:

- The database manager creates *built-in global variables*. Built-in global variables are registered to the database manager in the system catalog. Built-in global variables belong to the following schema:
  - SYSIBM
  - SYSIBMADM

  Some built-in *module global variables* are located inside modules within the SYSIBMADM schema.
- A user creates a *user-defined global variable* by using an SQL DDL statement. User-defined global variables are registered to the database manager in the system catalog. A *user-defined schema global variable* is created by using the CREATE VARIABLE SQL statement. A *user-defined module global variable* is created using the ADD VARIABLE or PUBLISH VARIABLE option of the ALTER MODULE SQL statement.

Scope of the value

A global variable can be classified as either session or database based on the scope of the value:

- The value of a *session global variable* is uniquely associated with each session that uses this particular global variable. Session global variables are either built-in global variables or user-defined global variables.
- The value of a *database global variable* is a single value that remains the same for all sessions that use this particular global variable. Database global variables are always built-in global variables.

Method by which the value is maintained

A global variable can be classified based on how the variable is maintained:

- A *constant global variable* has a fixed value that is instantiated based on evaluation of the CONSTANT clause when the global variable is first referenced in the session or the database, depending on the scope of the global variable. This type of global variable is created by using the CONSTANT clause in the CREATE VARIABLE statement. A value cannot be assigned to the global variable using an SQL statement. Constant global variables are read-only global variables.
- A *maintained-by-system global variable* has a value that is set by the database manager. A value cannot be assigned using an SQL statement. Only built-in global variables can be defined as maintained-by-system global variables, and most built-in global variables are defined as maintained-by-system global variables. Maintained-by-system global variables are read-only global variables.
- A *maintained-by-user global variable* can be assigned a value using an SQL statement; however, this assignment requires WRITE privilege on the global variable. This type of global variable is the default for user-defined global variables that are defined without using a CONSTANT clause. Built-in global variables can also be defined so values can be assigned using an SQL statement.

Authorization required for global variables

To access a global variable, the authorization ID requires certain privileges or DATAACCESS authority.

Schema global variables

To retrieve the value of a schema global variable, the authorization ID of the statement must have one of these authorizations:

- READ privilege on the schema global variable
- DATAACCESS authority

To specify a schema global variable as the target of a value assignment, the authorization ID of the statement must have one of these authorizations:

- WRITE privilege on the schema global variable
- DATAACCESS authority

Module global variables

If a module global variable is published and is then referenced from outside the module that defined it, the authorization ID of the statement must have one of these authorizations:

- EXECUTE privilege on the module in which the global variable is defined
- DATAACCESS authority

References to module global variables from objects that are defined within the same module as the global variable do not require any authorizations to be held by the authorization ID of the statement.

Resolution of global variable references

Global variable reference resolution depends on whether a global variable name is qualified and where the global variable is referenced.

The order of resolving a global variable reference in relation to the names for a column, SQL variable, SQL parameter, or row variable field is described in "References to SQL parameters, SQL variables, and global variables" in SQL Reference Volume 2.

The implicit qualification of an unqualified global variable name that is used as the main object of a CREATE, ALTER, COMMENT, DROP, GRANT, or REVOKE statement is described in “Unqualified user-defined type, function, procedure, specific, global variable and module names” on page 26.

A best practice is to fully qualify the name of the global variable when referencing the global variable in an SQL statement. This prevents a subsequent change in the SQL path from having an impact on the resolution of the global variable.

The resolution of a global variable reference by the database manager in all other contexts depends on whether the global variable name is qualified.

Qualified names

To resolve the name of a qualified global variable, the reference is evaluated according to the following process:

1. If the global variable reference is made from within a module and the qualifier matches the name of the module, the module is searched for a matching module global variable. The following rules are applied:
   - If the qualifier is a single identifier, the schema name of the module is ignored when the qualifier is compared to the module name.
   - If the qualifier is a two-part identifier, it is compared to the schema-qualified module name.

   If the name of a module global variable matches the unqualified global variable name in the reference, resolution is complete. If the qualifier does not match the name of the module or there is no matching module global variable, resolution continues with the next step.
2. The qualifier is now considered to be a schema name. That specified schema is searched for a matching schema global variable.
   - If a schema global variable name matches the unqualified global variable name in the reference, resolution is complete.
   - If the schema does not exist, then an error is returned.
   - If there are no matching schema global variables in the schema, and the qualifier matched the name of the module in the first step, then an error is returned.
   - Otherwise, resolution continues with the next step.
3. The qualifier is now considered to be a module name. The following rules are applied:
   - If the module name is qualified with a schema name, the module is searched for a matching published module global variable.
   - If the module name is not qualified with a schema name, the schema for the module is the first schema in the SQL path that has a matching module name. If the name of a module matches the schema name that is found in the SQL path, that module is searched for a matching published module global variable.
   - If the module is not found through the SQL path, the existence of a module public alias that matches the name of the global variable qualifier is considered. If a module public alias is found, the module that is associated with the module public alias is searched for a matching published module global variable.

If the name of a published module global variable matches the unqualified global variable name in the global variable reference, resolution is complete. If a matching module is not found or there is no matching module global variable in the matching module, an error is returned.

Unqualified names

To resolve the name of an unqualified global variable, the reference is evaluated according to the following process:

1. If an unqualified global variable reference is made from within a module, the module is searched for a matching module global variable. If a module global variable name matches the global variable name in the reference, resolution is complete. If there is no matching module global variable, resolution continues with the next step.
2. The schemas in the SQL path are searched in order from left to right for a matching schema global variable. If a schema global variable name matches the global variable name in the reference, resolution is complete. If no matching global variable is found after completing this step, an error is returned.

Using global variables

Using global variables requires an understanding of the usage restrictions, the rules for assignment to global variables, and the rules for retrieving global variable values.

Usage restrictions

Global variables can be referenced from within any SQL expression, unless the context of the expression requires that the expression be deterministic. The following situations are examples of contexts that require deterministic expressions and therefore preclude the use of global variables:

- In a table check constraint or data type check constraint
- In the definition of a generated expression column
- In a refresh-immediate materialized query table (MQT)

If the data type of the global variable is a cursor type, then the underlying cursor of the global cursor variable can be referenced anywhere that a *cursor-variable-name* can be specified.

If the data type of the global variable is a row type, a field of the global row variable can be referenced anywhere that a global variable with the same type as the field can be referenced. The global variable name that qualifies the field name is resolved in the same way as any other global variable name.

Assignment

The value of a global variable can be changed if both of the following conditions are true:

- The global variable is not a read-only variable.
- The authorization ID of the statement is authorized to write to the global variable.

A value can be assigned to a global variable using any of the following SQL statements:

- A SET variable statement with a global variable as the target variable
- An EXECUTE, FETCH, SELECT INTO, or VALUE INTO statement with a global variable as an assignment target in the INTO clause
- A CALL statement with a global variable as an argument for an OUT or INOUT parameter of the procedure
- A function invocation with a global variable as an argument for an OUT or INOUT parameter of the function (this is supported only for the source expression of a SET variable statement).

Retrieval

The value of a global variable is obtained by referencing the variable from within the SQL context where the value is needed.

The following table shows when the value of a global variable is read, for the indicated reference of that global variable.

Table 18. When the value of a global variable is read, based on the reference context

Context of the global variable reference	**The reference uses the value of the global variable at the beginning of:**
An SQL statement in a compound SQL (inlined) statement	The compound SQL (inlined) statement
An SQL statement in a compound SQL (compiled) statement	The SQL statement within the compound SQL (compiled) statement
An SQL statement, possibly with a function invocation or a trigger activation[1]	The SQL statement
An SQL statement in an invoked inlined SQL function	The SQL statement invoking the inlined SQL function
An SQL statement in an activated inlined trigger	The SQL statement activating the inlined trigger
An SQL statement in an invoked inlined SQL method	The SQL statement invoking the inlined SQL method
An SQL statement in an invoked compiled SQL function	The SQL statement in the compiled SQL function
An SQL statement in an activated compiled trigger	The SQL statement in the compiled trigger
An SQL statement in an invoked external routine	The SQL statement in the external routine

Note: [1] In this table, the SQL statement that might call a function or activate a trigger does not include the compound SQL (inlined) statement or the compound SQL (compiled) statement.

Functions

A *function* is an operation denoted by a function name followed by one or more operands that are enclosed in parentheses.

A function represents a relationship between a set of input values and a set of result values. The input values to a function are called *arguments*. For example, the TIMESTAMP function can be passed arguments of type DATE and TIME, and the result is a TIMESTAMP.

There are several ways to classify functions.

One way is to classify functions as ether built-in or user-defined.

- *Built-in functions* are functions provided with the database manager. Built-in functions include aggregate functions (for example, AVG), operator functions (for example, +), casting functions (for example, DECIMAL), scalar functions (for example, CEILING), and table functions (for example, BASE_TABLE). Built-in functions are generally defined in schemas that begin with 'SYS' (for example, SYSIBM, SYSFUN, and SYSIBMADM) although some are also defined in schemas that begin with 'DB2' (for example DB2MQ).
- *User-defined functions* are functions that are created using an SQL data definition statement and registered to the database manager in the catalog. *User-defined schema functions* are created using the CREATE FUNCTION statement.*User-defined module functions* are created using the ALTER MODULE ADD FUNCTION or ALTER MODULE PUBLISH FUNCTION statements. A set of *user-defined module functions* is provided with the database manager in a set of modules in a schema called SYSIBMADM. A

user-defined function resides in the schema in which it was created or in the module where it was added or published.

User-defined functions extend the capabilities of the database system by adding function definitions (provided by users or third party vendors) that can be applied in the database engine itself. Extending database functions lets the database exploit the same functions in the engine that an application uses, providing more synergy between application and database.

Another way to classify a user-defined function is as an external function, an SQL function, a sourced function, or an interface function:

- An *external function* is defined to the database with a reference to an object code library, and a function within that library that will be executed when the function is invoked. External functions cannot be aggregate functions.
- An *SQL function* is defined to the database using only SQL statements, including at least one RETURN statement. It can return a scalar value, a row, or a table. SQL functions cannot be aggregate functions.
- A *sourced function* is defined to the database with a reference to another built-in or user-defined function that is already known to the database. Sourced functions can be scalar functions or aggregate functions. They are useful for supporting existing functions with user-defined types.
- An *interface function* is defined to the database with reference to several user-defined routines that are already known to the database. Interface functions can be aggregate functions only.

Another way to classify functions is as scalar, aggregate, row, or table function, depending on the input data values and result values. A *scalar function* is a function that returns a single-valued answer each time it is called. For example, the built-in function SUBSTR() is a scalar function. Scalar UDFs can be either external or sourced.

An *aggregate function* is one which conceptually is passed a set of like values (a column) and returns a single-valued answer. An example of an aggregate function is the built-in function AVG(). A column UDF, which is sourced upon one of the built-in aggregate functions, can be defined. This is useful for distinct types. For example, if there is a distinct type SHOESIZE defined with base type INTEGER, a UDF AVG(SHOESIZE), which is sourced on the built-in function AVG(INTEGER), could be defined, and it would be an aggregate function. An aggregate interface function, which is sourced on multiple user-defined routines, can be defined.

A *row function* is a function that returns one row of values. It can be used in a context where a row expression is supported. It can also be used as a transform function, mapping attribute values of a structured type into values in a row. A row function must be defined as an SQL function.

A *table function* is a function that returns a table to the SQL statement which references it. It may only be referenced in the FROM clause of a SELECT statement. Such a function can be used to apply SQL language processing power to data that does not reside in the database, or to convert such data into a table in the database. A table function can read a file, get data from the Web, or access a Lotus Notes® database and return a result table. This information can be joined with other tables in the database. A table function can be defined as an external function or as an SQL function. (A table function cannot be a sourced function.)

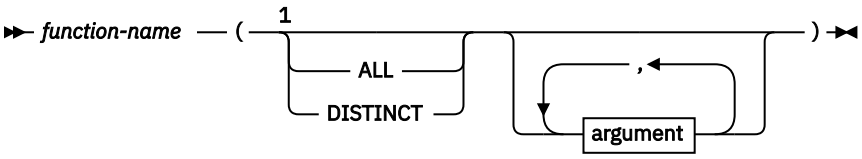
Function signatures

A schema function is identified by its schema name, a function name, the number of parameters, and the data types of its parameters. A module function is identified by its schema name, module name, a function name, the number of parameters, and the data types of its parameters. This identification of a schema function or a module function is called a *function signature*, which must be unique within the database; for example, TEST.RISK(INTEGER). There can be more than one function with the same name in a schema or a module, provided that the number of parameters or the data types of the parameters are different. A function name for which there are multiple function instances with the same number of parameters is called an *overloaded* function. A function name can be overloaded within a schema, in which case there is more than one function by that name with the same number of parameters in the schema. Similarly, a function name can be overloaded within a module, in which case there is more than one function by that name with the same number of parameters in the module. These functions must have different parameter data types. Functions can also be overloaded across the schemas of an SQL

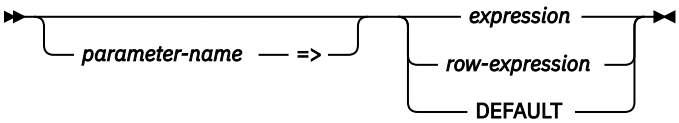
path, in which case there is more than one function by that name with the same number of parameters in different schemas of the SQL path. These functions do not necessarily have different parameter data types.

Function invocation

Each reference to a function conforms to the following syntax:



argument



Notes:

[1] The ALL or DISTINCT keyword can be specified for certain built-in aggregate functions or a user-defined function that is sourced on certain built-in aggregate functions. The ALL keyword can be specified for an aggregate interface function.

In the syntax shown previously, *expression* and *row-expression* cannot include an aggregate function. See "Expressions" for other rules for *expression*.

A function is invoked by referring (in an allowable context) to its qualified or unqualified function name followed by the list of arguments enclosed in parentheses. The possible qualifiers for a function name are:

- A schema name
- An unqualified module name
- A schema-qualified module name

The qualifier used when invoking a function determines the scope used to search for a matching function.

- If a schema-qualified module name is used as the qualifier, the scope is the specified module.
- If a single identifier is used as the qualifier, the scope includes:
  - The schema that matches the qualifier
  - One of the following modules:
    - The invoking module, if the invoking module name matches the qualifier
    - The first module in a schema in the SQL path that matches the qualifier
- If no qualifier is used, the scope includes the schemas in the SQL path and, if the function is invoked from within a module object, the same module from which the function is invoked.

For static SQL statements, the SQL path is specified using the **FUNCPATH** bind option. For dynamic SQL statements, the SQL path is the value of the CURRENT PATH special register.

When any function is invoked, the database manager must determine which function to execute. This process is called *function resolution* and applies to both built-in and user-defined functions. It is recommended that function invocations intending to invoke a user-defined function be fully qualified. This improves performance of function resolution and prevents unexpected function resolution results as new functions are added or privileges granted.

An *argument* is a value passed to a function upon invocation or the specification of DEFAULT. When a function is invoked in SQL, it is passed a list of zero or more arguments. They are positional in that the semantics of an argument are determined by its position in the argument list. A *parameter* is a formal definition of an input to a function or an output from a function. When a function is defined to

the database, either internally (a built-in function) or by a user (a user-defined function), its parameters (zero or more) are specified, and the order of their definitions defines their positions and their semantics. Therefore, every parameter is a particular positional input to a function or an output from a function. On invocation, an argument is assigned to a parameter using either the positional syntax or the named syntax. If using the positional syntax, an argument corresponds to a particular parameter according to its position in the list of arguments. If using the named syntax, an argument corresponds to a particular parameter by the name of the parameter. When an argument is assigned to a parameter using the named syntax, then all the arguments that follow it must also be assigned using the named syntax (SQLSTATE 4274K). The name of a named argument can appear only once in a function invocation (SQLSTATE 4274K). In cases where the data types of the arguments of the function invocation are not a match to the data types of the parameters of the selected function, the arguments are converted to the data type of the parameter at execution time using the same rules as assignment to columns. This includes the case where precision, scale, or length differs between the argument and the parameter. In cases where the arguments of the function invocation are the specification of DEFAULT, the actual value used for the argument is the value specified as the default for the corresponding parameter in the function definition. If no default value was defined for the parameter, the null value is used. If an untyped expression (a parameter marker, a NULL keyword, or a DEFAULT keyword) is used as the argument, the data type associated with the argument is determined by the parameter data type of the parameter of the selected function.

Access to schema functions is controlled through the EXECUTE privilege on the schema functions. If the authorization ID of the statement invoking the function does not have EXECUTE privilege, the schema function will not be considered by the function resolution algorithm, even if it is a better match. Built-in functions in the SYSIBM and SYSFUN schemas have the EXECUTE privilege implicitly granted to PUBLIC.

Access to module functions is controlled through EXECUTE privilege on the module for all functions within the module. The authorization ID of the statement invoking the function might not have EXECUTE privilege on a module. In such cases, module functions within that module, unlike schema functions, are still considered by the function resolution algorithm even though they cannot be executed.

When the user-defined function is invoked, the value of each of its arguments is assigned, using storage assignment, to the corresponding parameter of the function. Control is passed to external functions according to the calling conventions of the host language. When execution of a user-defined scalar function or a user-defined aggregate function is complete, the result of the function is assigned, using storage assignment, to the result data type. For details on the assignment rules, see "Assignments and comparisons".

Table functions can be referenced only in the FROM clause of a subselect. For more information on referencing a table function, see “table-reference” on page 644.

Function resolution

After a function is invoked, the database manager must determine which function to execute. This process is called function resolution and applies for both built-in and user-defined functions.

The database manager first determines the set of candidate functions based on the following information:

- The qualification of the name of the invoked function
- The context that invokes the function
- The unqualified name of the invoked function
- The number of arguments specified
- Any argument names that are specified
- The authorization of schema functions.

See “Determining the set of candidate functions” on page 116 for details.

The database manager then determines the best fit from the set of candidate functions based on the data types of the arguments of the invoked function as compared with the data types of the parameters of the functions in the set of candidate functions. The SQL path and number of parameters is also considered. See “Determining the best fit” on page 117 for details.

Once a function is selected, it is still possible for an error to be returned for one of the following reasons:

- If a module function is selected and either the function is invoked from outside a module or the function is invoked from within a module object and the qualifier does not match the context module name, the authorization ID of the statement that invoked the function must have EXECUTE privilege on the module that contains the selected function (SQLSTATE 42501).
- If a function is selected, its successful use depends on it being invoked in a context in which the returned result is allowed. For example, if the function returns a table where a table is not allowed, an error is returned (SQLSTATE 42887).
- If a cast function is selected, either built-in or user-defined, and any argument would need to be implicitly cast (not promoted) to the data type of the parameter, an error is returned (SQLSTATE 42884).
- If a function invocation involves an argument with an unnamed row type, an error is returned (SQLSTATE 42884) if either of the following conditions occur:
  - The number of fields of the argument does not match the number of fields of the parameter.
  - The data types of the fields of the argument are not assignable to the corresponding data type of the fields of the parameter.

Determining the set of candidate functions

- Let *A* be the number of arguments in a function invocation.
- Let *P* be the number of parameters in a function signature.
- Let *N* be the number of parameters in a function signature without a defined default.

Candidate functions for resolution of a function invocation are selected based on the following criteria:

- Each candidate function has a matching name and applicable number of parameters. An applicable number of parameters satisfies the condition $N \leq A \leq P$.
- Each candidate function has parameters such that for each named argument in the function invocation there exists a parameter with a matching name that does not already correspond to a positional (unnamed) argument.
- Each parameter of a candidate function that does not have a corresponding argument in the function invocation, specified by either position or name, is defined with a default.
- Each candidate function from a set of one or more schemas has the EXECUTE privilege associated with the authorization ID of the statement invoking the function.
- Each candidate function from a module other than the context module is a published module function.

The functions selected for the set of candidate functions are from one or more of the following search spaces.

1. The *context module*, that is, the module which contains the module object that invoked the function
2. A set of one or more schemas
3. A module other than the context module

The specific search spaces considered are affected by the qualification of the name of the invoked function.

- *Qualified function invocation*: When a function is invoked with a function name and a qualifier, the database manager uses the qualifier and, in some cases, the context of the invoked function to determine the set of candidate functions.
  1. If a function is invoked from within a module object using a function name with a qualifier, the database manager considers if the qualifier matches the context module name. If the qualifier is a single identifier, then the schema name of the module is ignored when determining a match. If the qualifier is a two-part identifier, then it is compared to the schema-qualified module name when determining a match. If the qualifier matches the context module name, the database manager searches the context module for candidate functions.

If one or more candidate functions are found in the context module, then this set of candidate functions is processed for best fit without consideration of possible candidate functions in any other search space (see "Determining the best fit"). Otherwise, continue to the next search space.

2. If the qualifier is a single identifier, the database manager considers the qualifier as a schema name and searches that schema for candidate functions.

   If one or more candidate functions are found in the schema, then this set of candidate functions is processed for best fit without consideration of possible candidate functions in any other search space (see "Determining the best fit"). Otherwise, continue to the next search space, if applicable.

3. If the function is invoked from outside a module or the qualifier does not match the context module name when it is invoked from within a module object, the database manager considers the qualifier as a module name. Without considering EXECUTE privilege on modules, the database manager then selects the first module that matches based on the following criteria:
   - If the module name is qualified with a schema name, select the module with that schema name and module name.
   - If the module name is not qualified with a schema name, select the module with that module name that is found in the schema earliest in the SQL path.
   - If the module is not found using the SQL path, select the module public alias with that module name.

   If a matching module is not found, then there are no candidate functions. If a matching module is found, the database manager searches the selected module for candidate functions.

   If one or more candidate functions are found in the selected modules, then this set of candidate functions is processed for best fit (see "Determining the best fit").

- *Unqualified function invocation*: When a function is invoked without a qualifier, the database manager considers the context of the invoked function to determine the sets of candidate functions.

  1. If a function is invoked with an unqualified function name from within a module object, the database manager searches the context module for candidate functions.

     If one or more candidate functions are found in the context module, then these candidate functions are included with any candidate functions from the schemas in the SQL path (see next item).

  2. If a function is invoked with an unqualified function name, either from within a module object or from outside a module, the database manager searches the list of schemas in the SQL path to resolve the function instance to execute. For each schema in the SQL path (see "SQL path"), the database manager searches the schema for candidate functions.

     If one or more candidate functions are found in the schemas in the SQL path, then these candidate functions are included with any candidate functions from the context module (see previous item). This set of candidate functions is processed for best fit (see "Determining the best fit").

If the database manager does not find any candidate functions, an error is returned (SQLSTATE 42884).

Determining the best fit

The set of candidate functions may contain one function or more than one function with the same name. In either case, the data types of the parameters, the position of the schema in the SQL path, and the total number of parameters of each function in the set of candidate functions are used to determine if the function meets the best fit requirements.

If the set of candidate functions contains more than one function and named arguments are used in the function invocation, the ordinal position of the parameter corresponding to a named argument must be the same for all candidate functions (SQLSTATE 4274K).

The term *set of parameters* is used to refer to all of the parameters at the same position in the parameter lists (where such a parameter exists) for the set of candidate functions. The corresponding argument of a parameter is determined based on how the arguments are specified in the function invocation. For positional arguments, the corresponding argument to a parameter is the argument in the same position in the function invocation as the position of the parameter in the parameter list of the candidate function.

For named arguments, the corresponding argument to a parameter is the argument with the same name as the parameter. In this case, the order of the arguments in the function invocation is not considered while determining the best fit. If the number of parameters in a candidate function is greater than the number of arguments in the function invocation, each parameter that does not have a corresponding argument is processed as if it does have a corresponding argument that has the DEFAULT keyword as the value.

The following steps are used to determine the function that is the best fit:

Step 1: Considering arguments that are typed expressions

The database manager determines the function, or set of functions, that meet the best fit requirements for the invocation by comparing the data type of each parameter with the data type of the corresponding argument.

When determining whether the data type of a parameter is the same as the data type of its corresponding argument:

- Synonyms of data types match. For example, FLOAT and DOUBLE are considered to be the same.
- Attributes of a data type such as length, precision, scale, and code page are ignored. Therefore, CHAR(8) and CHAR(35) are considered to be the same, as are DECIMAL(11,2), and DECIMAL(4,3).

A subset of the candidate functions is obtained by considering only those functions for which the data type of each argument of the function invocation that is not an untyped expression matches or is promotable to the data type of the corresponding parameter of the function instance. If the argument of the function invocation is an untyped expression, the data type of the corresponding parameter can be any data type. The precedence list for the promotion of data types in "Promotion of data types" on page 46 shows the data types that fit (considering promotion) for each data type in best-to-worst order. If this subset is not empty, then the best fit is determined using the Promotable process on this subset of candidate functions. If this subset is empty, then the best fit is determined using the Castable process on the original set of candidate functions.

Promotable process

This process determines the best fit only considering whether arguments in the function invocation match or can be promoted to the data type of the corresponding parameter of the function definition. For the subset of candidate functions, the parameter lists are processed from left to right, processing the set of parameters in the first position from the subset of candidate functions before moving on to the set of parameters in the second position, and so on. The following steps are used to eliminate candidate functions from the subset of candidate functions (only considering promotion):

1. If one candidate function has a parameter where the data type of the corresponding argument fits (only considering promotion) the data type of the parameter better than other candidate functions, those candidate functions that do not fit the function invocation equally well are eliminated. The precedence list for the promotion of data types in "Promotion of data types" on page 46 shows the data types that fit (considering promotion) for each data type in best-to-worst order.
2. If the data type of the corresponding argument is an untyped expression, no candidate functions are eliminated.
3. These steps are repeated for the next set of parameters from the remaining candidate functions until there are no more sets of parameters.

Castable process

This process determines the best fit first considering, for each parameter, if the data type of the corresponding argument in the function invocation matches or can be promoted to the data type of the parameter of the function definition. Then, for each set of parameters where no corresponding argument has a data type that was promotable, the database manager considers, for each parameter, if the data type of the corresponding argument can be implicitly cast for function resolution to the data type of the parameter.

For the set of candidate functions, the parameters in the parameter lists are processed from left to right, processing the set of parameters in the first position from all the candidate functions before

moving on to the set of parameters in the second position, and so on. The following steps are used to eliminate candidate functions from the set of candidate functions (only considering promotion):

1. If one candidate function has a parameter where the data type of the corresponding argument fits (only considering promotion) the data type of the parameter better than other candidate functions, those candidate functions that do not fit the function invocation equally well are eliminated. The precedence list for the promotion of data types in "Promotion of data types" on page 46 shows the data types that fit (considering promotion) for each data type in best-to-worst order.
2. If the data type for the corresponding argument is not promotable (which includes the case when the corresponding argument is an untyped expression) to the data type of the parameter of any candidate function, no candidate functions are eliminated.
3. These steps are repeated for the next set of parameters from the remaining candidate functions until there are no more sets of parameters.

If at least one set of parameters has no corresponding argument that fit (only considering promotion) and the corresponding argument for the set of parameters has a data type, the database manager compares each such set of parameters from left to right. The following steps are used to eliminate candidate functions from the set of candidate functions (considering implicit casting).

1. If all the data types of the set of parameters for all remaining candidate functions do not belong to the same data type precedence list, as specified in "Promotion of data types" on page 46, an error is returned (SQLSTATE 428F5).
2. If the data type of the corresponding arguments cannot be implicitly cast to the data type of the parameters, as specified in Implicit casting for function resolution, an error is returned (SQLSTATE 42884).
3. If one candidate function has a parameter where the data type of the corresponding argument fits (considering implicit casting) the data type of the parameter better than other candidate functions, those candidate functions that do not fit the function invocation equally well are eliminated. The data type list in Implicit casting for function resolution shows the data type that fits (considering implicit casting) better.
4. These steps are repeated for the next set of parameters which has no corresponding argument that fit (only considering promotion) and the corresponding argument for the set of parameters has a data type until there are no more such sets of parameters or an error occurs.

Step 2: Considering SQL path

If more than one candidate function remains and a context module exists that still includes candidate functions, the database manager selects those functions. If there is no context module or no candidate functions remain in the context module, the database manager selects those candidate functions whose schema is earliest in the SQL path.

Step 3: Considering number of arguments in the function invocation

If more than one candidate function remains and if one candidate function has a number of parameters that is less than or equal to the number of parameters of the other candidate functions, those candidate functions that have a greater number of parameters are eliminated.

Step 4: Considering arguments that are untyped expressions

If more than one candidate function remains and at least one set of parameters has a corresponding argument that is an untyped expression, the database manager compares each such set of parameters from left to right. The following steps are used to eliminate candidate functions from the set of candidate functions:

1. If all the data types of the set of parameters for all remaining candidate functions do not belong to the same data type precedence list, as specified in "Promotion of data types" on page 46, an error is returned (SQLSTATE 428F5).
2. If the data type of the parameter of one candidate function is further left in the data type ordering for implicit casting than other candidate functions, those candidate functions where the data type

of the parameter is further right in the data type ordering are eliminated. The data type list in "Implicit casting for function resolution" shows the data type ordering for implicit casting.

If there are still multiple candidate functions, an error is returned (SQLSTATE 428F5).

Implicit casting for function resolution

Implicit casting for function resolution is not supported for arguments with a user-defined type, reference type, or XML data type. It is also not supported for built-in or user-defined cast functions. It is supported for the following cases:

- A value of one data type can be cast to any other data type that is in the same data type precedence list, as specified in “Promotion of data types” on page 46.
- A numeric or datetime data type can be cast to:
  - In a Unicode database, a character data type other than CLOB or a graphic data type other than DBCLOB.
  - In a non-Unicode database, a character data type other than CLOB.
- A character data type other than CLOB can be cast to a numeric or datetime data type.
- In a Unicode database, a graphic data type other than DBCLOB can be cast to a numeric or datetime data type.
- A character FOR BIT DATA can be cast to a binary string data type.
- A binary string data type can be cast to a character FOR BIT DATA.
- A TIMESTAMP data type can be cast to a TIME data type.
- A BOOLEAN data type can be cast to a binary integer data type, a character data type other than CLOB, or a graphic data type other than DBCLOB.
- A binary integer data type, a character data type other than CLOB, or a graphic data type other than DBCLOB can be cast to BOOLEAN.
- An untyped argument can be cast to any data type.

Similar to the data type precedence list for promotion, for implicit casting there is an order to the data types that are in the group of related data types. This order is used when performing function resolution that considers implicit casting. Table 19 on page 120 shows the data type ordering for implicit casting for function resolution. The data types are listed in best-to-worst order (note that this is different than the ordering in the data type precedence list for promotion). In a Unicode database, when function resolution selects a built-in function from the SYSIBM schema and implicit casting is necessary for some argument, if the built-in function supports both character input and graphic input for the parameter, the argument is implicitly cast to character.

Table 19. Data type ordering for implicit casting for function resolution

Data type group	**Data type list for implicit casting for function resolution (in best-to-worst order)**
Numeric data types	DECFLOAT, double, real, decimal, BIGINT, INTEGER, SMALLINT
Character and graphic string data types	VARCHAR or VARGRAPHIC, CHAR or GRAPHIC, CLOB or DBCLOB
Binary string data types	VARBINARY, BINARY, BLOB
Datetime data types	TIMESTAMP, DATE

Notes:

1. The lowercase types in the previous table are defined as follows:
   - decimal = DECIMAL (*p*,*s*) or NUMERIC(*p*,*s*)
   - real = REAL or FLOAT(*n*) where *n* is not greater than 24

- double = DOUBLE, DOUBLE-PRECISION, FLOAT or FLOAT(*n*), where *n* is greater than 24

Shorter and longer form synonyms of the listed data types are considered to be the same as the listed form.

2. For a Unicode database only, the following are considered to be equivalent data types:
   - CHAR or GRAPHIC
   - VARCHAR and VARGRAPHIC
   - CLOB and DBCLOB

Table 20. Derived length of an argument when invoking a built-in scalar function from the SYSIBM schema in cases where implicit casting is needed

Source Data Type	Target Type and Length											
	Char	**Graphic**	**Varchar**	**Vargraphic**	**Clob**	**DBclob**	**Binary**	**Varbinary**	**Blob**	**Timestamp**	**Decfloat**	**Boolean**
UNTYPED	127	127	254	254	32767	32767	-	-	32767	12	34	5
SMALLINT	6	6	6	6	-	-	-	-	-	-	-	5
INTEGER	11	11	11	11	-	-	-	-	-	-	-	5
BIGINT	20	20	20	20	-	-	-	-	-	-	-	5
DECIMAL(*p*,*s*)	2+*p*	2+*p*	2+*p*	2+*p*	-	-	-	-	-	-	-	-
REAL	24	24	24	24	-	-	-	-	-	-	-	-
DOUBLE	24	24	24	24	-	-	-	-	-	-	-	-
DECFLOAT	42	42	42	42	-	-	-	-	-	-	-	-
CHAR(*n*)	-	-	-	-	-	-	*n*	*n*	*n*	12	34	-
VAR CHAR(*n*)	min(*n*, 254)	min(*n*, 127)	-	-	-	-	min(*n*, 254)	*n*	*n*	12	34	-
CLOB(*n*)	min(*n*, 254)	min(*n*, 127)	min(*n*, 32672)	min(*n*, 16336)	-	-	-	-	-	-	-	-
GRAPHIC(*n*)	-	-	-	-	-	-	-	-	-	12	34	-
VARGRAPHIC(*n*)	min(*n*, 254)	min(*n*, 127)	-	-	-	-	-	-	-	12	34	-
DBCLOB(*n*)	min(*n*, 254)	min(*n*, 127)	min(*n*, 32672)	min(*n*, 16336)	-	-	-	-	-	-	-	-
BINARY(*n*)	*n*	-	*n*	-	-	-	-	-	-	-	-	-
VARBINARY(*n*)	min(*n*, 254)	-	*n*	-	-	-	min(*n*, 254)	-	-	-	-	-
BLOB(*n*)	min(*n*, 254)	-	min(*n*, 32672)	-	-	-	min(*n*, 254)	min(*n*, 32672)	-	-	-	-
TIME	8	8	8	8	-	-	-	-	-	-	-	-
DATE	10	10	10	10	-	-	-	-	-	-	-	-
TIMESTAMP(*p*)	if *p*=0 then 19 else *p*+20	if *p*=0 then 19 else *p*+20	if *p*=0 then 19 else *p*+20	if *p*=0 then 19 else *p*+20	-	-	-	-	-	-	-	-
BOOLEAN	5	5	5	5	-	-	-	-	-	-	-	-

Note
This table shows character string and graphic string data types in string units associated with a Unicode database environment where the string units default is SYSTEM. If the Unicode database environment has the string units set to CODEUNITS32, then any character string or graphic string length attributes that represent the data type maximum length should be considered to represent the data type maximum in CODEUNITS32. All character string or graphic string data types have the default string units of the database environment.

SQL path considerations for built-in functions

Function resolution applies to all functions, including schema functions and modules functions that are built-in or user-defined. If a function is invoked without its schema name, the SQL path is used to resolve the function invocation to a specific function.

The built-in functions in the SYSIBM schema are always considered during function resolution, even when SYSIBM is not explicitly included in the SQL path. Omission of SYSIBM from the path results in the assumption (for function and data type resolution) that SYSIBM is the first schema on the path.

For example, if a user's SQL path is defined as:

```
"SHAREFUN","SYSIBM","SYSFUN"
```

and there is a LENGTH function defined in schema SHAREFUN with the same number and types of arguments as SYSIBM.LENGTH, then an unqualified reference to LENGTH in this user's SQL statement will result in selecting SHAREFUN.LENGTH. However, if the user's SQL path is defined as:

```
"SHAREFUN","SYSFUN"
```

and the same SHAREFUN.LENGTH function exists, then an unqualified reference to LENGTH in this user's SQL statement will result in selecting SYSIBM.LENGTH, because SYSIBM implicitly appears first in the path.

To minimize potential problems in this area:

- Never use the names of built-in functions for user-defined functions.
- If, for some reason, it is necessary to create a user-defined function with the same name as a built-in function, be sure to qualify any references to it.

Note: Some invocations of built-in functions do not support SYSIBM as an explicit qualifier and resolve directly to the built-in function without considering the SQL path. Specific cases are covered in the description of the built-in function.

Examples of function resolution

The following are examples of function resolution. (Note that not all required keywords are shown.)

- This is an example illustrating the SQL path considerations in function resolution. For this example, there are eight ACT functions, in three different schemas, registered as:

```
CREATE FUNCTION AUGUSTUS.ACT (CHAR(5), INT, DOUBLE) SPECIFIC ACT_1 ...
CREATE FUNCTION AUGUSTUS.ACT (INT, INT, DOUBLE) SPECIFIC ACT_2 ...
CREATE FUNCTION AUGUSTUS.ACT (INT, INT, DOUBLE, INT) SPECIFIC ACT_3 ...
CREATE FUNCTION JULIUS.ACT (INT, DOUBLE, DOUBLE) SPECIFIC ACT_4 ...
CREATE FUNCTION JULIUS.ACT (INT, INT, DOUBLE) SPECIFIC ACT_5 ...
CREATE FUNCTION JULIUS.ACT (SMALLINT, INT, DOUBLE) SPECIFIC ACT_6 ...
CREATE FUNCTION JULIUS.ACT (INT, INT, DECFLOAT) SPECIFIC ACT_7 ...
CREATE FUNCTION NERO.ACT (INT, INT, DEC(7,2)) SPECIFIC ACT_8 ...
```

  The function reference is as follows (where I1 and I2 are INTEGER columns, and D is a DECIMAL column):

```
SELECT ... ACT(I1, I2, D) ...
```

  Assume that the application making this reference has an SQL path established as:

```
"JULIUS","AUGUSTUS","CAESAR"
```

  Following through the algorithm...

  – The function with specific name ACT_8 is eliminated as a candidate, because the schema NERO is not included in the SQL path.
  – The function with specific name ACT_3 is eliminated as a candidate, because it has the wrong number of parameters. ACT_1 and ACT_6 are eliminated because, in both cases, the first argument cannot be promoted to the data type of the first parameter.
  – Because there is more than one candidate remaining, the arguments are considered in order.
  – For the first argument, the remaining functions, ACT_2, ACT_4, ACT_5, and ACT_7 are an exact match with the argument type. No functions can be eliminated from consideration; therefore the next argument must be examined.
  – For this second argument, ACT_2, ACT_5, and ACT_7 are exact matches, but ACT_4 is not, so it is eliminated from consideration. The next argument is examined to determine some differentiation among ACT_2, ACT_5, and ACT_7.

- For the third and last argument, neither ACT_2, ACT_5, nor ACT_7 match the argument type exactly. Although ACT_2 and ACT_5 are equally good, ACT_7 is not as good as the other two because the type DOUBLE is closer to DECIMAL than is DECFLOAT. ACT_7 is eliminated..
- There are two functions remaining, ACT_2 and ACT_5, with identical parameter signatures. The final tie-breaker is to see which function's schema comes first in the SQL path, and on this basis, ACT_5 is the function chosen.

- This is an example of a situation where function resolution will result in an error (SQLSTATE 428F5) since more than one candidate function fits the invocation equally well, but the corresponding parameters for one of the arguments do not belong to the same type precedence list.

  For this example, there are only three function in a single schema defined as follows:

  ```
  CREATE FUNCTION CAESAR.ACT (INT, VARCHAR(5), VARCHAR(5))SPECIFIC ACT_1 ...
  CREATE FUNCTION CAESAR.ACT (INT, INT, DATE)     SPECIFIC ACT_2 ...
  CREATE FUNCTION CAESAR.ACT (INT, INT, DOUBLE)   SPECIFIC ACT_3 ...
  ```

  The function reference is as follows (where I1 and I2 are INTEGER columns, and VC is a VARCHAR column):

  ```
  SELECT ... ACT(I1, I2, VC) ...
  ```

  Assume that the application making this reference has an SQL path established as:

  ```
  "CAESAR"
  ```

  Following through the algorithm ...

  - Each of the candidate functions is evaluated to determine if the data type of each input argument of the function invocation matches or is promotable to the data type of the corresponding parameter of the function instance:
    - For the first argument, all the candidate functions have an exact match with the parameter type.
    - For the second argument, ACT_1 is eliminated because INTEGER is not promotable to VARCHAR.
    - For the third argument, both ACT_2 and ACT_3 are eliminated since VARCHAR is not promotable to DATE or DOUBLE, so no candidate functions remain.
  - Since the subset of candidate functions is empty, the candidate functions are considered using the castable process:
    - For the first argument, all the candidate functions have an exact match with the parameter type.
    - For the second argument, ACT_1 is eliminated since INTEGER is not promotable to VARCHAR. ACT_2 and ACT_3 are better candidates.
    - For the third argument, the data type of the corresponding parameters of ACT_2 and ACT_3 do not belong to the same data type precedence list, so an error is returned (SQLSTATE 428F5).

- This example illustrates a situation where function resolution will succeed using the castable process. For this example, there are only three function in a single schema defined as follows:

  ```
  CREATE FUNCTION CAESAR.ACT (INT, VARCHAR(5), VARCHAR(5))SPECIFIC ACT_1 ...
  CREATE FUNCTION CAESAR.ACT (INT, INT, DECFLOAT)     SPECIFIC ACT_2 ...
  CREATE FUNCTION CAESAR.ACT (INT, INT, DOUBLE)   SPECIFIC ACT_3 ...
  ```

  The function reference is as follows (where I1 and I2 are INTEGER columns, and VC is a VARCHAR column):

  ```
  SELECT ... ACT(I1, I2, VC) ...
  ```

  Assume that the application making this reference has an SQL path established as:

  ```
  "CAESAR"
  ```

  Following through the algorithm ...

– Each of the candidate functions is evaluated to determine if the data type of each input argument of the function invocation matches or is promotable to the data type of the corresponding parameter of the function instance:
  - For the first argument, all the candidate functions have an exact match with the parameter type.
  - For the second argument, ACT_1 is eliminated because INTEGER is not promotable to VARCHAR.
  - For the third argument, both ACT_2 and ACT_3 are eliminated since VARCHAR is not promotable to DECFLOAT or DOUBLE, so no candidate functions remain.
– Since the subset of candidate functions is empty, the candidate functions are considered using the castable process:
  - For the first argument, all the candidate functions have an exact match with the parameter type.
  - For the second argument, ACT_1 is eliminated since INTEGER is not promotable to VARCHAR. ACT_2 and ACT_3 are better candidates.
  - For the third argument, both DECFLOAT and DOUBLE are in the same data type precedence list and VARCHAR can be implicitly cast to both DECFLOAT and DOUBLE. Since DECFLOAT is a better fit for the purpose of implicit casting, ACT_2 is the best fit

- This example illustrates that during function resolution using the castable process that promotion of later arguments takes precedence over implicit casting. For this example, there are only three function in a single schema defined as follows:

```
CREATE FUNCTION CAESAR.ACT (INT, INT, VARCHAR(5))SPECIFIC ACT_1 ...
CREATE FUNCTION CAESAR.ACT (INT, INT, DECFLOAT) SPECIFIC ACT_2 ...
CREATE FUNCTION CAESAR.ACT (INT, INT, DOUBLE)  SPECIFIC ACT_3 ...
```

  The function reference is as follows (where I1 is an INTEGER column, and VC1 is a VARCHAR column and C1 is a CHAR column):

```
SELECT ... ACT(I1, VC1, C1) ...
```

  Assume that the application making this reference has an SQL path established as:

```
"CAESAR"
```

  Following through the algorithm:

  – Each of the candidate functions is evaluated to determine if the data type of each input argument of the function invocation matches or is promotable to the data type of the corresponding parameter of the function instance:
    - For the first argument, all the candidate functions have an exact match with the parameter type.
    - For the second argument, all candidate functions are eliminated since VARCHAR is not promotable to INTEGER, so no candidate functions remain.
  – Since the subset of candidate functions is empty, the candidate functions are considered using the castable process
    - For the first argument, all the candidate functions have an exact match with the parameter type.
    - For the second argument, none of the candidate functions have a parameter to which the corresponding argument can be promoted, so no candidate functions are eliminated.
    - Since the third argument can be promoted to the parameter of ACT_1, but not to the parameters of ACT_2 or ACT_3, ACT_1 is the best fit.

Methods

A database method of a structured type is a relationship between a set of input data values and a set of result values, where the first input value (or *subject argument*) has the same type, or is a subtype of the subject type (also called *subject parameter*), of the method.

For example, a method called CITY, of type ADDRESS, can be passed input data values of type VARCHAR, and the result is an ADDRESS (or a subtype of ADDRESS).

Methods are defined implicitly or explicitly, as part of the definition of a user-defined structured type.

Implicitly defined methods are created for every structured type. *Observer methods* are defined for each attribute of the structured type. Observer methods allow applications to get the value of an attribute for an instance of the type. *Mutator methods* are also defined for each attribute, allowing applications to mutate the type instance by changing the value for an attribute of a type instance. The CITY method described previously is an example of a mutator method for the type ADDRESS.

Explicitly defined methods, or *user-defined methods*, are methods that are registered to a database in SYSCAT.ROUTINES, by using a combination of CREATE TYPE (or ALTER TYPE ADD METHOD) and CREATE METHOD statements. All methods defined for a structured type are defined in the same schema as the type.

User-defined methods for structured types extend the function of the database system by adding method definitions (provided by users or third party vendors) that can be applied to structured type instances in the database engine. Defining database methods lets the database exploit the same methods in the engine that an application uses, providing more synergy between application and database.

External and SQL user-defined methods

A user-defined method can be either external or based on an SQL expression. An external method is defined to the database with a reference to an object code library and a function within that library that will be executed when the method is invoked. A method based on an SQL expression returns the result of the SQL expression when the method is invoked. Such methods do not require any object code library, because they are written completely in SQL.

A user-defined method can return a single-valued answer each time it is called. This value can be a structured type. A method can be defined as *type preserving* (using SELF AS RESULT), to allow the dynamic type of the subject argument to be returned as the returned type of the method. All implicitly defined mutator methods are type preserving.

Method signatures

A method is identified by its subject type, a method name, the number of parameters, and the data types of its parameters. This is called a *method signature*, and it must be unique within the database.

There can be more than one method with the same name for a structured type, provided that:

- The number of parameters or the data types of the parameters are different, or
- The methods are part of the same method hierarchy (that is, the methods are in an overriding relationship or override the same original method), or
- The same function signature (using the subject type or any of its subtypes or supertypes as the first parameter) does not exist.

A method name that has multiple method instances is called an *overloaded method*. A method name can be overloaded within a type, in which case there is more than one method by that name for the type (all of which have different parameter types). A method name can also be overloaded in the subject type hierarchy, in which case there is more than one method by that name in the type hierarchy. These methods must have different parameter types.

A method can be invoked by referring (in an allowable context) to the method name, preceded by both a reference to a structured type instance (the subject argument), and the double dot operator. A list of arguments enclosed in parentheses must follow. Which method is actually invoked depends on the static

type of the subject type, using the method resolution process described in the following section. Methods defined WITH FUNCTION ACCESS can also be invoked using function invocation, in which case the regular rules for function resolution apply.

If function resolution results in a method defined WITH FUNCTION ACCESS, all subsequent steps of method invocation are processed.

Access to methods is controlled through the EXECUTE privilege. GRANT and REVOKE statements are used to specify who can or cannot execute a specific method or a set of methods. The EXECUTE privilege (or DATAACCESS authority) is needed to invoke a method. The definer of the method automatically receives the EXECUTE privilege. The definer of an external method or an SQL method having the WITH GRANT option on all underlying objects also receives the WITH GRANT option with the EXECUTE privilege on the method. The definer (or authorization ID with the ACCESSCTRL or SECADM authority) must then grant it to the user who wants to invoke the method from any SQL statement, or reference the method in any DDL statement (such as CREATE VIEW, CREATE TRIGGER, or when defining a constraint). If the EXECUTE privilege is not granted to a user, the method will not be considered by the method resolution algorithm, even if it is a better match.

Method resolution

After method invocation, the database manager must decide which of the possible methods with the same name is the "best fit". Functions (built-in or user-defined) are not considered during method resolution.

An *argument* is a value passed to a method upon invocation. When a method is invoked in SQL, it is passed the subject argument (of some structured type) and a list of zero or more arguments. They are positional in that the semantics of an argument are determined by its position in the argument list. A *parameter* is a formal definition of an input to a method. When a method is defined to the database, either implicitly (system-generated for a type) or by a user (a user-defined method), its parameters are specified (with the subject parameter as the first parameter), and the order of their definitions defines their positions and their semantics. Therefore, every parameter is a particular positional input to a method. On invocation, an argument corresponds to a particular parameter by virtue of its position in the list of arguments.

The database manager uses the name of the method given in the invocation, EXECUTE privilege on the method, the number and data types of the arguments, all the methods with the same name for the subject argument's static type (and it's supertypes), and the data types of their corresponding parameters as the basis for deciding whether or not to select a method. The following are the possible outcomes of the decision process:

- A particular method is deemed to be the best fit. For example, given the methods named RISK for the type SITE with signatures defined as:

  ```
  PROXIMITY(INTEGER) FOR SITE
  PROXIMITY(DOUBLE) FOR SITE
  ```

  the following method invocation (where ST is a SITE column, DB is a DOUBLE column):

  ```
  SELECT ST..PROXIMITY(DB) ...
  ```

  then, the second PROXIMITY will be chosen.

  The following method invocation (where SI is a SMALLINT column):

  ```
  SELECT ST..PROXIMITY(SI) ...
  ```

  would choose the first PROXIMITY, because SMALLINT can be promoted to INTEGER and is a better match than DOUBLE, which is further down the precedence list.

  When considering arguments that are structured types, the precedence list includes the supertypes of the static type of the argument. The best fit is the function defined with the supertype parameter that is closest in the structured type hierarchy to the static type of the function argument.

- No method is deemed to be an acceptable fit. For example, given the same two functions in the previous case and the following function reference (where C is a CHAR(5) column):

```
SELECT ST..PROXIMITY(C) ...
```

  the argument is inconsistent with the parameter of both PROXIMITY functions.
- A particular method is selected based on the methods in the type hierarchy and the number and data types of the arguments passed on invocation. For example, given methods named RISK for the types SITE and DRILLSITE (a subtype of SITE) with signatures defined as:

```
RISK(INTEGER) FOR DRILLSITE
RISK(DOUBLE) FOR SITE
```

  and the following method invocation (where DRST is a DRILLSITE column, DB is a DOUBLE column):

```
SELECT DRST..RISK(DB) ...
```

  the second RISK will be chosen, because DRILLSITE can be promoted to SITE.

  The following method reference (where SI is a SMALLINT column):

```
SELECT DRST..RISK(SI) ...
```

  would choose the first RISK, because SMALLINT can be promoted to INTEGER, which is closer on the precedence list than DOUBLE, and DRILLSITE is a better match than SITE, which is a supertype.

  Methods within the same type hierarchy cannot have the same signatures, considering parameters other than the subject parameter.

Determining the best fit

A comparison of the data types of the arguments with the defined data types of the parameters of the methods under consideration forms the basis for the decision of which method in a group of like-named methods is the "best fit". Note that the data types of the results of the methods under consideration do not enter into this determination.

For method resolution, whether the data type of the input arguments can be promoted to the data type of the corresponding parameter is considered when determining the best fit. Unlike function resolution, whether the input arguments can be implicitly cast to the data type of the corresponding parameter is not considered when determining the best fit. Modules are not considered during method resolution because methods cannot be defined in modules.

Method resolution is performed using the following steps:

1. First, find all methods from the catalog (SYSCAT.ROUTINES) such that all of the following are true:
   - The method name matches the invocation name, and the subject parameter is the same type or is a supertype of the static type of the subject argument.
   - The invoker has the EXECUTE privilege on the method.
   - The number of defined parameters matches the invocation.
   - Each invocation argument matches the method's corresponding defined parameter in data type, or is "promotable" to it.
2. Next, consider each argument of the method invocation, from left to right. The leftmost argument (and thus the first argument) is the implicit SELF parameter. For example, a method defined for type ADDRESS_T has an implicit first parameter of type ADDRESS_T. For each argument, eliminate all functions that are not the best match for that argument. The best match for a given argument is the first data type appearing in the precedence list corresponding to the argument data type for which there exists a function with a parameter of that data type. Length, precision, scale, and the FOR BIT DATA attribute are not considered in this comparison. For example, a DECIMAL(9,1) argument is considered an exact match for a DECIMAL(6,5) parameter, a DECFLOAT(34) argument is considered

an exact match for a DECFLOAT(16) parameter, and a VARCHAR(19) argument is an exact match for a VARCHAR(6) parameter.

The best match for a user-defined structured-type argument is itself; the next best match is its immediate supertype, and so on for each supertype of the argument. Note that only the static type (declared type) of the structured-type argument is considered, not the dynamic type (most specific type).

3. At most, one candidate method remains after Step 2. This is the method that is chosen.
4. If there are no candidate methods remaining after step 2, an error is returned (SQLSTATE 42884).

Example of method resolution

Following is an example of successful method resolution.

There are seven FOO methods for three structured types defined in a hierarchy of GOVERNOR as a subtype of EMPEROR as a subtype of HEADOFSTATE, registered with the following signatures:

```
CREATE METHOD FOO (CHAR(5), INT, DOUBLE)  FOR HEADOFSTATE SPECIFIC FOO_1 ...
CREATE METHOD FOO (INT, INT, DOUBLE)      FOR HEADOFSTATE SPECIFIC FOO_2 ...
CREATE METHOD FOO (INT, INT, DOUBLE, INT) FOR HEADOFSTATE SPECIFIC FOO_3 ...
CREATE METHOD FOO (INT, DOUBLE, DOUBLE)   FOR EMPEROR     SPECIFIC FOO_4 ...
CREATE METHOD FOO (INT, INT, DOUBLE)      FOR EMPEROR     SPECIFIC FOO_5 ...
CREATE METHOD FOO (SMALLINT, INT, DOUBLE) FOR EMPEROR     SPECIFIC FOO_6 ...
CREATE METHOD FOO (INT, INT, DEC(7,2))    FOR GOVERNOR    SPECIFIC FOO_7 ...
```

The method reference is as follows (where I1 and I2 are INTEGER columns, D is a DECIMAL column and E is an EMPEROR column):

```
SELECT E..FOO(I1, I2, D) ...
```

Following through the algorithm...

- FOO_7 is eliminated as a candidate, because the type GOVERNOR is a subtype (not a supertype) of EMPEROR.
- FOO_3 is eliminated as a candidate, because it has the wrong number of parameters.
- FOO_1 and FOO_6 are eliminated because, in both cases, the first argument (not the subject argument) cannot be promoted to the data type of the first parameter. Because there is more than one candidate remaining, the arguments are considered in order.
- For the subject argument, FOO_2 is a supertype, while FOO_4 and FOO_5 match the subject argument.
- For the first argument, the remaining methods, FOO_4 and FOO_5, are an exact match with the argument type. No methods can be eliminated from consideration; therefore the next argument must be examined.
- For this second argument, FOO_5 is an exact match, but FOO_4 is not, so it is eliminated from consideration. This leaves FOO_5 as the method chosen.

Method invocation

Once the method is selected, there are still possible reasons why the use of the method may not be permitted.

Each method is defined to return a result with a specific data type. If this result data type is not compatible with the context in which the method is invoked, an error will occur. For example, assume that the following methods named STEP are defined, each with a different data type as the result:

```
STEP(SMALLINT) FOR TYPEA RETURNS CHAR(5)
STEP(DOUBLE) FOR TYPEA RETURNS INTEGER
```

and the following method reference (where S is a SMALLINT column and TA is a column of TYPEA):

```
SELECT 3 + TA..STEP(S) ...
```

then, because there is an exact match on argument type, the first STEP is chosen. An error occurs on the statement, because the result type is CHAR(5) instead of a numeric type, as required for an argument of the addition operator.

Starting from the method that has been chosen, the algorithm described in "Dynamic dispatch of methods" is used to build the set of dispatchable methods at compile time. Exactly which method is invoked is described in "Dynamic dispatch of methods".

Note that when the selected method is a type preserving method:

- the static result type following function resolution is the same as the static type of the subject argument of the method invocation
- the dynamic result type when the method is invoked is the same as the dynamic type of the subject argument of the method invocation.

This may be a subtype of the result type specified in the type preserving method definition, which in turn may be a supertype of the dynamic type that is actually returned when the method is processed.

In cases where the arguments of the method invocation were not an exact match to the data types of the parameters of the selected method, the arguments are converted to the data type of the parameter at execution using the same rules as assignment to columns. This includes the case where precision, scale, or length differs between the argument and the parameter, but excludes the case where the dynamic type of the argument is a subtype of the parameter's static type.

Dynamic dispatch of methods

Methods provide the functionality and encapsulate the data of a type. A method is defined for a type and can always be associated with this type. One of the method's parameters is the implicit SELF parameter. The SELF parameter is of the type for which the method has been declared. The argument that is passed as the SELF argument when the method is invoked in a DML statement is called *subject*.

When a method is chosen using method resolution (see “Method resolution” on page 126), or a method has been specified in a DDL statement, this method is known as the "most specific applicable authorized method". If the subject is of a structured type, that method could have one or more overriding methods. A determination is then made to select which method to invoke, based on the dynamic type (most specific type) of the subject at run time. This determination is called "determining the most specific dispatchable method". That process is described here.

1. Find the original method in the method hierarchy that the most specific applicable authorized method is part of. This is called the *root method*.
2. Create the set of dispatchable methods, which includes the following:
   - The most specific applicable authorized method.
   - Any method that overrides the most specific applicable authorized method, and which is defined for a type that is a subtype of the subject of this invocation.
3. Determine the most specific dispatchable method, as follows:
   a. Start with an arbitrary method that is an element of the set of dispatchable methods and that is a method of the dynamic type of the subject, or of one of its supertypes. This is the initial most specific dispatchable method.
   b. Iterate through the elements of the set of dispatchable methods. For each method: If the method is defined for one of the proper subtypes of the type for which the most specific dispatchable method is defined, and if it is defined for one of the supertypes of the most specific type of the subject, then repeat step 2 with this method as the most specific dispatchable method; otherwise, continue iterating.
4. Invoke the most specific dispatchable method.

Example:

Given are three types, "Person", "Employee", and "Manager". There is an original method "income", defined for "Person", which computes a person's income. A person is by default unemployed (a child, a

retiree, and so on). Therefore, "income" for type "Person" always returns zero. For type "Employee" and for type "Manager", different algorithms have to be applied to calculate the income. Hence, the method "income" for type "Person" is overridden in "Employee" and "Manager".

Create and populate a table as follows:

```
CREATE TABLE aTable (id integer, personColumn Person);
  INSERT INTO aTable VALUES (0, Person()), (1, Employee()), (2, Manager());
```

List all persons who have a minimum income of $40000:

```
SELECT id, person, name
  FROM aTable
  WHERE person..income() >= 40000;
```

The method "income" for type "Person" is chosen, using method resolution, to be the most specific applicable authorized method.

1. The root method is "income" for "Person" itself.
2. The second step of the previous algorithm is carried out to construct the set of dispatchable methods:
   - The method "income" for type "Person" is included, because it is the most specific applicable authorized method.
   - The method "income" for type "Employee", and "income" for "Manager" is included, because both methods override the root method, and both "Employee" and "Manager" are subtypes of "Person".

   Therefore, the set of dispatchable methods is: {"income" for "Person", "income" for "Employee", "income" for "Manager"}.
3. Determine the most specific dispatchable method:
   - For a subject whose most specific type is "Person":
     a. Let the initial most specific dispatchable method be "income" for type "Person".
     b. Because there is no other method in the set of dispatchable methods that is defined for a proper subtype of "Person" and for a supertype of the most specific type of the subject, "income" for type "Person" is the most specific dispatchable method.
   - For a subject whose most specific type is "Employee":
     a. Let the initial most specific dispatchable method be "income" for type "Person".
     b. Iterate through the set of dispatchable methods. Because method "income" for type "Employee" is defined for a proper subtype of "Person" and for a supertype of the most specific type of the subject (Note: A type is its own super- and subtype.), method "income" for type "Employee" is a better match for the most specific dispatchable method. Repeat this step with method "income" for type "Employee" as the most specific dispatchable method.
     c. Because there is no other method in the set of dispatchable methods that is defined for a proper subtype of "Employee" and for a supertype of the most specific type of the subject, method "income" for type "Employee" is the most specific dispatchable method.
   - For a subject whose most specific type is "Manager":
     a. Let the initial most specific dispatchable method be "income" for type "Person".
     b. Iterate through the set of dispatchable methods. Because method "income" for type "Manager" is defined for a proper subtype of "Person" and for a supertype of the most specific type of the subject (Note: A type is its own super- and subtype.), method "income" for type "Manager" is a better match for the most specific dispatchable method. Repeat this step with method "income" for type "Manager" as the most specific dispatchable method.
     c. Because there is no other method in the set of dispatchable methods that is defined for a proper subtype of "Manager" and for a supertype of the most specific type of the subject, method "income" for type "Manager" is the most specific dispatchable method.
4. Invoke the most specific dispatchable method.

Conservative binding semantics

Object resolution takes place when defining an SQL object or processing a package bind operation.

The database manager chooses which particular defined SQL object to use for an SQL object referenced in a DDL statement or coded in an application.

At a later time, the database manager might resolve to a different SQL object, even though the original SQL object did not change in any way. This resolution to a different SQL object happens as a result of defining another SQL object (or adding a privilege to an existing function) that the object resolution algorithm defines as resolved ahead of the SQL object originally chosen. Examples of SQL objects and situations to which this resolution to a different SQL object applies include the following situations:

- Routines - a new routine could be defined that is a better fit or that is an equally good fit but earlier in the SQL path; or a privilege could be granted to an existing routine that is a better fit or that is an equally good fit but earlier in the SQL path
- User-defined data types - a new user-defined data type could be defined with the same name and in a schema that is earlier in the SQL path
- Global variables - a new global variable could be defined with the same name and in a schema that is earlier in the SQL path
- Tables or views that resolve using a public alias - an actual table, view, or private alias could be defined with the same name in the current schema
- Sequences that resolve using a public sequence alias - an actual sequence or private sequence alias could be defined with the same name in the current schema
- Modules that resolve to a public module alias - an actual module or private module alias could be defined with the same name in a schema that is in the SQL path

There are instances where the database manager must be able to repeat the resolution of SQL objects as originally resolved when the statement was processed. This is true when the following static objects are used:

- Static DML statements in packages
- Views
- Triggers
- Check constraints
- SQL routines
- Global variables with a user-defined type or default expression
- Routines with a user-defined parameter type or default expression

For static DML statements in packages, SQL object references are resolved during a bind operation. SQL object references in views, triggers, SQL routines, and check constraints are resolved when the SQL object is defined or revalidated. When an existing static object is used, *conservative binding semantics* are applied unless the object has been marked invalid or inoperative by a change in the database schema.

Conservative binding semantics ensure that SQL object references will be resolved using the same SQL path, default schema, and set of routines that were used when it was previously resolved. It also ensures that the timestamp of the definition of the SQL objects considered during conservative binding resolution is not later than the timestamp when the statement was last bound or validated using *non-conservative binding* semantics. Non-conservative binding semantics use the same SQL path and default schema as the original generation of the package or statement, but does not consider the timestamp of the definition of the SQL objects and does not consider any previously resolved set of routines.

Some changes to the database schema, such as dropping objects, altering objects, or revoking privileges, can impact an SQL object so that the database manager can no longer resolve all dependent SQL objects of an existing SQL object using conservative binding semantics.

- When this happens for a static statement in an SQL package, the package is marked inoperative. The next use of a statement in this package will cause an implicit rebind of the package using non-

conservative binding semantics in order to be able to resolve to SQL objects considering the latest changes in the database schema that caused that package to become inoperative.

- When this happens for a view, trigger, check constraint, or SQL routine, the SQL object is marked invalid. The next use of the object causes an implicit revalidation of the SQL object using non-conservative binding semantics.

Consider a database with two functions that have the signatures SCHEMA1.BAR(INTEGER) and SCHEMA2.BAR(DOUBLE). Assume the SQL path contains both schemas SCHEMA1 and SCHEMA2 (although their order within the SQL path does not matter). USER1 has been granted the EXECUTE privilege on the function SCHEMA2.BAR(DOUBLE). Suppose USER1 creates a view that invokes BAR(INT_VAL), where INT_VAL is a column or global variable defined with the INTEGER data type. This function reference in the view resolves to the function SCHEMA2.BAR(DOUBLE) because USER1 does not have the EXECUTE privilege on SCHEMA1.BAR(INTEGER). If USER1 is granted the EXECUTE privilege on SCHEMA1.BAR(INTEGER) after the view has been created, the view will continue to use SCHEMA2.BAR(DOUBLE) unless a database schema change causes the view to be marked invalid. The view is marked invalid if a required privilege is revoked or an object it depends on gets dropped or altered.

For static DML in packages, packages can rebind implicitly, or by explicitly issuing the REBIND command (or corresponding API), or the BIND command (or corresponding API). The implicit rebind is performed with conservative binding semantics if the package is marked invalid, but uses non-conservative binding semantics when the package is marked inoperative. A package is marked invalid only if an index on which it depends is dropped or altered. The REBIND command provides the option to resolve with conservative binding semantics (RESOLVE CONSERVATIVE) or to resolve by considering new routines, data types, or global variables (RESOLVE ANY, which is the default option). The RESOLVE CONSERVATIVE option can be used only if the package has not been marked inoperative by the database manager (SQLSTATE 51028).

The description of conservative binding semantics in this topic has assumed that the database configuration parameter **auto_reval** has a setting other than DISABLED. The default setting for **auto_reval** for new databases is DEFERRED. The default setting for **auto_reval** for databases upgraded to Version 9.7 is DISABLED. If **auto_reval** is set to DISABLED, then conservative binding semantics, invalidation, and revalidation behavior are the same as in releases previous to Version 9.7. Under a **auto_reval** setting of DISABLED, conservative binding semantics only considers the timestamp of the definition of the SQL objects for functions, methods, user-defined types, and global variables. For invalidation and revalidation behavior, this means, in the case of the DROP, REVOKE, and ALTER statements, that either the semantics are more restrictive or the impact on dependent objects is to cascade and drop the object. In the case of packages, most database schema changes result in marking the package invalid and using conservative binding semantics during implicit rebind. However, when the schema is changed by dropping a dependent function and **auto_reval** is set to DISABLED, the package dependent on the function is marked inoperative and there is no implicit rebind of the inoperative package.

Expressions

An expression specifies a value. It can be a simple value, consisting of only a constant or a column name, or it can be more complex. If you repeatedly use similar complex expressions, consider using an SQL function to encapsulate a common expression.

Authorization

The use of some of the expressions, such as a scalar-subselect, sequence-reference, or function-invocation, might require special authorization. For these expressions, the privileges held by the authorization ID of the statement must include the following authorizations:

scalar-subselect
For information about authorization considerations, see "SQL Queries".

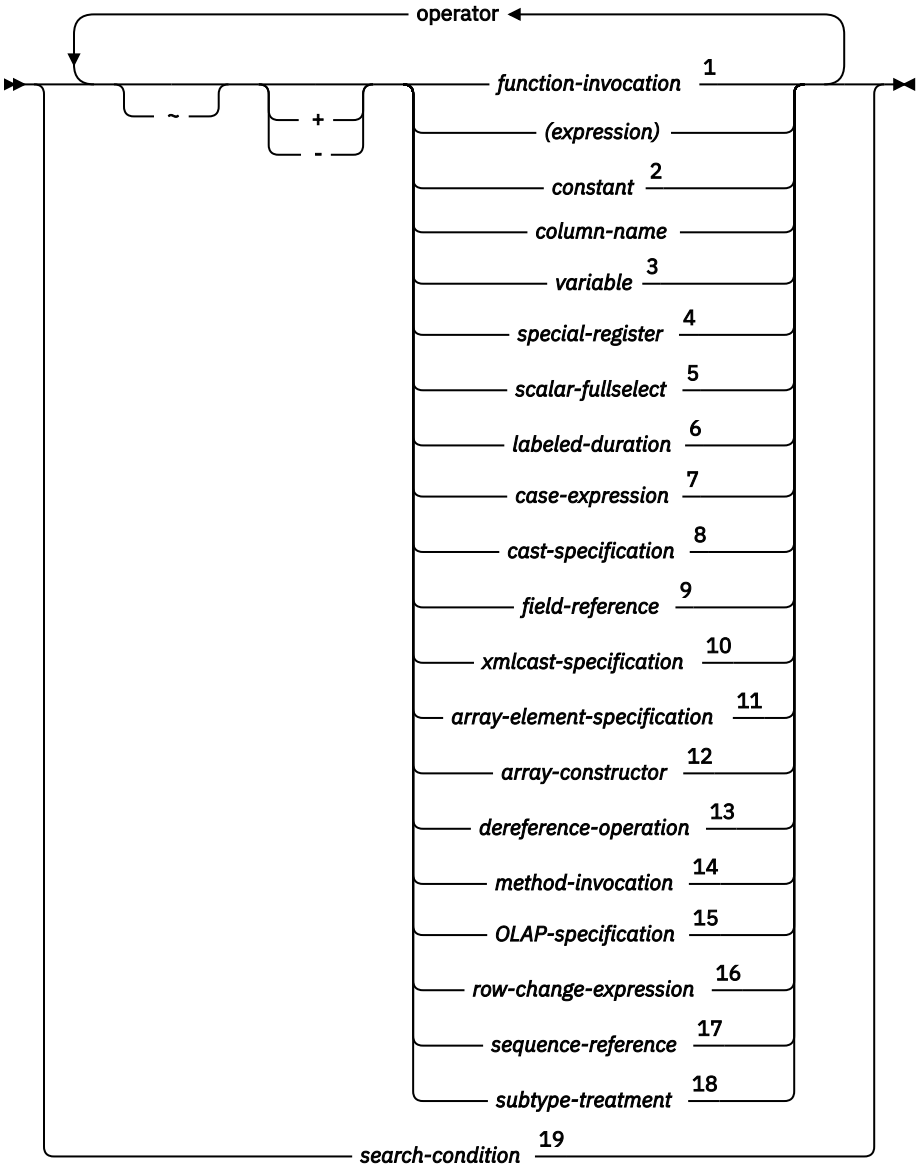
sequence-reference
The authorization to reference the sequence. For information about authorization considerations, see "Sequence authorization".

function-invocation
The authorization to execute a user-defined function. For information about authorization considerations, see the "Function invocation" section in "Functions".

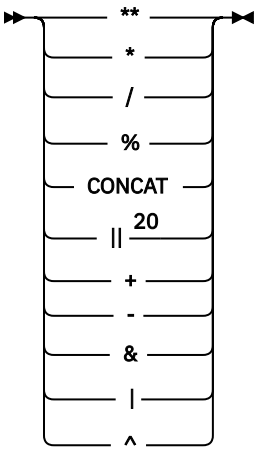
variable
If the variable is a global variable, the authorization to reference the global variable is required. For information, see "Global variables".

In a Unicode database, an expression that accepts a character or graphic string will accept any string types for which conversion is supported.

expression



operator



Notes:

[1] See "Function invocation" on page 114 for more information.
[2] See "Constants" on page 83 for more information.
[3] See "References to variables" on page 19 for more information.
[4] See "Special registers" on page 88 for more information.
[5] See "Scalar fullselect" on page 145 for more information.
[6] See "Durations" on page 145 for more information.
[7] See "CASE expression" on page 150 for more information.
[8] See "CAST specification" on page 152 for more information.
[9] See "Field reference" on page 157 for more information.
[10] See "XMLCAST specification" on page 158 for more information.
[11] See "ARRAY element specification" on page 159 for more information.
[12] See "Array constructor" on page 160 for more information.
[13] See "Dereference operation" on page 161 for more information.
[14] See "Method invocation" on page 162 for more information.
[15] See "OLAP specification" on page 163 for more information.
[16] See "ROW CHANGE expression" on page 175 for more information.
[17] See "Sequence reference" on page 176 for more information.
[18] See "Subtype treatment" on page 182 for more information.
[19] See "Search conditions" on page 191 for more information.
[20] The || operator can be used as a synonym for CONCAT.

Expressions without operators

If no operators are used, the result of the expression is the specified value. For example:

```
SALARY:SALARY'SALARY'MAX(SALARY)
```

Expressions with the concatenation operator

The concatenation operator (CONCAT or ||) combines two operands to form a *string expression*.

The first operand is an expression that returns a value of any string data type, any numeric data type, or any datetime data type. The second operand is also an expression that returns a value of any string data type, any numeric data type, or any datetime data type. However, some data types are not supported in combination with the data type of the first operand, as described in the remainder of this section.

Each operand can be of any of the following types:

- String (except binary string)

- Numeric (this is implicitly cast to VARCHAR)
- Datetime (this is implicitly cast to VARCHAR)
- Boolean (this is implicitly cast to VARCHAR)

However, a binary string can be concatenated only with another binary string or with a character string that is defined as FOR BIT DATA.

Concatenation involving both character string operands and graphic string operands is supported only in a Unicode database. Character operands are first converted to the graphic data type before the concatenation. Character strings defined as FOR BIT DATA cannot be cast to the graphic data type.

If either operand can be null, the result can be null, and if either is null, the result is the null value. Otherwise, the result consists of the first operand string followed by the second. Note that no check is made for improperly formed mixed data when doing concatenation.

The length of the result is the sum of the lengths of the operands. In a Unicode database, the string unit of the result is the maximum string unit of the operands, as described in "Rules for result data types".

The data type and length attribute of the result is determined from that of the operands as shown in the following table unless an operand is defined with CODEUNITS32:

Table 21. Data Type and Length of Concatenated Operands without CODEUNITS32

Operands	**Combined Length Attributes**[1]	**Result**
CHAR(A) CHAR(B)	<256	CHAR(A+B)
CHAR(A) CHAR(B)	>255	VARCHAR(A+B)
CHAR(A) VARCHAR(B)	<4001	VARCHAR(A+B)
CHAR(A) VARCHAR(B)	>4000	LONG VARCHAR
CHAR(A) LONG VARCHAR	-	LONG VARCHAR
VARCHAR(A) VARCHAR(B)	<4001	VARCHAR(A+B)
VARCHAR(A) VARCHAR(B)	>4000	LONG VARCHAR
VARCHAR(A) LONG VARCHAR	-	LONG VARCHAR
LONG VARCHAR LONG VARCHAR	-	LONG VARCHAR
CLOB(A) CHAR(B)	-	CLOB(MIN(A+B, 2147483647))
CLOB(A) VARCHAR(B)	-	CLOB(MIN(A+B, 2147483647))
CLOB(A) LONG VARCHAR	-	CLOB(MIN(A+32700, 2147483647))
CLOB(A) CLOB(B)	-	CLOB(MIN(A+B, 2147483647))
GRAPHIC(A) GRAPHIC(B)	<128	GRAPHIC(A+B)
GRAPHIC(A) GRAPHIC(B)	>127	VARGRAPHIC(A+B)
GRAPHIC(A) VARGRAPHIC(B)	<2001	VARGRAPHIC(A+B)
GRAPHIC(A) VARGRAPHIC(B)	>2000	LONG VARGRAPHIC
GRAPHIC(A) LONG VARGRAPHIC	-	LONG VARGRAPHIC
VARGRAPHIC(A) VARGRAPHIC(B)	<2001	VARGRAPHIC(A+B)
VARGRAPHIC(A) VARGRAPHIC(B)	>2000	LONG VARGRAPHIC

Table 21. Data Type and Length of Concatenated Operands without CODEUNITS32 (continued)		
Operands	**Combined Length Attributes[1]**	**Result**
VARGRAPHIC(A) LONG VARGRAPHIC	-	LONG VARGRAPHIC
LONG VARGRAPHIC LONG VARGRAPHIC	-	LONG VARGRAPHIC
DBCLOB(A) GRAPHIC(B)	-	DBCLOB(MIN(A+B, 1073741823))
DBCLOB(A) VARGRAPHIC(B)	-	DBCLOB(MIN(A+B, 1073741823))
DBCLOB(A) LONG VARGRAPHIC	-	DBCLOB(MIN(A+16350, 1073741823))
DBCLOB(A) DBCLOB(B)	-	DBCLOB(MIN(A+B, 1073741823))
BINARY(A) BINARY(B)	<256	BINARY(A+B)
BINARY(A) CHAR(B) FOR BIT DATA	<256	BINARY(A+B)
BINARY(A) BINARY(B)	>255	VARBINARY(A+B)
BINARY(A) CHAR(B) FOR BIT DATA	>255	VARBINARY(A+B)
BINARY(A) VARBINARY(B)	-	VARBINARY(MIN(A+B, 32672))
BINARY(A) VARCHAR(B) FOR BIT DATA	-	VARBINARY(MIN(A+B, 32672))
VARBINARY(A) VARBINARY(B)	-	VARBINARY(MIN(A+B, 32672))
VARBINARY(A) CHAR(B) FOR BIT DATA	-	VARBINARY(MIN(A+B, 32672))
VARBINARY(A) VARCHAR(B) FOR BIT DATA	-	VARBINARY(MIN(A+B, 32672))
BLOB(A) BINARY(B)	-	BLOB(MIN(A+B, 2147483647))
BLOB(A) CHAR(B) FOR BIT DATA	-	BLOB(MIN(A+B, 2147483647))
BLOB(A) VARBINARY(B)	-	BLOB(MIN(A+B, 2147483647))
BLOB(A) VARCHAR(B) FOR BIT DATA	-	BLOB(MIN(A+B, 2147483647))
BLOB(A) BLOB(B)	-	BLOB(MIN(A+B, 2147483647))
Notes [1] The numbers specified for the Combined Length Attributes are listed in OCTETS for character strings and CODEUNITS16 for graphic strings. Refer to the next table if an operand is defined with CODEUNITS32.		

If an operand is defined with CODEUNITS32, the other operand cannot be defined as FOR BIT DATA. Otherwise, when an operand is defined with CODEUNITS32, the data type and length attribute of the result is determined from that of the operands as shown in the following table:

Table 22. Data Type and Length of Concatenated Operands with CODEUNITS32		
Operands	**Combined Length Attributes**	**Result**
CHAR(A) CHAR(B)	<64	CHAR(A+B)
CHAR(A) CHAR(B)	>63	VARCHAR(A+B)
CHAR(A) VARCHAR(B)	-	VARCHAR(MIN(A+B, 8168))

Table 22. Data Type and Length of Concatenated Operands with CODEUNITS32 (continued)		
Operands	**Combined Length Attributes**	**Result**
VARCHAR(A) VARCHAR(B)	-	VARCHAR(MIN(A+B, 8168))
CLOB(A) CHAR(B)	-	CLOB(MIN(A+B, 536870911))
CLOB(A) VARCHAR(B	-	CLOB(MIN(A+B, 536870911))
CLOB(A) CLOB(B)	-	CLOB(MIN(A+B, 536870911))
GRAPHIC(A) GRAPHIC(B)	<64	GRAPHIC(A+B)
GRAPHIC(A) GRAPHIC(B)	>63	VARGRAPHIC(A+B)
GRAPHIC(A) VARGRAPHIC(B)	-	VARGRAPHIC(MIN(A+B, 8168))
VARGRAPHIC(A) VARGRAPHIC(B)	-	VARGRAPHIC(MIN(A+B, 8168))
DBCLOB(A) CHAR(B)	-	DBCLOB(MIN(A+B, 536870911))
DBCLOB(A) VARCHAR(B)	-	DBCLOB(MIN(A+B, 536870911))
DBCLOB(A) DBCLOB(B)	-	DBCLOB(MIN(A+B, 536870911))

For compatibility with previous versions or other database products, there is no automatic escalation of results involving LONG VARCHAR or LONG VARGRAPHIC data types to LOB data types. For example, concatenation of a CHAR(200) value and a completely full LONG VARCHAR value would result in an error rather than in a promotion to a CLOB data type.

The code page of the result is considered a derived code page and is determined by the code page of its operands.

One operand may be a parameter marker. If a parameter marker is used, then the data type and length attributes of that operand are considered to be the same as those for the non-parameter marker operand. The order of operations must be considered to determine these attributes in cases with nested concatenation.

Example 1: If FIRSTNME is Pierre and LASTNAME is Fermat, then the following:

```
    FIRSTNME CONCAT ' ' CONCAT LASTNAME
```

returns the value `Pierre Fermat`.

Example 2: Given:

- The column `COLA` is defined as VARCHAR(5) and the value `'AA'` was inserted into it.
- The character host variable `:host_var` is defined with length 5 and value `'BB  '`.
- The column `COLC` defined as CHAR(5) and the value `'CC'` was inserted into it.
- The column `COLD` defined as CHAR(5) and the value `'DDDDD'` was inserted into it.

The value of `COLA` **`CONCAT`** `:host_var` **`CONCAT`** `COLC` **`CONCAT`** `COLD` is `'AABB  CC  DDDDD'`

The data type of the result is VARCHAR and its length attribute is 2+5+5+5=17. The result code page is the section code page. For more information about section code pages, see "Derivation of code page values".

Example 3: Given:

- `COLA` defined as CHAR(10)
- `COLB` defined as VARCHAR(5)

The parameter marker in the expression:

```
COLA CONCAT COLB CONCAT ?
```

is considered VARCHAR(15), because COLA CONCAT COLB is evaluated first, giving a result that is the first operand of the second CONCAT operation.

User-defined types and the concatenation operator

A weakly typed distinct type is the only user-defined type that can be used with the concatenation operator. The source type of the weakly typed distinct type is used as the data type of the operand when processing the concatenation operator.

A strongly typed user-defined type cannot be used with the concatenation operator, even if it is a strongly typed distinct type with a source data type that is a string type. To concatenate, create a function with the CONCAT operator as its source. For example, if there were distinct types TITLE and TITLE_DESCRIPTION, both of which had VARCHAR(25) data types, the following user-defined function, ATTACH, can be used to concatenate them.

```
CREATE FUNCTION ATTACH (TITLE, TITLE_DESCRIPTION)
  RETURNS VARCHAR(50) SOURCE CONCAT (VARCHAR(), VARCHAR())
```

Alternately, the concatenation operator can be overloaded using a user-defined function to add the new data types.

```
CREATE FUNCTION CONCAT (TITLE, TITLE_DESCRIPTION)
  RETURNS VARCHAR(50) SOURCE CONCAT (VARCHAR(), VARCHAR())
```

Expressions with arithmetic operators

If an expression uses arithmetic operators, the result of the expression is a value derived from the application of the operators to the values of the operands. These operators can be specified using either infix or prefix notation. For example, the statements a+b and "+"(a,b) are equivalent.

If any operand in an arithmetic expression can be null, or if the database is configured with **dft_sqlmathwarn** set to *yes*, the result can be null. If any operand in an arithmetic expression has the null value, the result of the expression is the null value.

Arithmetic operators can be applied to signed numeric types and datetime types (see "Datetime arithmetic in SQL" on page 147). For example, USER+2 is invalid. When any operand of an arithmetic operation is a weakly typed distinct type, the operation is performed assuming that the data type of the operand is the source type of the weakly typed distinct type. Sourced functions can be defined for arithmetic operations on strongly typed distinct types with a source type that is a signed numeric type.

The unary plus (+) prefix operator does not change its operand. The unary minus (-) prefix operator reverses the sign of:

- A nonzero non-decimal floating-point operand
- A decimal floating-point operand, including 0, -0, quiet NaNs, signalling NaNs, +infinity, and -infinity

If the data type of a value x is small integer, the data type of -x is large integer. The first character of the token following a prefix operator cannot be a plus (+) or minus (-) symbol.

Table 23. Binary Arithmetic Operators

Arithmetic Operator	Name	Description
+	Addition	The result is the sum of the first and second arguments.
-	Subtraction	The result is the first argument minus the second argument.
*	Multiplication	The result is the first argument multiplied by the second argument.

Table 23. Binary Arithmetic Operators (continued)		
Arithmetic Operator	**Name**	**Description**
/	Division	The result is the first argument divided by the second argument. The value of the second operand may not be zero, unless the calculation is performed using decimal floating-point arithmetic.
%	Modulo	The result is the remainder of the first argument divided by the second argument.
**	Exponential	The result is the first argument raised to the power of the second argument. The data type of the result is: • INTEGER if both arguments are of type INTEGER or SMALLINT • BIGINT if one argument is of type BIGINT and the other argument is of type BIGINT, INTEGER, or SMALLINT • DECFLOAT(34) if either or both arguments are of type DECFLOAT, unless one of the following statements is true, in which case the result is of type NAN and an invalid operation condition is issued: – Both arguments are zero. – The second argument has a nonzero fractional part. – The second argument has more than 9 digits. – The second argument is INFINITY. • DOUBLE otherwise

Before an arithmetic operation is performed, an operand that is a non-LOB string is converted to DECFLOAT(34) using the rules for CAST specification. For more information, see "Casting between data types" on page 47. Arithmetic operations involving graphic string operands can be performed only for Unicode databases.

Arithmetic errors

If an arithmetic error such as divide-by-zero or a numeric overflow occurs during the processing of an non-decimal floating-point expression, an error is returned (SQLSTATE 22003 or 22012). For decimal floating-point expressions, a warning is returned (SQLSTATEs 0168C, 0168D, 0168E, or 0168F, depending on the nature of the arithmetic condition).

A database can be configured (using **dft_sqlmathwarn** set to YES) so that arithmetic errors return a null value for the non-decimal floating-point expression, the query returns a warning (SQLSTATE 01519 or 01564), and proceeds with processing the SQL statement.

For decimal floating-point expressions, the setting of **dft_sqlmathwarn** has no effect; arithmetic conditions return an appropriate value (possibly a decimal floating-point special value), the query returns a warning (SQLSTATEs 0168C, 0168D, 0168E, or 0168F), and proceeds with processing of the SQL statement. Special values returned include plus and minus infinity and not a number. Arithmetic expressions involving one or more decimal floating-point numbers never evaluate to a null value unless one or more of the arguments to the expression are null.

When arithmetic errors are treated as nulls, there are implications on the results of SQL statements. The following are some examples of these implications.

- An arithmetic error that occurs in the expression that is the argument of an aggregate function causes the row to be ignored in the determining the result of the aggregate function. If the arithmetic error was an overflow, this may significantly impact the result values.

- An arithmetic error that occurs in the expression of a predicate in a WHERE clause can cause rows to not be included in the result.
- An arithmetic error that occurs in the expression of a predicate in a check constraint results in the update or insert proceeding since the constraint is not false.

If these types of impacts are not acceptable, additional steps should be taken to handle the arithmetic error to produce acceptable results. Examples:

- Add a case expression to check for divide-by-zero and set the required value for such a situation
- Add additional predicates to handle nulls (like a check constraint on not nullable columns could become:

  ```
  check (c1*c2 is not null and c1*c2>5000)
  ```

  to cause the constraint to be violated on an overflow).

Two integer operands

If both operands of an arithmetic operator are integers, the operation is performed in binary and the result is a *large integer* unless either (or both) operand is a big integer, in which case the result is a big integer. Any remainder of division is lost. The result of an integer arithmetic operation (including negation by means of a unary minus operator) must be within the range of the result type.

Integer and decimal operands

If one operand is an integer and the other is a decimal, the operation is performed in decimal using a temporary copy of the integer that has been converted to a decimal number with precision *p* and scale 0; *p* is 19 for a big integer, 11 for a large integer, and 5 for a small integer.

Two decimal operands

If both operands are decimal, the operation is performed in decimal. The result of any decimal arithmetic operation is a decimal number with a precision and scale that are dependent on the operation and the precision and scale of the operands. If the operation is addition or subtraction and the operands do not have the same scale, the operation is performed with a temporary copy of one of the operands. The copy of the shorter operand is extended with trailing zeros so that its fractional part has the same number of digits as the longer operand.

The result of a decimal operation cannot have a precision greater than 31. The result of decimal addition, subtraction, and multiplication is derived from a temporary result which can have a precision greater than 31. If the precision of the temporary result is not greater than 31, the final result is the same as the temporary result.

Decimal arithmetic in SQL

Use the formulas shown here to calculate the precision and scale of the result of decimal operations in SQL. The formulas use the following symbols:

p
Precision of the first operand.

s
Scale of the first operand.

p'
Precision of the second operand.

s'
Scale of the second operand.

Addition and subtraction
The scale of the result is `MAX(s,s')`. If DEC15 mode is in effect and p<15 and p'<15, the precision is `MIN(15,MAX(p-s,p'-s')+MAX(s,s')+1)`. Otherwise, the precision is `MIN(31,MAX(p-s,p'-s')+MAX(s,s')+1)`.

Multiplication
If DEC15 mode is in effect and p<15 and p'<15, the precision is `MIN(15,p+p')` and the scale is `MIN(15,s+s')`. Otherwise the precision is `MIN(31,p+p')` and the scale is `MIN(31,s+s')`.

Division
The following table shows the result precision and scale based on various factors.

Table 24. Precision and scale of the result of a decimal division

DECIMAL arithmetic mode[1]	p	p'	Result precision	Result scale
default	n/a	n/a	31	`31-p+s-s'`
DEC15	<=15	<=15	15	`15-(p-s+s')`
DEC15	>15	<=15	31	*N*`-(p-s+s')`, where: • *N* is `30-p' if p'` is odd • *N* is `29-p' if p'` is even
DEC31	n/a	<=15	31	*N*`-(p-s+s')`, where: • *N* is `30-p' if p'` is odd • *N* is `29-p' if p'` is even
DEC15 or DEC31	n/a	>15	31	`15-(p-s+MAX(0,s'-(p'-15)))`

Note:

1. These modes are determined by the **dec_arithmetic** configuration parameter.

If a minimum DECIMAL division scale *S* is in effect, then the scale is the minimum of *S* and the scale derived from Table 24 on page 141. Otherwise, a negative scale results in an error (SQLSTATE 42911).

Exponential
The result type is DOUBLE.

Floating-point operands

If either operand of an arithmetic operator is floating-point, but not decimal floating-point, the operation is performed in floating-point. The operands are first converted to double-precision floating-point numbers, if necessary. Thus, if any element of an expression is a floating-point number, the result of the expression is a double-precision floating-point number.

An operation involving a floating-point number and an integer is performed with a temporary copy of the integer which has been converted to double-precision floating-point. An operation involving a floating-point number and a decimal number is performed with a temporary copy of the decimal number which has been converted to double-precision floating-point. The result of a floating-point operation must be within the range of floating-point numbers.

The order in which floating-point operands (or arguments to functions) are processed can slightly affect results because floating-point operands are approximate representations of real numbers. Since the order in which operands are processed may be implicitly modified by the optimizer (for example, the optimizer may decide what degree of parallelism to use and what access plan to use), an application that uses floating-point operands should not depend on the results being precisely the same each time an SQL statement is executed.

Decimal floating-point operands

If either operand of an arithmetic operator is decimal floating-point, the operation is performed in decimal floating-point.

Integer and decimal floating-point operands
If one operand is a small integer or large integer and the other is a DECFLOAT(*n*) number, the operation is performed in DECFLOAT(*n*) using a temporary copy of the integer that has been converted to a DECFLOAT(*n*) number. If one operand is a big integer, and the other is a decimal floating-point number, a temporary copy of the big integer is converted to a DECFLOAT(34) number. The rules for two-decimal floating-point operands then apply.

Decimal and decimal floating-point operands
If one operand is a decimal and the other is a decimal floating-point number, the operation is performed in decimal floating-point using a temporary copy of the decimal number that has been converted to a decimal floating-point number based on the precision of the decimal number. If the decimal number has a precision less than 17, the decimal number is converted to a DECFLOAT(16) number; otherwise, the decimal number is converted to a DECFLOAT(34) number. The rules for two-decimal floating-point operands then apply.

Floating-point and decimal floating-point operands
If one operand is a floating-point number (REAL or DOUBLE) and the other is a DECFLOAT(*n*) number, the operation is performed in decimal floating-point using a temporary copy of the floating-point number that has been converted to a DECFLOAT(*n*) number.

Two decimal floating-point operands
If both operands are DECFLOAT(*n*), the operation is performed in DECFLOAT(*n*). If one operand is DECFLOAT(16) and the other is DECFLOAT(34), the operation is performed in DECFLOAT(34).

General arithmetic operation rules for decimal floating-point

The following general rules apply to all arithmetic operations on the decimal floating-point data type:

- Every operation on finite numbers is carried out as though an exact mathematical result is computed, using integer arithmetic on the coefficient, where possible.

  If the coefficient of the theoretical exact result has no more than the number of digits that reflect its precision (16 or 34), it is used for the result without change (unless there is an underflow or overflow condition). If the coefficient has more than the number of digits that reflect its precision, it is rounded to exactly the number of digits that reflect its precision (16 or 34), and the exponent is increased by the number of digits that are removed.

  The CURRENT DECFLOAT ROUNDING MODE special register determines the rounding mode.

  If the value of the adjusted exponent of the result is less than E_{min}, the calculated coefficient and exponent form the result, unless the value of the exponent is less than E_{tiny}, in which case the exponent is set to E_{tiny}, the coefficient is rounded (possibly to zero) to match the adjustment of the exponent, and the sign remains unchanged. If this rounding gives an inexact result, an underflow exception condition is returned.

  If the value of the adjusted exponent of the result is larger than E_{max}, an overflow exception condition is returned. In this case, the result is defined as an overflow exception condition and might be infinite. It has the same sign as the theoretical result.
- Arithmetic that uses the special value infinity follows the usual rules, where negative infinity is less than every finite number and positive infinity is greater than every finite number. Under these rules, an infinite result is always exact. Certain uses of infinity return an invalid operation condition. The following list shows the operations that can cause an invalid operation condition. The result of such an operation is a NaN when one of the operands is infinity but the other operand is not a NaN (quiet or signalling).
  - Add +infinity to -infinity during an addition or subtraction operation
  - Multiply 0 by +infinity or -infinity
  - Divide either +infinity or -infinity by either +infinity or -infinity

– Either argument of the QUANTIZE function is +infinity or -infinity
– The second argument of the POWER function is +infinity or -infinity
– A signaling NaN is an operand of an arithmetic operation

The following rules apply to arithmetic operations and the NaN value:

– The result of any arithmetic operation that has a NaN (quiet or signalling) operand is NaN. The sign of the result is copied from the first operand that is a signalling NaN; if neither operand is signalling, the sign is copied from the first operand that is a NaN. Whenever a result is a NaN, the sign of the result depends only on the copied operand.
– The sign of the result of a multiplication or division operation is negative only if the operands have different signs and neither is a NaN.
– The sign of the result of an addition or subtraction operation is negative only if the result is less than zero and neither operand is a NaN, except for the following cases, in which the result is -0:
  - A result is rounded to 0, and the value, before rounding, had a negative sign
  - 0 is subtracted from -0
  - Operands with opposite signs are added, or operands with the same sign are subtracted; the result has a coefficient of 0, and the rounding mode is ROUND_FLOOR
  - Operands are multiplied or divided, the result has a coefficient of 0, and the signs of the operands are different
  - The first argument of the POWER function is -0, and the second argument is a positive odd number
  - The argument of the CEIL, FLOOR, or SQRT function is -0
  - The first argument of the ROUND or TRUNCATE function is -0

The following examples show special decimal floating-point values as operands:

```
INFINITY + 1          = INFINITY
INFINITY + INFINITY   = INFINITY
INFINITY + -INFINITY  = NAN         -- warning
NAN + 1               = NAN
NAN + INFINITY        = NAN
1 - INFINITY          = -INFINITY
INFINITY - INFINITY   = NAN         -- warning
-INFINITY - -INFINITY = NAN         -- warning
-0.0 - 0.0E1          = -0.0
-1.0 * 0.0E1          = -0.0
1.0E1 / 0             = INFINITY    -- warning
-1.0E5 / 0.0          = -INFINITY   -- warning
1.0E5 / -0            = -INFINITY   -- warning
INFINITY / -INFINITY  = NAN         -- warning
INFINITY / 0          = INFINITY
-INFINITY / 0         = -INFINITY
-INFINITY / -0        = INFINITY
```

User-defined types as operands of arithmetic operators

Weakly typed distinct type operands can be used with arithmetic operators, provided that source type of the weakly typed distinct type is supported by the arithmetic operator. There is no need to create additional user-defined functions to support arithmetic operations for weakly typed distinct type operands.

A strongly typed user-defined type cannot be used with arithmetic operators, even if its source data type is numeric. To perform an arithmetic operation, create a function with the arithmetic operator as its source. For example, if there were strongly typed distinct types INCOME and EXPENSES, both of which had DECIMAL(8,2) data types, then the following user-defined function, REVENUE, could be used to subtract one from the other.

```
CREATE FUNCTION REVENUE (INCOME, EXPENSES)
  RETURNS DECIMAL(8,2) SOURCE "-" (DECIMAL, DECIMAL)
```

Alternately, the minus (-) operator could be overloaded using a user-defined function to subtract the new data types.

```
CREATE FUNCTION "-" (INCOME, EXPENSES)
  RETURNS DECIMAL(8,2) SOURCE "-" (DECIMAL, DECIMAL)
```

Expressions with bitwise operators

The bitwise operators BITAND (&), BITOR (|), BITXOR (^), and BITNOT (~) correspond to the similarly named scalar functions described in "BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT " on page 293.

If any operand in a bitwise expression can be null, the result can be null. If any operand in a bitwise expression has the null value, the result of the expression is the null value.

The unary bitwise BITNOT (~) prefix operator reverses each bit of the operand to which it applies. If the data type of a value x is DECIMAL, REAL, DOUBLE, or DECFLOAT(16), the data type of ~x is DECFLOAT(34); otherwise, the data type is the same as that of x.

Table 25. Binary Bitwise Operators

Bitwise Operator	Name	Description
&	BITAND	The result is a bit pattern in which each bit is the result of a logical AND operation performed on the corresponding bits of the input arguments.
|	BITOR	The result is a bit pattern in which each bit is the result of a logical OR operation performed on the corresponding bits of the input arguments.
^	BITXOR	The result is a bit pattern in which each bit is the result of a logical XOR (exclusive OR) operation performed on the corresponding bits of the input arguments.

If the data type of either operand in a bitwise BITAND (&), BITOR (|), or BITXOR (^) expression is DECFLOAT, the data type of the result is DECFLOAT(34). Otherwise, the data type of the result is that of the operand whose data type is ranked highest in the order of data type precedence (see Table 6 on page 46).

Before a bitwise operation is performed, an operand that is a non-LOB string is converted to DECFLOAT(34) using the rules for CAST specification. For more information, see "Casting between data types" on page 47. Bitwise operations involving graphic string operands can be performed only for Unicode databases.

Precedence of operations

Expressions within parentheses and dereference operations are evaluated first and from left to right. (Parentheses are also used in fullselects, search conditions, and functions. However, they should not be used to arbitrarily group sections within SQL statements.) When the order of evaluation is not specified by parentheses, operators are evaluated in the following order:

1. Unary BITNOT (~)
2. Unary positive (+) or unary negative (-) prefix
3. Exponential (**)
4. Multiplication (*), division (/), modulo (%), or concatenation (CONCAT or ||)
5. Addition (+) or subtraction (-)
6. BITAND (&), BITOR (|), or BITXOR (^).
7. Predicates.
8. Logical NOT.

9. Logical AND.
10. Logical OR.

Operators at the same precedence level are evaluated from left to right.

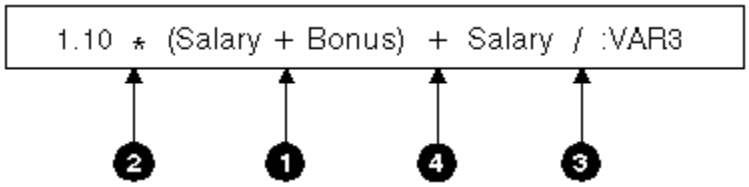


Figure 2. Example illustrating the precedence of operations

Scalar fullselect

Scalar fullselect

A *scalar fullselect* is a fullselect, enclosed in parentheses, that returns a single row consisting of a single column value. If the fullselect does not return a row, the result of the expression is the null value. If the select list element is an expression that is simply a column name or a dereference operation, the result column name is based on the name of the column. The authorization required for a scalar fullselect is the same as that required for an SQL query.

Compatability features

When the SQL_COMPAT global variable is set to 'NPS':

- The symbol ^ is interpreted as the exponential operator (equivalent to **) and not as the BITXOR operator. The symbol ** is also interpreted as the exponential operator.
- The symbol # is interpreted as the BITXOR operator.

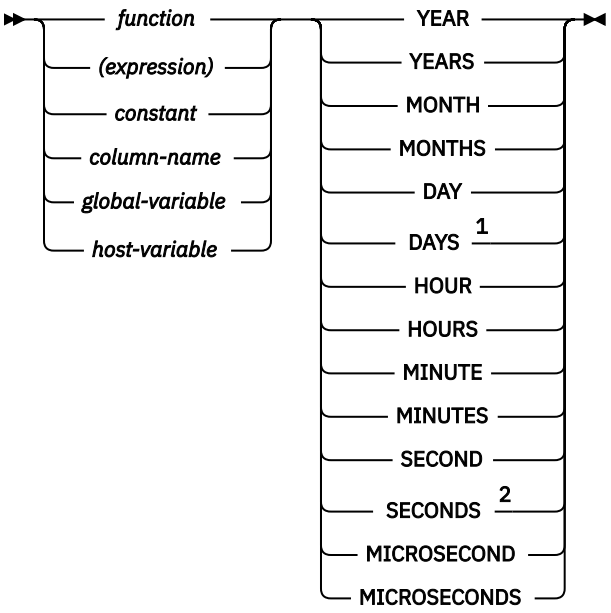
Datetime operations and durations

Datetime values can be incremented, decremented, and subtracted. These operations can involve decimal numbers called durations.

Durations

A *duration* is a number representing an interval of time.

labeled-duration



Notes:

[1] DAYS is the default for non-decimal numeric arithmetic involving dates and timestamps.

[2] SECONDS is the default for non-decimal numeric arithmetic involving time.

A duration can be of one of the following types:

labeled duration
A specific unit of time as expressed by a number (which can be the result of an expression) followed by one of the seven duration keywords: YEARS, MONTHS, DAYS, HOURS, MINUTES, SECONDS, or MICROSECONDS. (The singular form of these keywords is also acceptable: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and MICROSECOND.) The number specified is converted as if it were assigned to a DECIMAL(15,0) number, except for SECONDS which uses DECIMAL(27,12) to allow 0 to 12 digits of fractional seconds to be included. A labeled duration can only be used as an operand of an arithmetic operator in which the other operand is a value of data type DATE, TIME, or TIMESTAMP. Thus, the expression HIREDATE + 2 MONTHS + 14 DAYS is valid, whereas the expression HIREDATE + (2 MONTHS + 14 DAYS) is not. In both of these expressions, the labeled durations are 2 MONTHS and 14 DAYS.

date duration
A number of years, months, and days, expressed as a DECIMAL(8,0) number. To be properly interpreted, the number must have the format *yyyymmdd.*, where *yyyy* represents the number of years, *mm* the number of months, and *dd* the number of days. (The period in the format indicates a DECIMAL data type.) The result of subtracting one date value from another, as in the expression HIREDATE - BRTHDATE, is a date duration.

time duration
A number of hours, minutes, and seconds, expressed as a DECIMAL(6,0) number. To be properly interpreted, the number must have the format *hhmmss.*, where *hh* represents the number of hours, *mm* the number of minutes, and *ss* the number of seconds. (The period in the format indicates a DECIMAL data type.) The result of subtracting one time value from another is a time duration.

timestamp duration
A number of years, months, days, hours, minutes, seconds, and fractional seconds, expressed as a DECIMAL(14+*s*,*s*) number, where *s* is the number of digits of fractional seconds ranging from 0 to 12. To be properly interpreted, the number must have the format *yyyymmddhhmmss.nnnnnnnnnnnn*, where *yyyy*, *mm*, *dd*, *hh*, *mm*, *ss*, and *nnnnnnnnnnnn* represent, respectively, the number of years, months, days, hours, minutes, seconds, and fractional seconds. The result of subtracting one timestamp value from another is a timestamp duration, with scale that matches the maximum timestamp precision of the timestamp operands.

Datetime arithmetic in SQL

The only arithmetic operations that can be performed on datetime values are addition and subtraction. If a datetime value is the operand of addition, the other operand must be a duration. The specific rules governing the use of the addition operator with datetime values follow.

- If one operand is a date, the other operand must be a date duration or labeled duration of YEARS, MONTHS, or DAYS.
- If one operand is a time, the other operand must be a time duration or a labeled duration of HOURS, MINUTES, or SECONDS.
- If one operand is a timestamp, the other operand must be a duration. Any type of duration is valid.
- Neither operand of the addition operator can be a parameter marker.

The rules for the use of the subtraction operator on datetime values are not the same as those for addition because a datetime value cannot be subtracted from a duration, and because the operation of subtracting two datetime values is not the same as the operation of subtracting a duration from a datetime value. The specific rules governing the use of the subtraction operator with datetime values follow.

- If the first operand is a timestamp, the second operand must be a date, a timestamp, a string representation of a date, a string representation of a timestamp, or a duration. If the second operand is a string representation of a timestamp, it is implicitly converted to a timestamp with the same precision of the first operand.
- If the second operand is a timestamp, the first operand must be a date, a timestamp, a string representation of a date, or a string representation of a timestamp. If the first operand is a string representation of a timestamp, it is implicitly converted to a timestamp with the same precision of the second operand.
- If the first operand is a date, the second operand must be a date, a date duration, a string representation of a date, or a labeled duration of YEARS, MONTHS, or DAYS.
- If the second operand is a date, the first operand must be a date, or a string representation of a date.
- If the first operand is a time, the second operand must be a time, a time duration, a string representation of a time, or a labeled duration of HOURS, MINUTES, or SECONDS.
- If the second operand is a time, the first operand must be a time, or string representation of a time.
- Neither operand of the subtraction operator can be a parameter marker.

Date arithmetic

Dates can be subtracted, incremented, or decremented.

- The result of subtracting one date (DATE2) from another (DATE1) is a date duration that specifies the number of years, months, and days between the two dates. The data type of the result is DECIMAL(8,0). If DATE1 is greater than or equal to DATE2, DATE2 is subtracted from DATE1. If DATE1 is less than DATE2, however, DATE1 is subtracted from DATE2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation result = DATE1 - DATE2.

```
If DAY(DATE2) <= DAY(DATE1)
then DAY(RESULT) = DAY(DATE1) - DAY(DATE2).
```

```
If DAY(DATE2) > DAY(DATE1)
then DAY(RESULT) = N + DAY(DATE1) - DAY(DATE2)
where N = the last day of MONTH(DATE2).
MONTH(DATE2) is then incremented by 1.
```

```
If MONTH(DATE2) <= MONTH(DATE1)
then MONTH(RESULT) = MONTH(DATE1) - MONTH(DATE2).
```

```
If MONTH(DATE2) > MONTH(DATE1)
then MONTH(RESULT) = 12 + MONTH(DATE1) - MONTH(DATE2).
YEAR(DATE2) is then incremented by 1.
```

```
YEAR(RESULT) = YEAR(DATE1) - YEAR(DATE2).
```

For example, the result of DATE('3/15/2000') - '12/31/1999' is 00000215. (or, a duration of 0 years, 2 months, and 15 days).

- The result of adding a duration to a date, or of subtracting a duration from a date, is itself a date. (For the purposes of this operation, a month denotes the equivalent of a calendar page. Adding months to a date, then, is like turning the pages of a calendar, starting with the page on which the date appears.) The result must fall between the dates January 1, 0001 and December 31, 9999 inclusive.

  If a duration of years is added or subtracted, only the year portion of the date is affected. The month is unchanged, as is the day unless the result would be February 29 of a non-leap-year. In this case, the day is changed to 28, and a warning indicator in the SQLCA is set to indicate the adjustment.

  Similarly, if a duration of months is added or subtracted, only months and, if necessary, years are affected. The day portion of the date is unchanged unless the result would be invalid (September 31, for example). In this case, the day is set to the last day of the month, and a warning indicator in the SQLCA is set to indicate the adjustment.

  Adding or subtracting a duration of days will, of course, affect the day portion of the date, and potentially the month and year.

  Date durations, whether positive or negative, may also be added to and subtracted from dates. As with labeled durations, the result is a valid date, and a warning indicator is set in the SQLCA whenever an end-of-month adjustment is necessary.

  When a positive date duration is added to a date, or a negative date duration is subtracted from a date, the date is incremented by the specified number of years, months, and days, in that order. Thus, DATE1 + X, where X is a positive DECIMAL(8,0) number, is equivalent to the expression:

```
DATE1 + YEAR(X) YEARS + MONTH(X) MONTHS + DAY(X) DAYS.
```

  When a positive date duration is subtracted from a date, or a negative date duration is added to a date, the date is decremented by the specified number of days, months, and years, in that order. Thus, DATE1 - X, where X is a positive DECIMAL(8,0) number, is equivalent to the expression:

```
DATE1 - DAY(X) DAYS - MONTH(X) MONTHS - YEAR(X) YEARS.
```

  When adding durations to dates, adding one month to a given date gives the same date one month later unless that date does not exist in the later month. In that case, the date is set to that of the last day of the later month. For example, January 28 plus one month gives February 28; and one month added to January 29, 30, or 31 results in either February 28 or, for a leap year, February 29.

  Note: If one or more months is added to a given date and then the same number of months is subtracted from the result, the final date is not necessarily the same as the original date.

Time arithmetic

Times can be subtracted, incremented, or decremented.

- The result of subtracting one time (TIME2) from another (TIME1) is a time duration that specifies the number of hours, minutes, and seconds between the two times. The data type of the result is DECIMAL(6,0).

  If TIME1 is greater than or equal to TIME2, TIME2 is subtracted from TIME1.

  If TIME1 is less than TIME2, however, TIME1 is subtracted from TIME2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation result = TIME1 - TIME2.

```
If SECOND(TIME2) <= SECOND(TIME1)
then SECOND(RESULT) = SECOND(TIME1) - SECOND(TIME2).
```

```
If SECOND(TIME2) > SECOND(TIME1)
then SECOND(RESULT) = 60 + SECOND(TIME1) - SECOND(TIME2).
MINUTE(TIME2) is then incremented by 1.
```

```
If MINUTE(TIME2) <= MINUTE(TIME1)
then MINUTE(RESULT) = MINUTE(TIME1) - MINUTE(TIME2).
```

```
If MINUTE(TIME1) > MINUTE(TIME1)
then MINUTE(RESULT) = 60 + MINUTE(TIME1) - MINUTE(TIME2).
HOUR(TIME2) is then incremented by 1.
```

```
HOUR(RESULT) = HOUR(TIME1) - HOUR(TIME2).
```

For example, the result of TIME('11:02:26') - '00:32:56' is 102930. (a duration of 10 hours, 29 minutes, and 30 seconds).

- The result of adding a duration to a time, or of subtracting a duration from a time, is itself a time. Any overflow or underflow of hours is discarded, thereby ensuring that the result is always a time. If a duration of hours is added or subtracted, only the hours portion of the time is affected. The minutes and seconds are unchanged.

  Similarly, if a duration of minutes is added or subtracted, only minutes and, if necessary, hours are affected. The seconds portion of the time is unchanged.

  Adding or subtracting a duration of seconds will, of course, affect the seconds portion of the time, and potentially the minutes and hours.

  Time durations, whether positive or negative, also can be added to and subtracted from times. The result is a time that has been incremented or decremented by the specified number of hours, minutes, and seconds, in that order. TIME1 + X, where "X" is a DECIMAL(6,0) number, is equivalent to the expression:

  ```
  TIME1 + HOUR(X) HOURS + MINUTE(X) MINUTES + SECOND(X) SECONDS
  ```

  When subtracting a labeled duration of SECOND or SECONDS with a value that includes fractions of a second, the subtraction is performed as if the time value has up to 12 fractional second digits but the result is returned with the fractional seconds truncated.

  Note: Although the time '24:00:00' is accepted as a valid time, it is never returned as the result of time addition or subtraction, even if the duration operand is zero (for example, time('24:00:00')±0 seconds = '00:00:00').

Timestamp arithmetic

Timestamps can be subtracted, incremented, or decremented.

- The result of subtracting one timestamp (TS2) from another (TS1) is a timestamp duration that specifies the number of years, months, days, hours, minutes, seconds, and fractional seconds between the two timestamps. The data type of the result is DECIMAL(14+*s*,*s*), where *s* is the maximum timestamp precision of TS1 and TS2.

  If TS1 is greater than or equal to TS2, TS2 is subtracted from TS1. If TS1 is less than TS2, however, TS1 is subtracted from TS2 and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation result = TS1 - TS2:

  ```
  If SECOND(TS2,s) <= SECOND(TS1,s)
  then SECOND(RESULT,s) = SECOND(TS1,s) -
  SECOND(TS2,s).
  ```

  ```
  If SECOND(TS2,s) > SECOND(TS1,s)
  then SECOND(RESULT,s) = 60 +
  SECOND(TS1,s) - SECOND(TS2,s).
  MINUTE(TS2) is then incremented by 1.
  ```

The minutes part of the timestamps are subtracted as specified in the rules for subtracting times.

```
If HOUR(TS2) <= HOUR(TS1)
then HOUR(RESULT) = HOUR(TS1) - HOUR(TS2).
```

```
If HOUR(TS2) > HOUR(TS1)
then HOUR(RESULT) = 24 + HOUR(TS1) - HOUR(TS2)
and DAY(TS2) is incremented by 1.
```

The date part of the timestamps is subtracted as specified in the rules for subtracting dates.

- The result of subtracting a date (D1) from a timestamp (TS1) is the same as subtracting TIMESTAMP(D1) from TS1. Similarly, the result of subtracting one timestamp (TS1) from a date (D2) is the same as subtracting TS1 from TIMESTAMP(D2).
- The result of adding a duration to a timestamp, or of subtracting a duration from a timestamp is itself a timestamp.The precision of the result timestamp matches the precision of the timestamp operand. The date arithmetic portion is performed as previously defined, except that an overflow or underflow of hours is carried into the date part of the result, which must be within the range of valid dates. The time arithmetic portion is similar to time arithmetic except that it also considers the fractional seconds included in the duration. Thus, subtracting a duration, *X*, from a timestamp, TIMESTAMP1, where *X* is a DECIMAL(14+s,s) number, is equivalent to the expression:

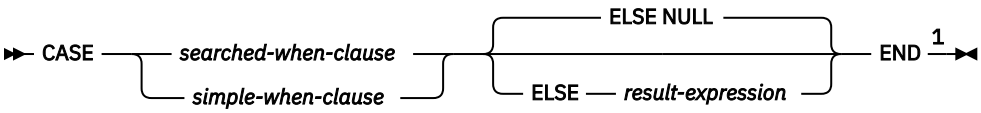
```
TIMESTAMP1 - YEAR(X)  YEARS - MONTH(X)  MONTHS - DAY(X)  DAYS
           - HOUR(X)  HOURS - MINUTE(X)  MINUTES - SECOND(X, s)  SECONDS
```

When subtracting a duration with non-zero scale or a labeled duration of SECOND or SECONDS with a value that includes fractions of a second, the subtraction is performed as if the timestamp value has up to 12 fractional second digits. The resulting value is assigned to a timestamp value with the timestamp precision of the timestamp operand which could result in truncation of fractional second digits.

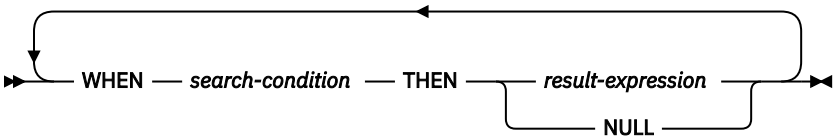
CASE expression

CASE expressions allow an expression to be selected based on the evaluation of one or more conditions.

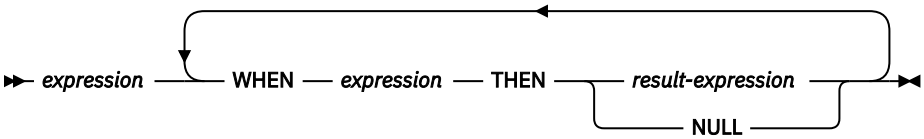
case-expression



searched-when-clause



simple-when-clause



Notes:

[1] If the result type of *result-expression* is a row type, then the syntax represents a *row-case-expression* and can only be used where a *row-expression* is allowed.

In general, the value of the *case-expression* is the value of the *result-expression* following the first (leftmost) case that evaluates to true. If no case evaluates to true and the ELSE keyword is present then the result is the value of the *result-expression* or NULL. If no case evaluates to true and the ELSE keyword

is not present then the result is NULL. Note that when a case evaluates to unknown (because of NULLs), the case is not true and hence is treated the same way as a case that evaluates to false.

If the CASE expression is in a VALUES clause, an IN predicate, a GROUP BY clause, or an ORDER BY clause, the *search-condition* in a *searched-when-clause* cannot be a quantified predicate, IN predicate using a fullselect, or an EXISTS predicate (SQLSTATE 42625).

When using the *simple-when-clause*, the value of the *expression* before the first *WHEN* keyword is tested for equality with the value of the *expression* following the WHEN keyword. The data type of the *expression* before the first *WHEN* keyword must therefore be comparable to the data types of each *expression* following the WHEN keyword(s). The *expression* before the first *WHEN* keyword in a *simple-when-clause* cannot include a function that is not deterministic or has an external action (SQLSTATE 42845).

A *result-expression* is an *expression* following the THEN or ELSE keywords. There must be at least one *result-expression* in the CASE expression (NULL cannot be specified for every case) (SQLSTATE 42625). All result expressions must have compatible data types (SQLSTATE 42804).

Examples

- If the first character of a department number is a division in the organization, then a CASE expression can be used to list the full name of the division to which each employee belongs:

```
SELECT EMPNO, LASTNAME,
  CASE SUBSTR(WORKDEPT,1,1)
    WHEN 'A' THEN 'Administration'
    WHEN 'B' THEN 'Human Resources'
    WHEN 'C' THEN 'Accounting'
    WHEN 'D' THEN 'Design'
    WHEN 'E' THEN 'Operations'
  END
  FROM EMPLOYEE;
```

- The number of years of education are used in the EMPLOYEE table to give the education level. A CASE expression can be used to group these and to show the level of education.

```
SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME,
  CASE
    WHEN EDLEVEL < 15 THEN 'SECONDARY'
    WHEN EDLEVEL < 19 THEN 'COLLEGE'
    ELSE 'POST GRADUATE'
  END
  FROM EMPLOYEE
```

- Another interesting example of CASE statement usage is in protecting from division by 0 errors. For example, the following code finds the employees who earn more than 25% of their income from commission, but who are not fully paid on commission:

```
SELECT EMPNO, WORKDEPT, SALARY+COMM FROM EMPLOYEE
WHERE (CASE WHEN SALARY=0 THEN NULL
  ELSE COMM/SALARY
  END) > 0.25;
```

- The following CASE expressions are the same:

```
SELECT LASTNAME,
  CASE
  WHEN LASTNAME = 'Haas' THEN 'President'
  ...

SELECT LASTNAME,
  CASE LASTNAME
  WHEN 'Haas' THEN 'President'
  ...
```

There are two scalar functions, NULLIF and COALESCE, that are specialized to handle a subset of the functionality provided by CASE. Table 26 on page 152 shows the equivalent expressions using CASE or these functions.

Table 26. Equivalent CASE Expressions

Expression	Equivalent Expression

`CASE`
`  WHEN e1=e2 THEN NULL`
`  ELSE e1`
`END`
	NULLIF(e1,e2)

`CASE`
`  WHEN e1 IS NOT NULL THEN e1`
`  ELSE e2`
`END`
	COALESCE(e1,e2)

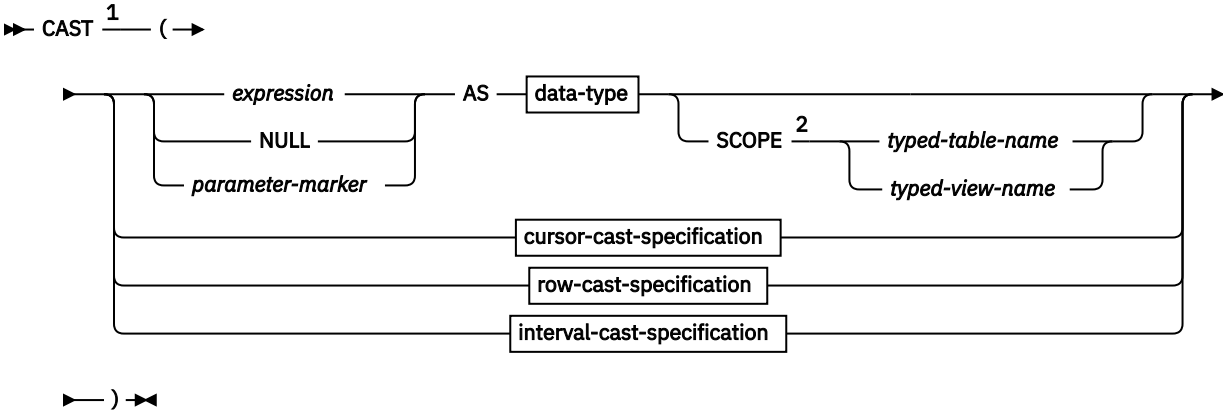
`CASE`
`  WHEN e1 IS NOT NULL THEN e1`
`  ELSE COALESCE(e2,...,eN)`
`END`
	COALESCE(e1,e2,...,eN)

`CASE`
`  WHEN c1=var1 OR (c1 IS NULL AND var1 IS NULL)`
`  THEN 'a'`
`  WHEN c1=var2 OR (c1 IS NULL AND var2 IS NULL)`
`  THEN 'b'`
`  ELSE NULL`
`END`
	DECODE(c1,var1, 'a', var2, 'b')

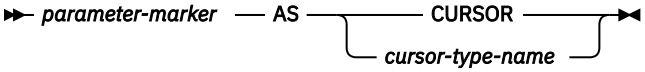
CAST specification

The CAST specification returns the cast operand (the first operand) cast to the type specified by the *data-type*. If the cast is not supported, an error is returned (SQLSTATE 42846).

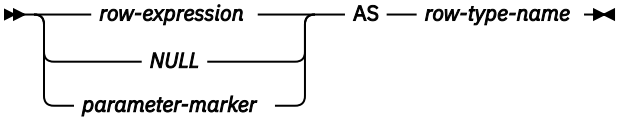
cast-specification



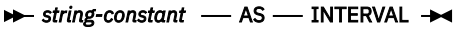
cursor-cast-specification



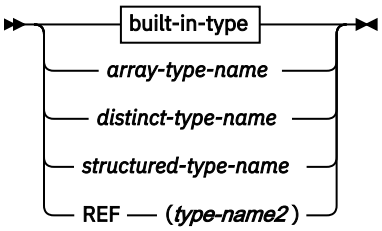
row-cast-specification



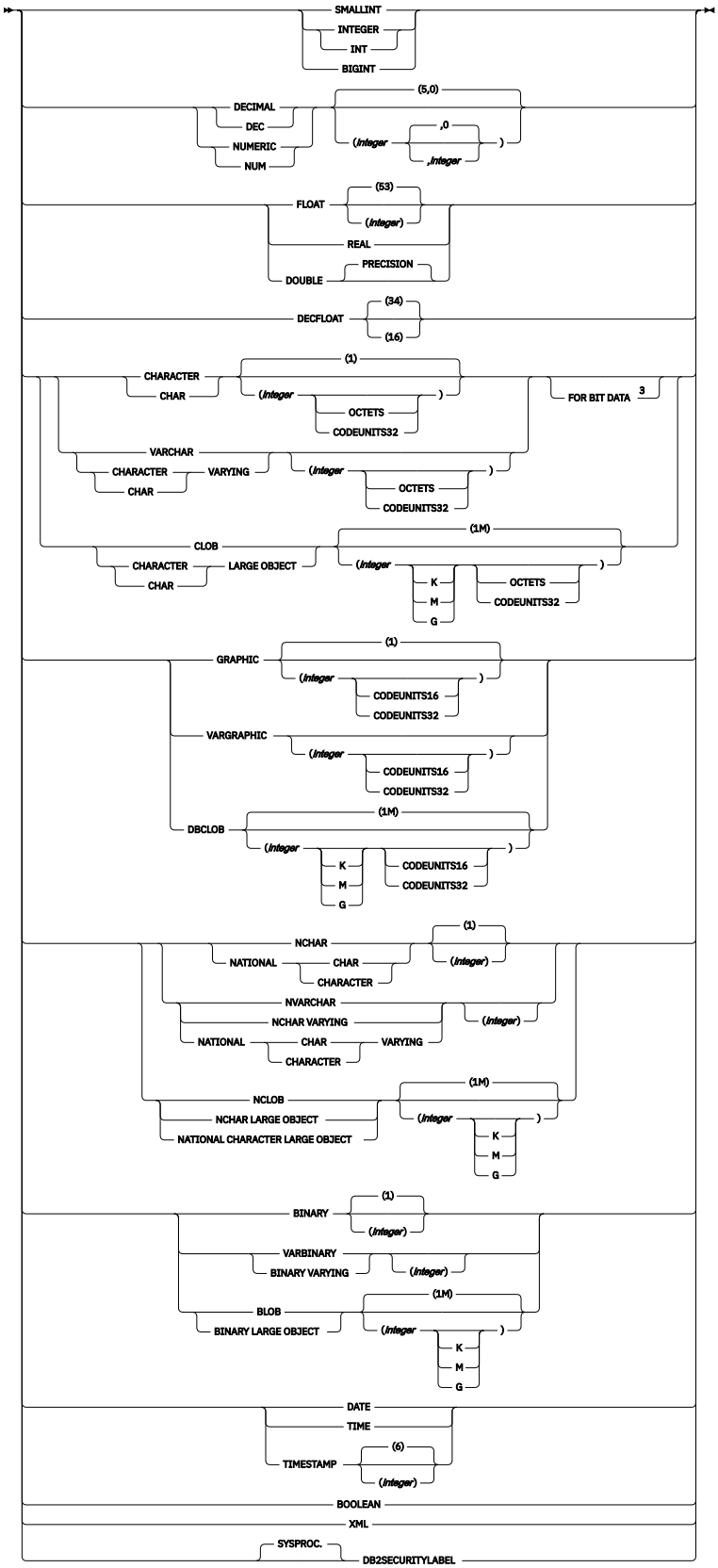
interval-cast-specification



data-type



built-in-type



Notes:

[1] For compatibility purposes, you can use `::` as the type cast operator. For example, the statements `C1::INTEGER` and `cast(C1 as INTEGER)` are equivalent.

[2] The SCOPE clause only applies to the REF data type.

[3] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

expression
If the cast operand is an expression (other than parameter marker or NULL), the result is the argument value converted to the specified target *data-type*.

When casting character strings (other than CLOBs) to a character string with a different length, a warning (SQLSTATE 01004) is returned if truncation of other than trailing blanks occurs. When casting graphic character strings (other than DBCLOBs) to a graphic character string with a different length, a warning (SQLSTATE 01004) is returned if truncation of other than trailing blanks occurs. For BLOB, CLOB and DBCLOB cast operands, the warning is issued if any characters are truncated.

When casting an array, the target data type must be a user-defined array data type (SQLSTATE 42821). The data type of the elements of the array must be the same as the data type of the elements of the target array data type (SQLSTATE 42846). The cardinality of the array must be less than or equal to the maximum cardinality of the target array data type (SQLSTATE 2202F).

NULL
If the cast operand is the keyword NULL, the result is a null value that has the specified *data-type*.

parameter-marker
A parameter marker is normally considered an expression, but is documented separately in this case because it has a special meaning. If the cast operand is a *parameter-marker*, the specified *data-type* is considered a promise that the replacement will be assignable to the specified data type (using store assignment for strings). Such a parameter marker is considered a *typed parameter marker*. Typed parameter markers will be treated like any other typed value for the purpose of function resolution, DESCRIBE of a select list or for column assignment.

cursor-cast-specification
A cast specification used to indicate that a parameter marker is expected to be a cursor type. It can be used wherever an expression is supported in contexts that allow cursor types.

parameter-marker
The cast operand is a parameter marker and is considered a promise that the replacement will be assignable to the specified cursor type.

CURSOR
Specifies the built-in data type CURSOR.

cursor-type-name
Specifies the name of a user-defined cursor type.

row-cast-specification
A cast specification where the input is a row value and the result is a user-defined row type. A *row-cast-specification* is valid only where a *row-expression* is allowed.

row-expression
The data type of *row-expression* must be a variable of row type that is anchored to the definition of a table or view. The data type of *row-expression* must not be a user-defined row type (SQLSTATE 42846).

NULL
Specifies that the cast operand is the null value. The result is a row with the null value for every field of the specified data type.

parameter-marker
The cast operand is a parameter marker and is considered a promise that the replacement will be assignable to the specified *row-type-name*.

row-type-name
Specifies the name of a user-defined row type. The *row-expression* must be castable to *row-type-name* (SQLSTATE 42846).

interval-cast-specification
A cast specification where the input is a character string representation of an interval and the result is a decimal duration. The following statements are equivalent:

```
CAST (string-constant as INTERVAL)
INTERVAL string-constant
```

For more information about possible string-constant values, see “INTERVAL ” on page 376.

data-type
The name of an existing data type. If the type name is not qualified, the SQL path is used to perform data type resolution. A data type that has associated attributes, such as length or precision and scale, should include these attributes when specifying *data-type*.

- CHAR defaults to a length of 1
- BINARY defaults to a length of 1
- DECIMAL defaults to a precision of 5 and a scale of 0
- DECFLOAT defaults to a precision of 34 if not specified

The FOR SBCS DATA clause or the FOR MIXED DATA clause (only one is supported depending on whether or not the database supports the graphic data type) can be used to cast a FOR BIT DATA string to the database code page. Restrictions on the supported data types are based on the specified cast operand.

- For a cast operand that is an *expression*, the supported target data types depend on the data type of the cast operand (source data type). If the length attribute is not specified for a VARCHAR, VARGRAPHIC, NVARCHAR, or VARBINARY data type, the length attribute is determined based on the data type of the first argument using the rules of the corresponding built-in cast function when specified with no length argument.
- For a cast operand that is the keyword NULL, any existing data type can be used. If the length attribute is not specified for a VARCHAR, VARGRAPHIC, NVARCHAR, or VARBINARY data type, a length attribute of 1 is used.
- For a cast operand that is a parameter marker, the target data type can be any existing data type. If the data type is a user-defined distinct type, the application using the parameter marker will use the source data type of the user-defined distinct type. If the data type is a user-defined structured type, the application using the parameter marker will use the input parameter type of the TO SQL transform function for the user-defined structured type. If the length attribute is not specified for a VARCHAR, VARGRAPHIC, NVARCHAR, or VARBINARY data type, a length attribute of 254 is used.

If the data type is a distinct type defined with data type constraints, the data type constraints are applied and the constraints must evaluate to true or unknown otherwise an error is returned (SQLSTATE 23528).

built-in-type
See "CREATE TABLE" for the description of built-in data types.

SCOPE
When the data type is a reference type, a scope may be defined that identifies the target table or target view of the reference.

typed-table-name
The name of a typed table. The table must already exist (SQLSTATE 42704). The cast must be to *data-type* REF(*S*), where *S* is the type of *typed-table-name* (SQLSTATE 428DM).

typed-view-name
The name of a typed view. The view must exist or have the same name as the view being created that includes the cast as part of the view definition (SQLSTATE 42704). The cast must be to *data-type* REF(*S*), where *S* is the type of *typed-view-name* (SQLSTATE 428DM).

When numeric data is cast to character data, the result data type is a fixed-length character string. When character data is cast to numeric data, the result data type depends on the type of number specified. For example, if cast to integer, it becomes a large integer.

Examples

- An application is only interested in the integer portion of the SALARY (defined as decimal(9,2)) from the EMPLOYEE table. The following query, including the employee number and the integer value of SALARY, could be prepared.

```
SELECT EMPNO, CAST(SALARY AS INTEGER) FROM EMPLOYEE
```

- Assume the existence of a distinct type called T_AGE that is defined on SMALLINT and used to create column AGE in PERSONNEL table. Also assume the existence of a distinct type called R_YEAR that is defined on INTEGER and used to create column RETIRE_YEAR in PERSONNEL table. The following update statement could be prepared.

```
UPDATE PERSONNEL SET RETIRE_YEAR =?
  WHERE AGE = CAST( ? AS T_AGE)
```

  The first parameter is an untyped parameter marker that would have a data type of R_YEAR, although the application will use an integer for this parameter marker. This does not require the explicit CAST specification because it is an assignment.

  The second parameter marker is a typed parameter marker that is cast as a distinct type T_AGE. This satisfies the requirement that the comparison must be performed with compatible data types. The application will use the source data type (which is SMALLINT) for processing this parameter marker.

  Successful processing of this statement assumes that the SQL path includes the schema name of the schema (or schemas) where the two distinct types are defined.

- An application supplies a value that is a series of bits, for example an audio stream, and it should not undergo code page conversion before being used in an SQL statement. The application could use the following CAST:

```
CAST( ? AS VARCHAR(10000) FOR BIT DATA)
```

- Assume that an array type and a table have been created as follows:

```
CREATE TYPE PHONELIST AS DECIMAL(10, 0) ARRAY[5]

CREATE TABLE EMP_PHONES
  (ID          INTEGER,
   PHONENUMBER DECIMAL(10,0) )
```

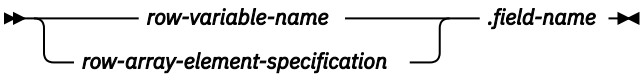
  The following procedure returns an array with the phone numbers for the employee with ID 1775. If there are more than five phone numbers for this employee, an error is returned (SQLSTATE 2202F).

```
CREATE PROCEDURE GET_PHONES(OUT EPHONES PHONELIST)
BEGIN
  SELECT CAST(ARRAY_AGG(PHONENUMBER) AS PHONELIST)
  INTO EPHONES
  FROM EMP_PHONES
  WHERE ID = 1775;
END
```

Field reference

A field of a row type is referenced by using the field name qualified by a variable that returns a row type which includes a field with that field name, or an array element specification that returns a row type which includes a field with that field name.

field-reference



row-variable-name
The name of a variable with a data type that is a row type.

row-array-element-specification
An *array-element-specification* where the data type of the array element is a row type.

field-name
The name of a field within the row type.

Examples

1. To reference a field of a simple row type:

```
CREATE OR REPLACE  TYPE REC_TYPE AS ROW
(
   ID INTEGER,
   NAME VARCHAR(10)
)@

BEGIN
     DECLARE NAME VARCHAR(10);
     DECLARE TMPREC REC_TYPE;
     SET NAME = TMPREC.NAME;
END @
```

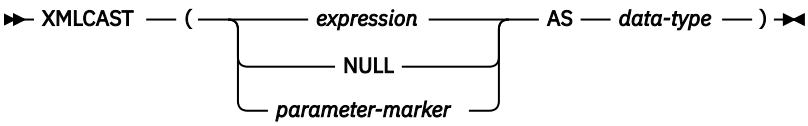
2. To reference a field of a nested row type, you must assign the inner row to a temporary variable before a field of this inner row can be referenced:

```
CREATE OR REPLACE TYPE INNER_REC_TYPE AS ROW
(
  ID INTEGER,
  NAME VARCHAR(10)
)@
CREATE OR REPLACE  TYPE REC_TYPE AS ROW
(
  INNER_REC INNER_REC_TYPE
)@
CREATE OR REPLACE TYPE ARRAY_TYPE AS REC_TYPE ARRAY[INTEGER]@

BEGIN
  DECLARE TMPRECORD INNER_REC_TYPE;
  DECLARE TMPARRAY ARRAY_TYPE;
  DECLARE NAME VARCHAR(10);

  SET TMPRECORD = TMPARRAY[1].INNER_REC;
  SET NAME = TMPRECORD.NAME;
  END @
```

XMLCAST specification

The XMLCAST specification returns the cast operand (the first operand) cast to the type specified by the data type. XMLCAST supports casts involving XML values, including conversions between non-XML data types and the XML data type. If the cast is not supported, an error is returned (SQLSTATE 22003).

xmlcast-specification



expression
If the cast operand is an expression (other than a parameter marker or NULL), the result is the argument value converted to the specified target data type. The expression or the target data type must be the XML data type (SQLSTATE 42846).

NULL
If the cast operand is the keyword NULL, the target data type must be the XML data type (SQLSTATE 42846). The result is a null XML value.

parameter-marker
If the cast operand is a parameter marker, the target data type must be XML (SQLSTATE 42846). A parameter marker is normally considered to be an expression, but is documented separately in this

case because it has special meaning. If the cast operand is a parameter marker, the specified data type is considered to be a promise that the replacement will be assignable to the specified (XML) data type (using store assignment). Such a parameter marker is considered to be a typed parameter marker, which is treated like any other typed value for the purpose of function resolution, a describe operation on a select list, or column assignment.

data-type
The name of an existing SQL data type. If the name is not qualified, the SQL path is used to perform data type resolution. If a data type has associated attributes, such as length or precision and scale, these attributes should be included when specifying a value for *data-type*. CHAR defaults to a length of 1, and DECIMAL defaults to a precision of 5 and a scale of 0 if not specified. Restrictions on the supported data types are based on the specified cast operand.

- For a cast operand that is an expression, the supported target data types depend on the data type of the cast operand (source data type).
- For a cast operand that is the keyword NULL, the target data type must be XML.
- For a cast operand that is a parameter marker, the target data type must be XML.

Note: Support in non-Unicode databases: When XMLCAST is used to convert an XML value to an SQL data type, code page conversion is performed. The encoding of the cast expression is converted from UTF-8 to the database code page. Characters in the original expression that are not present in the database code page are replaced by substitution characters as a result of this conversion.

Examples

- Create a null XML value.

```
XMLCAST(NULL AS XML)
```

- Convert a value extracted from an XMLQUERY expression into an INTEGER:

```
XMLCAST(XMLQUERY('$m/PRODUCT/QUANTITY'
  PASSING xmlcol AS "m") AS INTEGER)
```

- Convert a value extracted from an XMLQUERY expression into a varying-length character string:

```
XMLCAST(XMLQUERY('$m/PRODUCT/ADD-TIMESTAMP'
  PASSING xmlcol AS "m") AS VARCHAR(30))
```

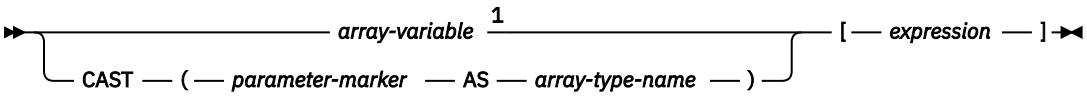
- Convert a value extracted from an SQL scalar subquery into an XML value.

```
XMLCAST((SELECT quantity FROM product AS p
  WHERE p.id = 1077) AS XML)
```

ARRAY element specification

The ARRAY element specification returns the element from an array specified by *expression*. If any argument to *expression* is null, the null value is returned.

array-element-specification



Notes:

[1] If the data type of the elements in the array is a row type, then the syntax represents an array-element-specification with a row data type and can only be used where a *row-expression* is allowed.

array-variable
An SQL variable, SQL parameter, or global variable of an array type.

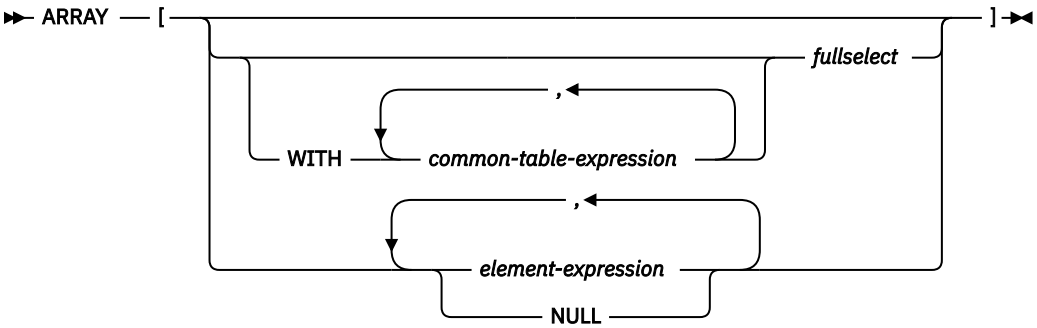
CAST (*parameter-marker* AS *array-type-name*)
A parameter marker is normally considered to be an expression, but in this case it must explicitly be cast to a user-defined array data type.

[*expression*]
Specifies the array index of the element that is to be extracted from the array. The array index of an ordinary array must be assignable to INTEGER (SQLSTATE 428H1). The value must be between 1 and the cardinality of the array (SQLSTATE 2202E). The array index of an associative array must be assignable to the index data type (SQLSTATE 428H1).

Array constructor

An array constructor is a language element that can be used to define and construct an array data type value within a valid context.

Syntax



Authorization

No specific authorizations are required to reference an array constructor within an SQL statement, however for the statement execution to be successful all other authorization requirements for the statement must be satisfied.

Description

WITH *common-table-expression*
Defines a common table expression for use with the following fullselect.

fullselect
A fullselect that returns a single column. The values that are returned by the fullselect for each row are the elements of the array. The cardinality of the array is equal to the number of rows that are returned by the fullselect. If the fullselect includes an *order-by-clause*, the order determines the order in which row values are assigned to elements of the array. If no *order-by-clause* is specified, the order in which row values are assigned to elements of the array is not deterministic.

element-expression
An expression that defines the value of an element in the array. The cardinality of the array is equal to the number of element expressions. The first *element-expression* is assigned to the array element with array index 1. The second *element-expression* is assigned to the array element with array index 2 and so on. Every *element-expression* must have a compatible data type with every other *element-expression* and the base type of the array is determined using the "Rules for result data types" topic.

NULL
Specifies the null value.

If no value is specified within the brackets, the result is an empty array.

Rules

- The base type of the *array-constructor*, as derived from the *element-expression*s or the *fullselect*, must be assignable to the base type of the target array (SQLSTATE 42821).
- The number of elements in the *array-constructor* must not exceed the maximum cardinality of the target array variable (SQLSTATE 2202F).

Notes

- An array constructor can be used to define only an ordinary array with elements that are not a row type. An array constructor cannot be used to define an associative array or an ordinary array with elements that are a row type. Such arrays can only be constructed by assigning the individual elements.

Examples

Example 1: Set the array variable RECENT_CALLS of array type PHONENUMBERS to an array of fixed numbers.

```
SET RECENT_CALLS = ARRAY[9055553907, 4165554213, 4085553678]
```

Example 2: Set the array variable DEPT_PHONES of array type PHONENUMBERS to an array of phone numbers retrieved from the DEPARTMENT_INFO table.

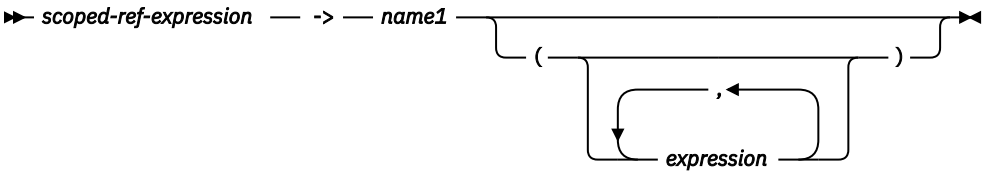
```
SET DEPT_PHONES = ARRAY[SELECT DECIMAL(AREA_CODE CONCAT '555' CONCAT EXTENSION,16)
                                  FROM DEPARTMENT_INFO
                                  WHERE DEPTID = 624]
```

Dereference operation

The scope of the scoped reference expression is a table or view called the *target* table or view.

The scoped reference expression identifies a *target row*. The *target row* is the row in the target table or view (or in one of its subtables or subviews) whose object identifier (OID) column value matches the reference expression. The dereference operation can be used to access a column of the target row, or to invoke a method, using the target row as the subject of the method. The result of a dereference operation can always be null. The dereference operation takes precedence over all other operators.

dereference-operation



scoped-ref-expression
An expression that is a reference type that has a scope (SQLSTATE 428DT). If the expression is a host variable, parameter marker or other unscoped reference type value, a CAST specification with a SCOPE clause is required to give the reference a scope.

name1
Specifies an unqualified identifier.

If no parentheses follow *name1*, and *name1* matches the name of an attribute of the target type, then the value of the dereference operation is the value of the named column in the target row. In this case, the data type of the column (made nullable) determines the result type of the dereference operation. If no target row exists whose object identifier matches the reference expression, then the result of the dereference operation is null. If the dereference operation is used in a select list and is not included as part of an expression, *name1* becomes the result column name.

If parentheses follow *name1*, or if *name1* does not match the name of an attribute of the target type, then the dereference operation is treated as a method invocation. The name of the invoked method is *name1*. The subject of the method is the target row, considered as an instance of its structured type. If no target row exists whose object identifier matches the reference expression, the subject of the method is a null value of the target type. The expressions inside parentheses, if any, provide the remaining parameters of the method invocation. The normal process is used for resolution of the method invocation. The result type of the selected method (made nullable) determines the result type of the dereference operation.

The authorization ID of the statement that uses a dereference operation must have SELECT privilege on the target table of the *scoped-ref-expression* (SQLSTATE 42501).

A dereference operation can never modify values in the database. If a dereference operation is used to invoke a mutator method, the mutator method modifies a copy of the target row and returns the copy, leaving the database unchanged.

Examples

- Assume the existence of an EMPLOYEE table that contains a column called DEPTREF which is a reference type scoped to a typed table based on a type that includes the attribute DEPTNAME. The values of DEPTREF in the table EMPLOYEE should correspond to the OID column values in the target table of DEPTREF column.

```
SELECT EMPNO, DEPTREF->DEPTNAME
  FROM EMPLOYEE
```

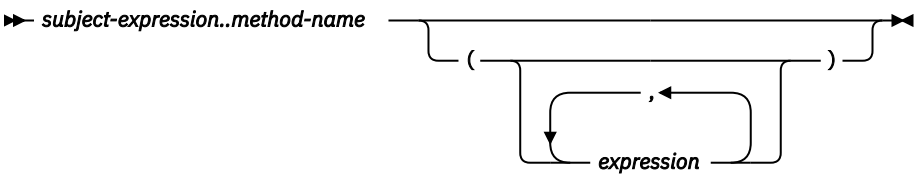
- Using the same tables as in the previous example, use a dereference operation to invoke a method named BUDGET, with the target row as subject parameter, and '1997' as an additional parameter.

```
SELECT EMPNO, DEPTREF->BUDGET('1997') AS DEPTBUDGET97
  FROM EMPLOYEE
```

Method invocation

Both system-generated observer and mutator methods, as well as user-defined methods are invoked using the double-dot operator.

method-invocation



subject-expression
An expression with a static result type that is a user-defined structured type.

method-name
The unqualified name of a method. The static type of *subject-expression* or one of its supertypes must include a method with the specified name.

(*expression*,...)
The arguments of *method-name* are specified within parentheses. Empty parentheses can be used to indicate that there are no arguments. The *method-name* and the data types of the specified argument expressions are used to resolve to the specific method, based on the static type of *subject-expression*.

The double-dot operator used for method invocation is a high precedence left to right infix operator. For example, the following two expressions are equivalent:

```
a..b..c + x..y..z
```

and

```
((a..b)..c) + ((x..y)..z)
```

If a method has no parameters other than its subject, it can be invoked with or without parentheses. For example, the following two expressions are equivalent:

```
point1..x
point1..x()
```

Null subjects in method calls are handled as follows:

- If a system-generated mutator method is invoked with a null subject, an error results (SQLSTATE 2202D)
- If any method other than a system-generated mutator is invoked with a null subject, the method is not executed, and its result is null. This rule includes user-defined methods with SELF AS RESULT.

When a database object (a package, view, or trigger, for example) is created, the best fit method that exists for each of its method invocations is found.

Note: Methods of types defined WITH FUNCTION ACCESS can also be invoked using the regular function notation. Function resolution considers all functions, as well as methods with function access as candidate functions. However, functions cannot be invoked using method invocation. Method resolution considers all methods and does not consider functions as candidate methods. Failure to resolve to an appropriate function or method results in an error (SQLSTATE 42884).

Example

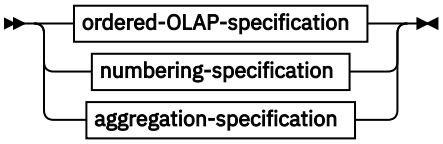
- Use the double-dot operator to invoke a method called AREA. Assume the existence of a table called RINGS, with a column CIRCLE_COL of structured type CIRCLE. Also, assume that the method AREA has been defined previously for the CIRCLE type as AREA() RETURNS DOUBLE.

```
SELECT CIRCLE_COL..AREA() FROM RINGS
```

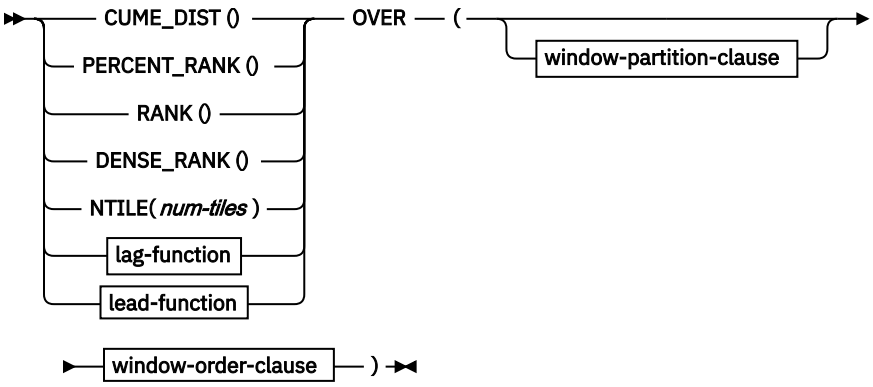
OLAP specification

On-Line Analytical Processing (OLAP) functions provide the ability to return ranking, row numbering and existing aggregate function information as a scalar value in a query result.

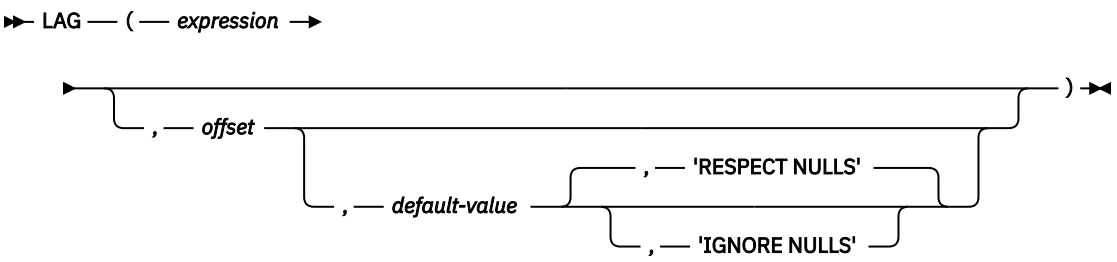
OLAP-specification



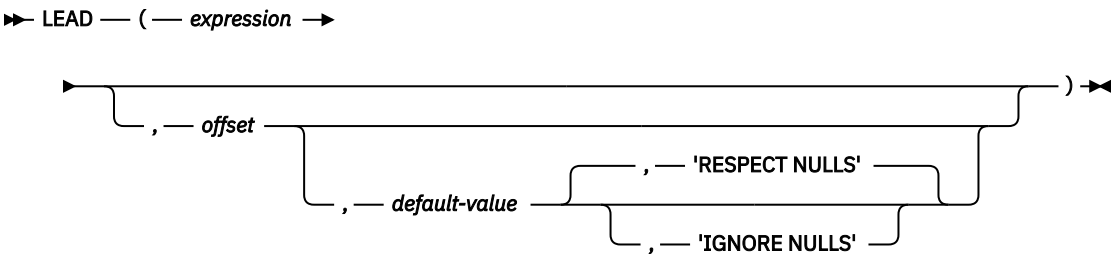
ordered-OLAP-specification



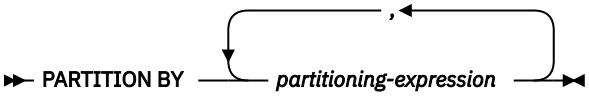
lag-function



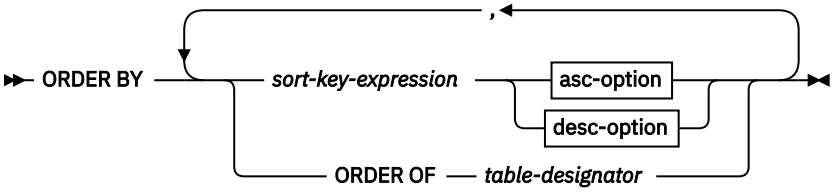
lead-function



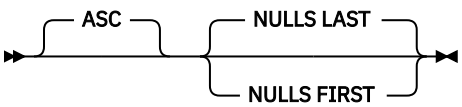
window-partition-clause



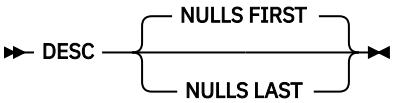
window-order-clause



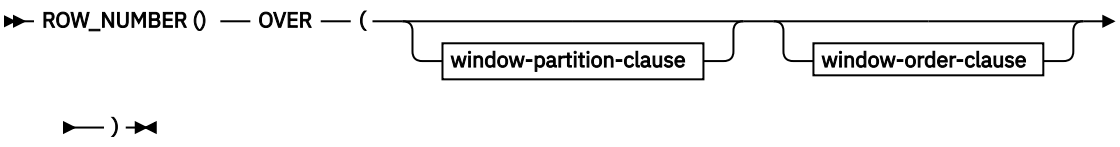
asc-option



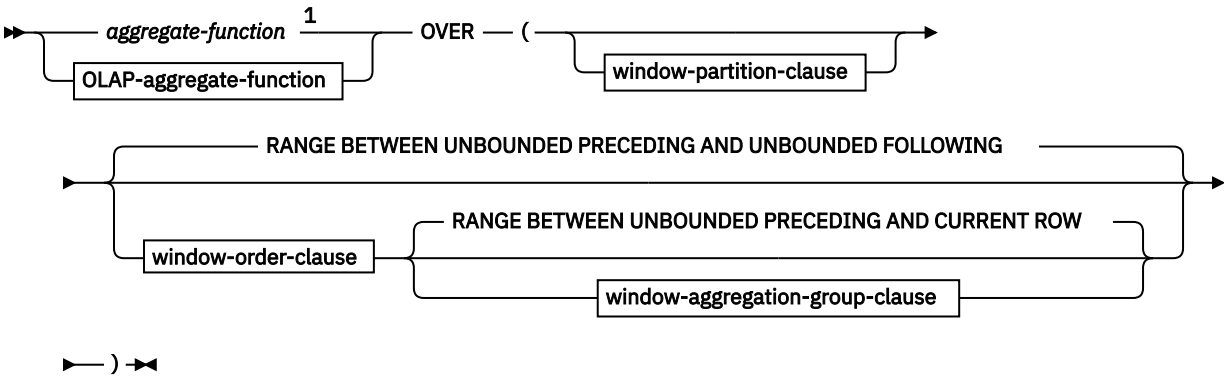
desc-option



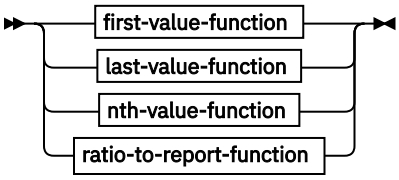
numbering-specification



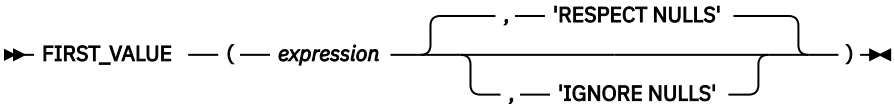
aggregation-specification



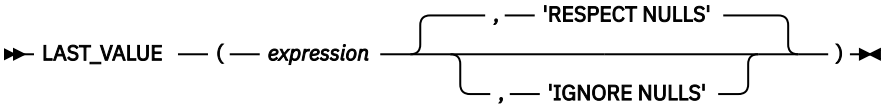
OLAP-aggregate-function



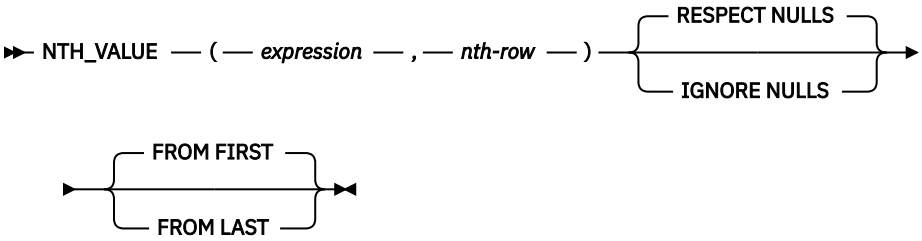
first-value-function



last-value-function

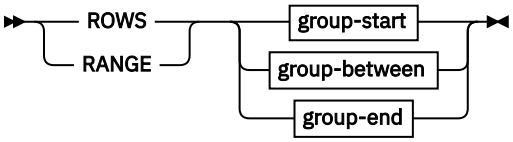


nth-value-function

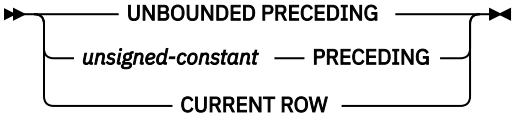


ratio-to-report-function

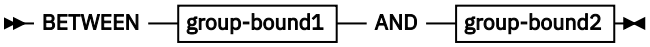
window-aggregation-group-clause



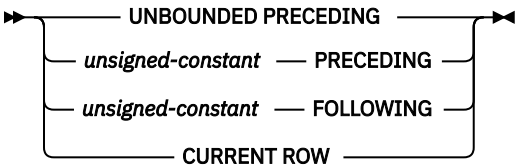
group-start



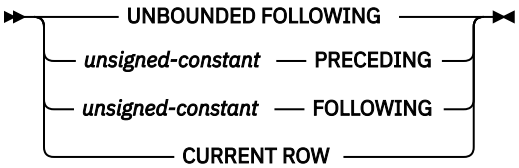
group-between



group-bound1



group-bound2



group-end

Notes:

[1] ARRAY_AGG, CUME_DIST, and PERCENT_RANK are not supported as an aggregate function in *aggregation-specification* (SQLSTATE 42887).

An OLAP function can be included in expressions in a select-list or the ORDER BY clause of a select-statement (SQLSTATE 42903). An OLAP function cannot be used within an argument to an XMLQUERY or XMLEXISTS expression (SQLSTATE 42903). An OLAP function cannot be used as an argument of an aggregate function (SQLSTATE 42607). The query result to which the OLAP function is applied is the result table of the innermost subselect that includes the OLAP function.

When specifying an OLAP function, a window is specified that defines the rows over which the function is applied, and in what order. When used with an aggregate function, the applicable rows can be further refined, relative to the current row, as either a range or a number of rows preceding and following the current row. For example, within a partition by month, an average can be calculated over the previous three month period.

The **CUME_DIST** function is a distribution function that returns a cumulative distribution of a row within an OLAP window, expressed as a value between 0.0 - 1.0. The result is computed as follows:

The number of rows preceding or peer with the current row in the OLAP window, divided by the number of rows in the OLAP window.

The data type of the result is DECFLOAT(34). The result cannot be NULL.

The **PERCENT_RANK** function is a distribution function that returns a relative percentile rank of a row within an OLAP window, expressed as a value between 0.0 - 1.0. When the number of rows in the OLAP window is greater than 1, the result is computed as follows:

The RANK of the current row in the OLAP window minus 1 divided by the number of rows in the OLAP window minus 1.

Otherwise, the result is 0.0.

The data type of the result is DECFLOAT(34). The result cannot be NULL.

The ranking function computes the ordinal rank of a row within the window. Rows that are not distinct with respect to the ordering within their window are assigned the same rank. The results of ranking may be defined with or without gaps in the numbers resulting from duplicate values.

If **RANK** is specified, the rank of a row is defined as 1 plus the number of rows that strictly precede the row. Thus, if two or more rows are not distinct with respect to the ordering, then there will be one or more gaps in the sequential rank numbering.

If **DENSE_RANK (or DENSERANK)** is specified, the rank of a row is defined as 1 plus the number of preceding rows that are distinct with respect to the ordering. Therefore, there will be no gaps in the sequential rank numbering.

The **ROW_NUMBER (or ROWNUMBER)** function computes the sequential row number of the row within the window defined by the ordering, starting with 1 for the first row. If the ORDER BY clause is not specified in the window, the row numbers are assigned to the rows in arbitrary order, as returned by the subselect (not according to any ORDER BY clause in the select-statement).

If the fetch-clause is used along with the ROW_NUMBER function, the row numbers might not be displayed in order. The fetch-clause is applied after the result set (including any ROW_NUMBER assignments) is generated; therefore, if the row number order is not the same as the order of the result set, some assigned numbers might be missing from the sequence.

The data type of the result of RANK, DENSE_RANK or ROW_NUMBER is BIGINT. The result cannot be null.

The **NTILE** function returns the quantile rank of a row.

num-tiles
An expression that specifies the number of quantiles. The expression must return a value that is a built-in numeric data type, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If the expression is not a SMALLINT, INTEGER, or BIGINT, it is cast to BIGINT before the function is evaluated. The value must be greater than 0 (SQLSTATE 22014). The expression must be a constant, a variable, or a cast of a constant or variable (SQLSTATE 42601).

The data type of the result of NTILE is the same data type as the data type of *num-tiles* after any implicit casting. If the argument can be null, the result can be null. If the argument is null, the result is the null value.

The NTILE function computes the quantile rank of a row by dividing the ordered rows within the OLAP window into *num-tiles* quantiles and returns a value between 1 and MIN(*n*, *num-tiles*), where *n* is the number of rows within the OLAP window. If *n* is evenly divisible by *num-tiles*, the rows in the OLAP window are grouped into *num-tiles* quantiles, each containing (*n* / *num-tiles*) rows. Otherwise, each of the quantiles 1 through MOD(*n*, *num-tiles*) is assigned (*n* / *num-tiles* + 1) rows while each of the quantiles (MOD(*n*, *num-tiles*) + 1) through *num-tiles* is assigned (*n* / *num-tiles*) rows. The result is the quantile rank which is associated with the current row.

Equivalent sort keys are not considered when rows are divided into quantiles. Rows with equivalent sort keys can be assigned to different quantiles based on the non-deterministic order of these sort keys. Therefore, NTILE is a non-deterministic function.

The **LAG** function returns the expression value for the row at *offset* rows before the current row. The *offset* must be a positive integer constant (SQLSTATE 42815). An *offset* value of 0 means the current row. If a window-partition-clause is specified, *offset* means *offset* rows before the current row and within the current partition. If *offset* is not specified, the value 1 is used. If *default-value* (which can be an expression) is specified, it will be returned if the offset goes beyond the scope of the current partition. Otherwise, the null value is returned. If 'IGNORE NULLS' is specified, all rows where the expression value for the row is the null value are not considered in the calculation. If 'IGNORE NULLS' is specified and all rows are null, *default-value* (or the null value if *default-value* was not specified) is returned.

The **LEAD** function returns the expression value for the row at *offset* rows after the current row. The *offset* must be a positive integer constant (SQLSTATE 42815). An *offset* value of 0 means the current row. If a window-partition-clause is specified, *offset* means *offset* rows after the current row and within the current partition. If *offset* is not specified, the value 1 is used. If *default-value* (which can be an expression) is specified, it will be returned if the offset goes beyond the scope of the current partition. Otherwise, the null value is returned. If 'IGNORE NULLS' is specified, all rows where the expression value for the row is the null value are not considered in the calculation. If 'IGNORE NULLS' is specified and all rows are null, *default-value* (or the null value if *default-value* was not specified) is returned.

The **FIRST_VALUE** function returns the expression value for the first row in an OLAP window. If 'IGNORE NULLS' is specified, all rows where the expression value for the row is the null value are not considered in the calculation. If 'IGNORE NULLS' is specified and all values in the OLAP window are null, FIRST_VALUE returns the null value.

The **LAST_VALUE** function returns the expression value for the last row in an OLAP window. If 'IGNORE NULLS' is specified, all rows where the expression value for the row is the null value are not considered in the calculation. If 'IGNORE NULLS' is specified and all values in the OLAP window are null, LAST_VALUE returns the null value.

The data type of the result of FIRST_VALUE, LAG, LAST_VALUE, and LEAD is the data type of the expression. The result can be null.

The NTH_VALUE function returns the expression value for the *nth-row* row in an OLAP window.

expression
An expression that specifies the current row in an OLAP window. The expression must return a value that is a built-in data type.(SQLSTATE 42884).

nth-row
An expression that specifies which row of the OLAP window to return. The expression must return a value that is a built-in numeric data type, a CHAR, or a VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported using implicit casting. If the expression is not a SMALLINT, INTEGER, or BIGINT, it is cast to BIGINT before the function is evaluated. The value must be greater than 0 (SQLSTATE 22016). The expression must be a constant, a variable, or a cast of a constant or variable (SQLSTATE 428I9).

FROM FIRST or FROM LAST
Specifies how *nth-row* is applied. If FROM FIRST is specified, *nth-row* is treated as counting forward from the first row in the OLAP window. If FROM LAST is specified, *nth-row* is treated as counting backward from the last row in the OLAP window.

RESPECT NULLS or IGNORE NULLS
Specifies how NULL values in the OLAP window are handled. If RESPECT NULLS is specified, all rows where the expression value for the row is the null value are considered in the calculation. If IGNORE NULLS is specified, all rows where the expression value for the row is the null value are not considered in the calculation.

The data type of the result of NTH_VALUE is the same as the data type of *expression*.

The result can be null. If *nth-row* is null, the result is the null value. If the number of rows in the OLAP window (including null values if RESPECT NULLS is specified or excluding null values if IGNORE NULLS is specified) is less than the value of *nth-row*, the result is the null value.

The NTH_VALUE function is a non-deterministic function because the window-order-clause is not required and when window-order-clause is specified, rows with equivalent sort keys have a non-deterministic order.

The RATIO_TO_REPORT function returns the ratio of an argument to the sum of the arguments in an OLAP partition. For example, the following functions are equivalent:

```
RATIO_TO_REPORT(expression) OVER (...)
CAST(expression AS DECFLOAT(34)) / SUM(expression) OVER(...)
```

The division is always performed using DECFLOAT(34). The result data type is DECFLOAT(34). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

PARTITION BY (*partitioning-expression*,...)
Defines the partition within which the function is applied. A *partitioning-expression* is an expression that is used in defining the partitioning of the result set. Each *column-name* that is referenced in a *partitioning-expression* must unambiguously reference a column of the result table of the subselect that contains the OLAP specification (SQLSTATE 42702 or 42703). A *partitioning-expression* cannot include a scalar fullselect or an XMLQUERY or XMLEXISTS expression (SQLSTATE 42822), or any function or query that is not deterministic or that has an external action (SQLSTATE 42845).

window-order-clause

ORDER BY (*sort-key-expression*,...)
Defines the ordering of rows within a partition that determines the value of the OLAP function or the meaning of the ROW values in the window-aggregation-group-clause (it does not define the ordering of the query result set).

sort-key-expression
An expression used in defining the ordering of the rows within a window partition. Each column name referenced in a *sort-key-expression* must unambiguously reference a column of the result set of the subselect, including the OLAP function (SQLSTATE 42702 or 42703). A *sort-key-expression* cannot include a scalar fullselect or an XMLQUERY or XMLEXISTS expression (SQLSTATE 42822), or any function or query that is not deterministic or that has an external action (SQLSTATE 42845). This clause is required for the RANK and DENSE_RANK functions (SQLSTATE 42601).

ASC
Uses the values of the sort-key-expression in ascending order.

DESC
Uses the values of the sort-key-expression in descending order.

NULLS FIRST
The window ordering considers null values *before* all non-null values in the sort order.

NULLS LAST
The window ordering considers null values *after* all non-null values in the sort order.

ORDER OF *table-designator*
Specifies that the same ordering used in *table-designator* should be applied to the result table of the subselect. There must be a table reference matching *table-designator* in the FROM clause of the subselect that specifies this clause (SQLSTATE 42703). The ordering that is applied is the same as if the columns of the ORDER BY clause in the nested subselect (or fullselect) were included in the outer subselect (or fullselect), and these columns were specified in place of the ORDER OF clause.

window-aggregation-group-clause
The aggregation group of a row R is a set of rows defined in relation to R (in the ordering of the rows of R's partition). This clause specifies the aggregation group. If this clause is not specified and a window-order-clause is also not specified, the aggregation group consists of all the rows of the window partition. This default can be specified explicitly using RANGE (as shown) or ROWS.

If window-order-clause is specified, the default behavior is different when window-aggregation-group-clause is not specified. The window aggregation group consists of all rows of the partition of R that precede R and that are peers of R in the window ordering of the window partition defined by the window-order-clause.

ROWS
Indicates the aggregation group is defined by counting rows.

RANGE
Indicates the aggregation group is defined by an offset from a sort key.

group-start
Specifies the starting point for the aggregation group. The aggregation group end is the current row. Specification of the group-start clause is equivalent to a group-between clause of the form "BETWEEN group-start AND CURRENT ROW".

group-between
Specifies the aggregation group start and end based on either ROWS or RANGE.

group-end
Specifies the ending point for the aggregation group. The aggregation group start is the current row. Specification of the group-end clause is equivalent to a group-between clause of the form "BETWEEN CURRENT ROW AND group-end".

UNBOUNDED PRECEDING
Includes the entire partition preceding the current row. This can be specified with either ROWS or RANGE. Also, this can be specified with multiple sort-key-expressions in the window-order-clause.

UNBOUNDED FOLLOWING
Includes the entire partition following the current row. This can be specified with either ROWS or RANGE. Also, this can be specified with multiple sort-key-expressions in the window-order-clause.

CURRENT ROW
Specifies the start or end of the aggregation group based on the current row. If ROWS is specified, the current row is the aggregation group boundary. If RANGE is specified, the aggregation group boundary includes the set of rows with the same values for the *sort-key-expressions* as the current row. This clause cannot be specified in *group-bound2* if *group-bound1* specifies *value* FOLLOWING.

***unsigned-constant* PRECEDING**
Specifies either the range or number of rows preceding the current row. If ROWS is specified, then *unsigned-constant* must be zero or a positive integer indicating a number of rows. If RANGE is specified, then the data type of *unsigned-constant* must be comparable to the type of the *sort-key-expression* of the *window-order-clause*. There can only be one *sort-key-expression*, and the data type of the *sort-key-expression* must allow subtraction. This clause cannot be specified in *group-bound2* if *group-bound1* is CURRENT ROW or *unsigned-constant* FOLLOWING.

***unsigned-constant* FOLLOWING**
Specifies either the range or number of rows following the current row. If ROWS is specified, then *unsigned-constant* must be zero or a positive integer indicating a number of rows. If RANGE is specified, then the data type of *unsigned-constant* must be comparable to the type of the *sort-key-expression* of the *window-order-clause*. There can only be one *sort-key-expression*, and the data type of the *sort-key-expression* must allow addition.

Examples

1. Display the ranking of employees, in order by surname, according to their total salary (based on salary plus bonus) that have a total salary more than $30,000.

```
SELECT EMPNO, LASTNAME, FIRSTNME, SALARY+BONUS AS TOTAL_SALARY,
    RANK() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY
  FROM EMPLOYEE WHERE SALARY+BONUS > 30000
  ORDER BY LASTNAME
```

Note that if the result is to be ordered by the ranking, then replace ORDER BY LASTNAME with:

```
ORDER BY RANK_SALARY
```

or

```
ORDER BY RANK() OVER (ORDER BY SALARY+BONUS DESC)
```

2. Rank the departments according to their average total salary.

```
SELECT WORKDEPT, AVG(SALARY+BONUS) AS AVG_TOTAL_SALARY,
    RANK() OVER (ORDER BY AVG(SALARY+BONUS) DESC) AS RANK_AVG_SAL
  FROM EMPLOYEE
  GROUP BY WORKDEPT
  ORDER BY RANK_AVG_SAL
```

3. Rank the employees within a department according to their education level. Having multiple employees with the same rank in the department should not increase the next ranking value.

```
SELECT WORKDEPT, EMPNO, LASTNAME, FIRSTNME, EDLEVEL,
    DENSE_RANK() OVER
      (PARTITION BY WORKDEPT ORDER BY EDLEVEL DESC) AS RANK_EDLEVEL
```

```
  FROM EMPLOYEE
  ORDER BY WORKDEPT, LASTNAME
```

4. Provide row numbers in the result of a query.

```
SELECT ROW_NUMBER() OVER (ORDER BY WORKDEPT, LASTNAME) AS NUMBER,
    LASTNAME, SALARY
  FROM EMPLOYEE
  ORDER BY WORKDEPT, LASTNAME
```

5. List the top five wage earners.

```
SELECT EMPNO, LASTNAME, FIRSTNME, TOTAL_SALARY, RANK_SALARY
  FROM (SELECT EMPNO, LASTNAME, FIRSTNME, SALARY+BONUS AS TOTAL_SALARY,
    RANK() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY
    FROM EMPLOYEE) AS RANKED_EMPLOYEE
  WHERE RANK_SALARY < 6
  ORDER BY RANK_SALARY
```

Note that a nested table expression was used to first compute the result, including the rankings, before the rank could be used in the WHERE clause. A common table expression could also have been used.

6. For each department, list employee salaries and show how much less each person makes compared to the employee in that department with the next highest salary.

```
SELECT EMPNO, WORKDEPT, LASTNAME, FIRSTNME, JOB, SALARY,
    LEAD(SALARY, 1) OVER (PARTITION BY WORKDEPT
      ORDER BY SALARY) - SALARY AS DELTA_SALARY
  FROM EMPLOYEE
  ORDER BY WORKDEPT, SALARY
```

7. Calculate an employee's salary relative to the salary of the employee who was first hired for the same type of job.

```
SELECT JOB, HIREDATE, EMPNO, LASTNAME, FIRSTNME, SALARY,
    FIRST_VALUE(SALARY) OVER (PARTITION BY JOB
      ORDER BY HIREDATE) AS FIRST_SALARY,
    SALARY - FIRST_VALUE(SALARY) OVER (PARTITION BY JOB
      ORDER BY HIREDATE) AS DELTA_SALARY
  FROM EMPLOYEE
  ORDER BY JOB, HIREDATE
```

8. Calculate the average close price for stock XYZ during the month of January, 2006. If a stock doesn't trade on a given day, its close price in the DAILYSTOCKDATA table is the null value. Instead of returning the null value for days that a stock doesn't trade, use the COALESCE function and LAG function to return the close price for the most recent day the stock was traded. Limit the search for a previous not-null close value to one month before January 1st, 2006.

```
WITH V1(SYMBOL, TRADINGDATE, CLOSEPRICE) AS
(
SELECT SYMBOL, TRADINGDATE,
  COALESCE(CLOSEPRICE,
    LAG(CLOSEPRICE,
      1,
      CAST(NULL AS DECIMAL(8,2)),
      'IGNORE NULLS')
    OVER (PARTITION BY SYMBOL
    ORDER BY TRADINGDATE)
)
FROM DAILYSTOCKDATA
WHERE SYMBOL = 'XYZ' AND
  TRADINGDATE BETWEEN '2005-12-01' AND '2006-01-31'
)
SELECT SYMBOL, AVG(CLOSEPRICE) AS AVG
  FROM V1
  WHERE TRADINGDATE BETWEEN '2006-01-01' AND '2006-01-31'
  GROUP BY SYMBOL
```

9. Calculate the 30-day moving average for stocks ABC and XYZ during the year 2005.

```
WITH V1(SYMBOL, TRADINGDATE, MOVINGAVG30DAY) AS
(
```

```
SELECT SYMBOL, TRADINGDATE,
  AVG(CLOSEPRICE) OVER (PARTITION BY SYMBOL
    ORDER BY TRADINGDATE
  ROWS BETWEEN 29 PRECEDING AND CURRENT ROW)
FROM DAILYSTOCKDATA
WHERE SYMBOL IN ('ABC', 'XYZ')
  AND TRADINGDATE BETWEEN DATE('2005-01-01') - 2 MONTHS
    AND '2005-12-31'
)
SELECT SYMBOL, TRADINGDATE, MOVINGAVG30DAY
  FROM V1
  WHERE TRADINGDATE BETWEEN '2005-01-01' AND '2005-12-31'
  ORDER BY SYMBOL, TRADINGDATE
```

10. Use an expression to define the cursor position and query a sliding window of 50 rows before that position.

```
SELECT DATE, FIRST_VALUE(CLOSEPRICE + 100) OVER
    (PARTITION BY SYMBOL
    ORDER BY DATE
    ROWS BETWEEN 50 PRECEDING AND 1 PRECEDING) AS FV
  FROM DAILYSTOCKDATA
  ORDER BY DATE
```

11. For each employee, calculate the average salary for the set of employees that includes those employees in the same department who have an education level 1 lower and 1 higher than the employee.

```
SELECT WORKDEPT, EDLEVEL, SALARY, AVG(SALARY)
      OVER (PARTITION BY WORKDEPT
      ORDER BY EDLEVEL
      RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING)
   FROM EMPLOYEE
   ORDER BY WORKDEPT, EDLEVEL
```

12. Calculate which quartile (4-quantiles) each employee's salary is in.

```
SELECT EMPNO, SALARY, NTILE(4) OVER
      (ORDER BY SALARY) AS QUARTILE
   FROM EMPLOYEE
   ORDER BY SALARY
```

The result set is:

```
EMPNO  SALARY      QUARTILE
------ ----------- -----------
200340    31840.00           1
000290    35340.00           1
200330    35370.00           1
000310    35900.00           1
200310    35900.00           1
000280    36250.00           1
000270    37380.00           1
000300    37750.00           1
200240    37760.00           1
200120    39250.00           1
000320    39950.00           1
000230    42180.00           2
000340    43840.00           2
000170    44680.00           2
000330    45370.00           2
200280    46250.00           2
200010    46500.00           2
000260    47250.00           2
000240    48760.00           2
000250    49180.00           2
000120    49250.00           2
000220    49840.00           2
000190    50450.00           3
000180    51340.00           3
000150    55280.00           3
000200    57740.00           3
000160    62250.00           3
200170    64680.00           3
000110    66500.00           3
000210    68270.00           3
000140    68420.00           3
```

```
200140    68420.00           3
200220    69840.00           4
000060    72250.00           4
000130    73800.00           4
000050    80175.00           4
000100    86150.00           4
000090    89750.00           4
000020    94250.00           4
000070    96170.00           4
000030    98250.00           4
000010   152750.00           4

  42 record(s) selected.
```

13. The query in the following example divides the rows into 3 buckets, grouping them by maximum salary. The maximum salary is included to show what values go into each bucket:

```
   SELECT NTILE(3) OVER (ORDER BY MAX_SALARY) AS Bucket,
MAX_SALARY FROM GOSALESDW.EMP_POSITION_DIM;
```

A portion of the output from the query is in the following table:

Table 27. Example output

BUCKET	MAX_SALARY
1	0.00
...	...
1	35000.00
2	5000.00
...	...
2	12000.00
3	13000.00
...	...
3	301500.00

14. Find the cumulative distribution and the relative percentile rank of each employee's salary within their department.

```
   SELECT EMPNO, WORKDEPT, SALARY,
      CAST(CUME_DIST() OVER (PARTITION BY WORKDEPT ORDER BY SALARY) AS DECIMAL(4,3))
         AS CUME_DIST,
      CAST(PERCENT_RANK() OVER (PARTITION BY WORKDEPT ORDER BY SALARY)
         AS DECIMAL(4,3))
         AS PERCENT_RANK FROM EMP
      ORDER BY WORKDEPT, SALARY
```

The result set is:

```
EMPNO  WORKDEPT SALARY      CUME_DIST PERCENT_RANK
------ -------- ----------- --------- ------------
200120 A00         39250.00     0.200        0.000
200010 A00         46500.00     0.400        0.250
000120 A00         49250.00     0.600        0.500
000110 A00         66500.00     0.800        0.750
000010 A00        152750.00     1.000        1.000
000020 B01         94250.00     1.000        0.000
000140 C01         68420.00     0.500        0.000
200140 C01         68420.00     0.500        0.000
000130 C01         73800.00     0.750        0.666
000030 C01         98250.00     1.000        1.000
000170 D11         44680.00     0.090        0.000
000220 D11         49840.00     0.181        0.100
000190 D11         50450.00     0.272        0.200
000180 D11         51340.00     0.363        0.300
000150 D11         55280.00     0.454        0.400
000200 D11         57740.00     0.545        0.500
```

```
000160 D11         62250.00     0.636        0.600
200170 D11         64680.00     0.727        0.700
000210 D11         68270.00     0.818        0.800
200220 D11         69840.00     0.909        0.900
000060 D11         72250.00     1.000        1.000
000270 D21         37380.00     0.142        0.000
200240 D21         37760.00     0.285        0.166
000230 D21         42180.00     0.428        0.333
000260 D21         47250.00     0.571        0.500
000240 D21         48760.00     0.714        0.666
000250 D21         49180.00     0.857        0.833
000070 D21         96170.00     1.000        1.000
000050 E01         80175.00     1.000        0.000
000290 E11         35340.00     0.142        0.000
000310 E11         35900.00     0.428        0.166
200310 E11         35900.00     0.428        0.166
000280 E11         36250.00     0.571        0.500
000300 E11         37750.00     0.714        0.666
200280 E11         46250.00     0.857        0.833
000090 E11         89750.00     1.000        1.000
200340 E21         31840.00     0.166        0.000
200330 E21         35370.00     0.333        0.200
000320 E21         39950.00     0.500        0.400
000340 E21         43840.00     0.666        0.600
000330 E21         45370.00     0.833        0.800
000100 E21         86150.00     1.000        1.000

  42 record(s) selected.
```

15. Compare each employee's salary to the highest salary and second highest salary in the department.

```
SELECT WORKDEPT, SALARY, FIRST_VALUE(SALARY)
OVER (PARTITION BY WORKDEPT ORDER BY SALARY DESC) AS FIRST,
NTH_VALUE(SALARY, 2) OVER (PARTITION BY WORKDEPT ORDER BY SALARY DESC) AS SECOND
   FROM EMP
   ORDER BY WORKDEPT, SALARY
```

The result set is:

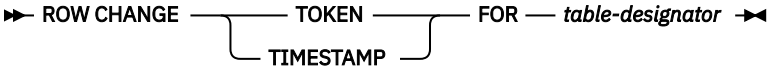
```
WORKDEPT SALARY      FIRST       SECOND
-------- ----------- ----------- --------
A00         39250.00   152750.00 66500.00
A00         46500.00   152750.00 66500.00
A00         49250.00   152750.00 66500.00
A00         66500.00   152750.00 66500.00
A00        152750.00   152750.00 66500.00
B01         94250.00    94250.00 -
C01         68420.00    98250.00 73800.00
C01         68420.00    98250.00 73800.00
C01         73800.00    98250.00 73800.00
C01         98250.00    98250.00 73800.00
D11         44680.00    72250.00 69840.00
D11         49840.00    72250.00 69840.00
D11         50450.00    72250.00 69840.00
D11         51340.00    72250.00 69840.00
D11         55280.00    72250.00 69840.00
D11         57740.00    72250.00 69840.00
D11         62250.00    72250.00 69840.00
D11         64680.00    72250.00 69840.00
D11         68270.00    72250.00 69840.00
D11         69840.00    72250.00 69840.00
D11         72250.00    72250.00 69840.00
D21         37380.00    96170.00 49180.00
D21         37760.00    96170.00 49180.00
D21         42180.00    96170.00 49180.00
D21         47250.00    96170.00 49180.00
D21         48760.00    96170.00 49180.00
D21         49180.00    96170.00 49180.00
D21         96170.00    96170.00 49180.00
E01         80175.00    80175.00 -
E11         35340.00    89750.00 46250.00
E11         35900.00    89750.00 46250.00
E11         35900.00    89750.00 46250.00
E11         36250.00    89750.00 46250.00
E11         37750.00    89750.00 46250.00
E11         46250.00    89750.00 46250.00
E11         89750.00    89750.00 46250.00
E21         31840.00    86150.00 45370.00
E21         35370.00    86150.00 45370.00
E21         39950.00    86150.00 45370.00
```

```
E21         43840.00    86150.00 45370.00
E21         45370.00    86150.00 45370.00
E21         86150.00    86150.00 45370.00

  42 record(s) selected.
```

ROW CHANGE expression

A ROW CHANGE expression returns a token or a timestamp that represents the last change to a row.

row-change-expression



TOKEN
Specifies that a BIGINT value representing a relative point in the modification sequence of a row is to be returned. If the row has not been changed, the result is a token that represents when the initial value was inserted. The result can be null. ROW CHANGE TOKEN is not deterministic.

TIMESTAMP
Specifies that a TIMESTAMP value representing the last time that a row was changed is to be returned. If the row has not been changed, the result is the time that the initial value was inserted. The result can be null. ROW CHANGE TIMESTAMP is not deterministic.

FOR *table-designator*
Identifies the table in which the expression is referenced. The *table-designator* must uniquely identify a base table, view, or nested table expression (SQLSTATE 42867). If *table-designator* identifies a view or a nested table expression, the ROW CHANGE expression returns the TOKEN or TIMESTAMP of the base table of the view or nested table expression. The view or nested table expression must contain only one base table in its outer subselect (SQLSTATE 42867). If the *table-designator* is a view or nested table expression, it must be deletable (SQLSTATE 42703). For information about deletable views, see the "Notes" section of "CREATE VIEW". The table designator of a ROW CHANGE TIMESTAMP expression must resolve to a base table that contains a row change timestamp column (SQLSTATE 55068).

Notes

- ROW CHANGE TOKEN and ROW CHANGE TIMESTAMP are not valid expressions for a column-organized table (SQLSTATE 42703).

Examples

- Return a timestamp value that corresponds to the most recent change to each row from the EMPLOYEE table for employees in department 20. Assume that the EMPLOYEE table has been altered to contain a column defined with the ROW CHANGE TIMESTAMP clause.

```
SELECT ROW CHANGE TIMESTAMP FOR EMPLOYEE
  FROM EMPLOYEE WHERE DEPTNO = 20
```

- Return a BIGINT value that represents a relative point in the modification sequence of the row corresponding to employee number 3500. Also return the RID_BIT scalar function value that is to be used in an optimistic locking DELETE scenario. Specify the WITH UR option to get the latest ROW CHANGE TOKEN value.

```
SELECT ROW CHANGE TOKEN FOR EMPLOYEE, RID_BIT (EMPLOYEE)
  FROM EMPLOYEE WHERE EMPNO = '3500' WITH UR
```

The preceding statement succeeds whether or not there is a row change timestamp column in the EMPLOYEE table. The following searched DELETE statement deletes the row specified by the

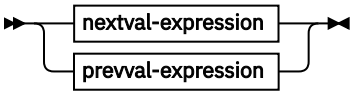
ROW CHANGE TOKEN and RID_BIT values from the preceding SELECT statement, assuming the two parameter marker values are set to the values obtained from the preceding statement.

```
DELETE FROM EMPLOYEE E
  WHERE RID_BIT (E) = ? AND ROW CHANGE TOKEN FOR E = ?
```

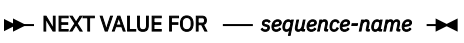
Sequence reference

A sequence reference is an expression which references a sequence defined at the application server.

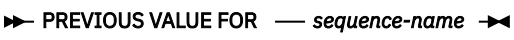
sequence-reference



nextval-expression



prevval-expression



NEXT VALUE FOR *sequence-name*
A NEXT VALUE expression generates and returns the next value for the sequence specified by *sequence-name*.

PREVIOUS VALUE FOR *sequence-name*
A PREVIOUS VALUE expression returns the most recently generated value for the specified sequence for a previous statement within the current application process. This value can be referenced repeatedly by using PREVIOUS VALUE expressions that specify the name of the sequence. There may be multiple instances of PREVIOUS VALUE expressions specifying the same sequence name within a single statement; they all return the same value. In a partitioned database environment, a PREVIOUS VALUE expression may not return the most recently generated value.

A PREVIOUS VALUE expression can only be used if a NEXT VALUE expression specifying the same sequence name has already been referenced in the current application process, in either the current or a previous transaction (SQLSTATE 51035).

Notes

- ***Authorization:*** If a sequence-reference is used in a statement, the privileges held by the authorization ID of the statement must include at least one of the following privileges:
  - The USAGE privilege on the sequence
  - DATAACCESS authority
- A new value is generated for a sequence when a NEXT VALUE expression specifies the name of that sequence. However, if there are multiple instances of a NEXT VALUE expression specifying the same sequence name within a query, the counter for the sequence is incremented only once for each row of the result, and all instances of NEXT VALUE return the same value for a row of the result.
- The same sequence number can be used as a unique key value in two separate tables by referencing the sequence number with a NEXT VALUE expression for the first row (this generates the sequence value), and a PREVIOUS VALUE expression for the other rows (the instance of PREVIOUS VALUE refers to the sequence value most recently generated in the current session), as shown in the following example:

```
INSERT INTO order(orderno, cutno)
  VALUES (NEXT VALUE FOR order_seq, 123456);

INSERT INTO line_item (orderno, partno, quantity)
  VALUES (PREVIOUS VALUE FOR order_seq, 987654, 1);
```

- NEXT VALUE and PREVIOUS VALUE expressions can be specified in the following places:

  - select-statement or SELECT INTO statement (within the select-clause, provided that the statement does not contain a DISTINCT keyword, a GROUP BY clause, an ORDER BY clause, a UNION keyword, an INTERSECT keyword, or EXCEPT keyword)
  - INSERT statement (within a VALUES clause)
  - INSERT statement (within the select-clause of the fullselect)
  - UPDATE statement (within the SET clause (either a searched or a positioned UPDATE statement), except that NEXT VALUE cannot be specified in the select-clause of the fullselect of an expression in the SET clause)
  - SET Variable statement (except within the select-clause of the fullselect of an expression; a NEXT VALUE expression can be specified in a trigger, but a PREVIOUS VALUE expression cannot)
  - VALUES INTO statement (within the select-clause of the fullselect of an expression)
  - CREATE PROCEDURE statement (within the routine-body of an SQL procedure)
  - CREATE TRIGGER statement within the triggered-action (a NEXT VALUE expression may be specified, but a PREVIOUS VALUE expression cannot)
- NEXT VALUE and PREVIOUS VALUE expressions cannot be specified (SQLSTATE 428F9) in the following places:
  - Join condition of a full outer join
  - DEFAULT value for a column in a CREATE TABLE or ALTER TABLE statement
  - Generated column definition in a CREATE TABLE or ALTER TABLE statement
  - Summary table definition in a CREATE TABLE or ALTER TABLE statement
  - Condition of a CHECK constraint
  - CREATE TRIGGER statement (a NEXT VALUE expression may be specified, but a PREVIOUS VALUE expression cannot)
  - CREATE VIEW statement
  - CREATE METHOD statement
  - CREATE FUNCTION statement
  - An argument list of an XMLQUERY, XMLEXISTS, or XMLTABLE expression
- In addition, a NEXT VALUE expression cannot be specified (SQLSTATE 428F9) in the following places:
  - CASE expression
  - Parameter list of an aggregate function
  - Subquery in a context other than those explicitly allowed, as described previously
  - SELECT statement for which the outer SELECT contains a DISTINCT operator
  - Join condition of a join
  - SELECT statement for which the outer SELECT contains a GROUP BY clause
  - SELECT statement for which the outer SELECT is combined with another SELECT statement using the UNION, INTERSECT, or EXCEPT set operator
  - Nested table expression
  - Parameter list of a table function
  - WHERE clause of the outer-most SELECT statement, or a DELETE or UPDATE statement
  - ORDER BY clause of the outer-most SELECT statement
  - select-clause of the fullselect of an expression, in the SET clause of an UPDATE statement
  - IF, WHILE, DO ... UNTIL, or CASE statement in an SQL routine
- When a value is generated for a sequence, that value is consumed, and the next time that a value is requested, a new value will be generated. This is true even when the statement containing the NEXT VALUE expression fails or is rolled back.

If an INSERT statement includes a NEXT VALUE expression in the VALUES list for the column, and if an error occurs at some point during the execution of the INSERT (it could be a problem in generating the next sequence value, or a problem with the value for another column), then an insertion failure occurs (SQLSTATE 23505), and the value generated for the sequence is considered to be consumed. In some cases, reissuing the same INSERT statement might lead to success.

For example, consider an error that is the result of the existence of a unique index for the column for which NEXT VALUE was used and the sequence value generated already exists in the index. It is possible that the next value generated for the sequence is a value that does not exist in the index and so the subsequent INSERT would succeed.

- ***Scope of PREVIOUS VALUE:*** The value of PREVIOUS VALUE persists until the next value is generated for the sequence in the current session, the sequence is dropped or altered, or the application session ends. The value is unaffected by COMMIT or ROLLBACK statements. The value of PREVIOUS VALUE cannot be directly set and is a result of executing the NEXT VALUE expression for the sequence.

  A technique commonly used, especially for performance, is for an application or product to manage a set of connections and route transactions to an arbitrary connection. In these situations, the availability of the PREVIOUS VALUE for a sequence should be relied on only until the end of the transaction. Examples of where this type of situation can occur include applications that use XA protocols, use connection pooling, use the connection concentrator, and use HADR to achieve failover.
- If in generating a value for a sequence, the maximum value for the sequence is exceeded (or the minimum value for a descending sequence) and cycles are not permitted, then an error occurs (SQLSTATE 23522). In this case, the user could ALTER the sequence to extend the range of acceptable values, or enable cycles for the sequence, or DROP and CREATE a new sequence with a different data type that has a larger range of values.

  For example, a sequence may have been defined with a data type of SMALLINT, and eventually the sequence runs out of assignable values. DROP and re-create the sequence with the new definition to redefine the sequence as INTEGER.
- A reference to a NEXT VALUE expression in the select statement of a cursor refers to a value that is generated for a row of the result table. A sequence value is generated for a NEXT VALUE expression for each row that is fetched from the database. If blocking is done at the client, the values may have been generated at the server before the processing of the FETCH statement. This can occur when there is blocking of the rows of the result table. If the client application does not explicitly FETCH all the rows that the database has materialized, then the application will not see the results of all the generated sequence values (for the materialized rows that were not returned).
- A reference to a PREVIOUS VALUE expression in the select statement of a cursor refers to a value that was generated for the specified sequence before the opening of the cursor. However, closing the cursor can affect the values returned by PREVIOUS VALUE for the specified sequence in subsequent statements, or even for the same statement in the event that the cursor is reopened. This would be the case when the select statement of the cursor included a reference to NEXT VALUE for the same sequence name.
- ***Syntax alternatives***: The following are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - NEXTVAL and PREVVAL can be specified in place of NEXT VALUE and PREVIOUS VALUE
  - *sequence-name*.NEXTVAL can be specified in place of NEXT VALUE FOR *sequence-name*
  - *sequence-name*.CURRVAL can be specified in place of PREVIOUS VALUE FOR *sequence-name*

Examples

Assume that there is a table called "order", and that a sequence called "order_seq" is created as follows:

```
CREATE SEQUENCE order_seq
  START WITH 1
  INCREMENT BY 1
  NO MAXVALUE
  NO CYCLE
  CACHE 24
```

Following are some examples of how to generate an "order_seq" sequence number with a NEXT VALUE expression:

```
INSERT INTO order(orderno, custno)
  VALUES (NEXT VALUE FOR order_seq, 123456);
```

or

```
UPDATE order
  SET orderno = NEXT VALUE FOR order_seq
  WHERE custno = 123456;
```

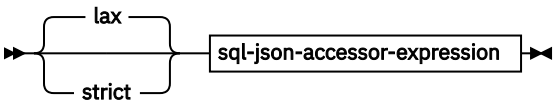
or

```
VALUES NEXT VALUE FOR order_seq INTO :hv_seq;
```

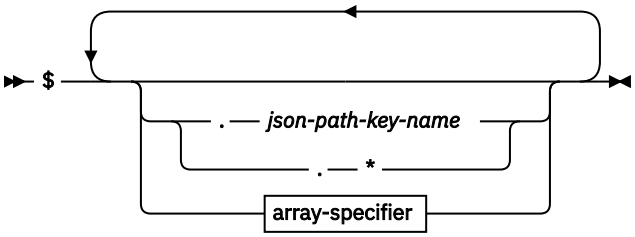
sql-json-path-expression

An SQL/JSON path expression defines access to the elements of a JSON document.

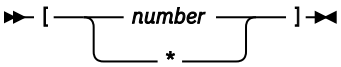
sql-json-path-expression



sql-json-accessor-expression



array-specifier



lax or strict

Specifies the JSON path mode.

lax

Specifies that certain structural errors are tolerated when the current JSON document is navigated, including the following structural errors:

- Automatic unnesting of arrays.
- Automatic wrapping of scalar values to be a single element array, if referenced as an array.
- Specifying nonexistent items, including array index values that are out of range.

If an item does not exist, the SQL/JSON path expression returns an empty string, which is handled according to the options specified in a function's ON EMPTY clause.

strict

Specifies that an error is reported when the specified path expression cannot be used to navigate the current JSON document. The error is handled according to the current ON ERROR clause.

sql-json-accessor-expression

$

Specifies the start of the context item to which the rest of the SQL/JSON path expression is applied.

json-path-key-name
Specifies the key name of a key:value pair in the JSON document.

*****
Specifies that the values for all the keys are returned as an SQL/JSON sequence.

array-specifier
Specifies the index values to apply to an array. The first element of the array has an index of 0. If specified index is out of range, then it is considered an error.

number
An unsigned integer constant that represents an array element. The first element of the array has an index of 0.

*****
Indicates that all array elements are selected.

Special semantics are associated with some characters, when used as part of a SQL/JSON path expression:

.
Used for going to the next level within JSON document.

*****
Used for matching all the keys at the current level.

[and]
Used for specifying an index of an array.

To allow these characters to be used within *json-path-key-name*, use an escape backslash character (\) before these characters to indicate that these characters are part of the key name.

Examples

1. This example is based on the following JSON document:

```
{ "isbn": "123-456-222", "author": [ { "name":"Jones"},{"name":"Smith"}]}
```

The following table shows the results of using various SQL/JSON path expressions to access items in the JSON document:

Path	Number of matches	Matches
$.isbn	1	"123-456-222"
$.author[0].name	1	"Jones"
$.author[1].name	1	"Smith"

2. This example is based on the following JSON document:

```
{
  "person"   : {"firstname":"Fred", "lastname":"Gauss"},
  "where"    : "General Products",
  "friends"  : [ { "name": "Lili","rank": 5 }, {"name": "Hank", "rank": 7} ],
  "work.area": "Finance"
}
```

The following table shows the results of using various SQL/JSON path expressions to access items in the JSON document:

Path	Number of matches	Matches
$.person.lastname	1	"Gauss"
$.friends	1	[{ "name": "Lili", "rank": 5 }, { "name": "Hank", "rank": 7}]

Path	Number of matches	Matches
$.*.firstname	1	"Fred"
$.person.*	2	"Fred" and "Gauss"
$.friends[*]	2	{ "name": "Lili", "rank": 5 } and { "name": "Hank", "rank": 7}
$.friends[*].rank	2	5 and 7
$.work.area	0	
$.work\.area	1	"Finance"

3. This example is based on the following JSON document:

```
{ "a":[{"b1":10}, {"b2":11}], "c":"hi" }
```

The following table shows the results of using various SQL/JSON path expressions to access items in the JSON document:

Path	Matches	Remark
lax $.a.b1	10	Automatic unnesting of array element at `'a'`. Treated as `'a[*]'`.
strict $.a.b1	Error	No automatic unnesting of array.
lax $.c[0]	"hi"	Automatic wrapping of scalar value into single element array.
strict $.c[0]	Error	No automatic wrapping of scalar value into single element array.

4. This example illustrates the difference between JSON_VALUE and JSON_QUERY behavior. This example is based on the following JSON document:

```
{ "a": [1,2], "b": { "c1":1, "c2":2 } }
```

This example is based on the following path results:

Path expression	Number of values returned	Values returned
$.a	Single	[1,2]
$.b.*	Multiple	1 2

The following table shows the results of using various SQL/JSON path expressions to access items in the JSON document:

Operator	$.a	$.b.*	Remark
JSON_VALUE	Error	Error	Error for accessing array type and multiple matches.
JSON_QUERY WITHOUT ARRAY WRAPPER	[1,2]	Error	Error for multiple values with no array wrapper.
JSON_QUERY WITH UNCONDITIONAL ARRAY WRAPPER	[[1,2]]	[1,2]	You must use an array wrapper, even if the type is array.

Operator	$.a	$.b.*	Remark
JSON_QUERY WITH CONDITIONAL ARRAY WRAPPER	[1,2]	[1,2]	An array wrapper is not necessary, if the type is array.

Subtype treatment

The *subtype-treatment* is used to cast a structured type expression into one of its subtypes.

subtype-treatment

The static type of *expression* must be a user-defined structured type, and that type must be the same type as, or a supertype of, *data-type*. If the type name in *data-type* is unqualified, the SQL path is used to resolve the type reference. The static type of the result of subtype-treatment is *data-type*, and the value of the subtype-treatment is the value of the expression. At run time, if the dynamic type of the expression is not *data-type* or a subtype of *data-type*, an error is returned (SQLSTATE 0D000).

Example

- If an application knows that all column object instances in a column CIRCLE_COL have the dynamic type COLOREDCIRCLE, use the following query to invoke the method RGB on such objects. Assume the existence of a table called RINGS, with a column CIRCLE_COL of structured type CIRCLE. Also, assume that COLOREDCIRCLE is a subtype of CIRCLE and that the method RGB has been defined previously for COLOREDCIRCLE as `RGB() RETURNS DOUBLE`.

  ```
  SELECT TREAT (CIRCLE_COL AS COLOREDCIRCLE)..RGB()
    FROM RINGS
  ```

  At run time, if there are instances of dynamic type CIRCLE, an error is raised (SQLSTATE 0D000). This error can be avoided by using the TYPE predicate in a CASE expression, as follows:

  ```
  SELECT (CASE
    WHEN CIRCLE_COL IS OF (COLOREDCIRCLE)
      THEN TREAT (CIRCLE_COL AS COLOREDCIRCLE)..RGB()
      ELSE NULL
    END)
    FROM RINGS
  ```

Determining data types of untyped expressions

An untyped expression refers to the usage of a parameter marker which is specified without a target data type associated with it, a null value which is specified without a target data type associated with it, or a DEFAULT keyword.

Untyped expressions can be used in SQL statements as long as one of the following conditions is true:

- A PREPARE statement is being executed by a CLI or JDBC application to compile the SQL statement; the client interface is using deferred prepare; and the registry variable, DB2_DEFERRED_PREPARE_SEMANTICS is set to YES. In this case, any untyped parameter marker derives its data type based on the input descriptor associated with the subsequent OPEN or EXECUTE statement. The length attribute is set to the maximum of the length according to the UNTYPED row, as described in the Table 20 on page 121 in "Functions" and the length as determined from the following tables. For data types not listed as a target type in Table 20 on page 121 in "Functions", the length from the input descriptor associated with the subsequent OPEN or EXECUTE statement will be used. The data types and lengths may be modified depending on the usage of the untyped parameter marker in the SQL statement.
- The data type can be determined based on the context in the SQL statement. These locations and the resulting data types are shown in the following table. The locations are grouped into expressions, predicates, built-in functions, and user-defined routines to assist in determining the applicability of an untyped expression. If the data type cannot be determined based on the context, an error is issued.

For some cases not listed, untyped expressions in a select list will be resolved to a data type determined based on the usage in the SQL statement.

The code page of the untyped expression is determined by the context. Where there is no context, the code page is the same as if the untyped expression was cast to a VARCHAR data type.

The tables that follow show character string and graphic string data types in string units associated with a database environment where the string units default is SYSTEM. If the Unicode database environment has the string units set to CODEUNITS32, then any character string or graphic string length attributes that represent the data type maximum length should be considered to represent the data type maximum in CODEUNITS32. All character string or graphic string data types have the default string units of the database environment.

Table 28. Untyped Expression Usage in Expressions (Including Select List, CASE, and VALUES)

Untyped Expression Location	**Data Type**
Alone in a select list	If the untyped expression is unnamed or is named but not subsequently referenced in the SQL statement, then an error is returned, unless the untyped expression is the null value. In such cases, the data type is VARCHAR(1). If the untyped expression is named and subsequently referenced in the SQL statement, then the data type may be determined from the subsequent usage. For more information, refer to the "Determining data type from usage" note that follows this table.
Both operands of a single arithmetic operator, after considering operator precedence and order of operation rules Includes cases such as: `(? + ?) + 10`	DECFLOAT(34)
One operand of a single operator in an arithmetic expression (not a datetime expression) Includes cases such as: `? + (? * 10)`	The data type of the other operand
Labelled duration within a datetime expression (note that the portion of a labelled duration that indicates the type of units cannot be a parameter marker)	DECIMAL(15,0)
Any other operand of a datetime expression (for example, 'timecol + ?' or '? - datecol')	Error
Both operands of a CONCAT operator	VARCHAR(254)
One operand of a CONCAT operator when the other operand is a non-CLOB character data type	If one operand is either CHAR(*n*) or VARCHAR(*n*), where *n* is less than 128, the other is VARCHAR(254 - *n*); in all other cases, the data type is VARCHAR(254)

Table 28. Untyped Expression Usage in Expressions (Including Select List, CASE, and VALUES) (continued)

Untyped Expression Location	Data Type
One operand of a CONCAT operator when the other operand is a non-BLOB binary data type	If one operand is either BINARY(*n*) or VARBINARY(*n*), where *n* is less than 128, the other is VARBINARY(254 - *n*); in all other cases, the data type is VARBINARY(254)
One operand of a CONCAT operator, when the other operand is a non-DBCLOB graphic data type	If one operand is either GRAPHIC(*n*) or VARGRAPHIC(*n*), where *n* is less than 64, the other is VARGRAPHIC(127 - *n*); in all other cases, the data type is VARGRAPHIC(127)
One operand of a CONCAT operator, when the other operand is a large object string	Same as that of the other operand
The expression following the CASE keyword in a simple CASE expression	Result of applying the "Rules for the result data types" to the expressions following the WHEN keyword that are other than untyped expressions
At least one of the result-expressions in a CASE expression (both simple and searched), with the rest of the result-expressions being untyped expressions	Error
Any or all expressions following the WHEN keyword in a simple CASE expression	Result of applying the "Rules for result data types" to the expression following CASE and the expressions following WHEN keyword that are other than an untyped expression
A result-expression in a CASE expression (both simple and searched), when at least one result-expression is not an untyped expression	Result of applying the "Rules for result data types" to all result-expressions that are other than an untyped expression
Alone as a column-expression in a single-row VALUES clause that is not within an INSERT statement and not within the VALUES clause of in insert operation of a MERGE statement	Error if the untyped expression is unnamed or is named but not subsequently referenced in the SQL statement. If the untyped expression is named and subsequently referenced in the SQL statement, then the data type may be determined from the subsequent usage. For more information, refer to the "Determining data type from usage" note that follows this table.
Alone as a column-expression in a multi-row VALUES clause that is not within an INSERT statement, and for which the column-expressions in the same position in all other row-expressions are untyped expressions	Error if the untyped expression is unnamed or is named but not subsequently referenced in the SQL statement. If the untyped expression is named and subsequently referenced in the SQL statement, then the data type may be determined from the subsequent usage. For more information, refer to the "Determining data type from usage" note that follows this table.
Alone as a column-expression in a multi-row VALUES clause that is not within an INSERT statement, and for which the expression in the same position of at least one other row-expression is not an untyped expression	Result of applying the "Rules for result data types" on all operands that are other than untyped expressions

Table 28. Untyped Expression Usage in Expressions (Including Select List, CASE, and VALUES) (continued)

Untyped Expression Location	Data Type
Alone as a column-expression in a single-row VALUES clause within an INSERT statement	The data type of the column. If the column is defined as a user-defined distinct type, it is the source data type of the user-defined distinct type. If the column is defined as a user-defined structured type, it is the structured type, also indicating the return type of the transform function.
Alone as a column-expression in a multi-row VALUES clause within an INSERT statement	The data type of the column. If the column is defined as a user-defined distinct type, it is the source data type of the user-defined distinct type. If the column is defined as a user-defined structured type, it is the structured type, also indicating the return type of the transform function.
Alone as a column-expression in a values-clause of the source table for a MERGE statement	Error if the untyped expression is unnamed or is named but not subsequently referenced in the SQL statement. If the untyped expression is named and subsequently referenced in the SQL statement, then the data type may be determined from the subsequent usage. For more information, refer to the "Determining data type from usage" note that follows this table.
Alone as a column-expression in the VALUES clause of an insert operation of a MERGE statement	The data type of the column. If the column is defined as a user-defined distinct type, it is the source data type of the user-defined distinct type. If the column is defined as a user-defined structured type, it is the structured type, also indicating the return type of the transform function.
Alone as a column-expression on the right side of assignment-clause for an update operation of a MERGE statement	The data type of the column. If the column is defined as a user-defined distinct type, it is the source data type of the user-defined distinct type. If the column is defined as a user-defined structured type, it is the structured type, also indicating the return type of the transform function.
Alone as a column-expression on the right side of a SET clause in an UPDATE statement	The data type of the column. If the column is defined as a user-defined distinct type, it is the source data type of the user-defined distinct type. If the column is defined as a user-defined structured type, it is the structured type, also indicating the return type of the transform function.
As a value on the right side of a SET special register statement	The data type of the special register
Argument of the TABLESAMPLE clause of the tablesample-clause of a table-reference	DOUBLE
Argument of the REPEATABLE subclause of the tablesample-clause of a table-reference	INTEGER
Alone as fetch-row-count in a fetch-clause	BIGINT
Alone as offset-row-count in an OFFSET clause	BIGINT
As a value in a FREE LOCATOR statement	Locator

Table 28. Untyped Expression Usage in Expressions (Including Select List, CASE, and VALUES) (continued)

Untyped Expression Location	Data Type
As a value for the password in a SET ENCRYPTION PASSWORD statement	VARCHAR(128)

Note:

Determining data type from usage

The following is an example of how the data type for an untyped expression can be determined from subsequent usage:

If the named untyped expression is subsequently referenced in a comparison operator, it will then have the data type of the other operand. If there are multiple references of the named untyped expression in the SQL statement, the data type, length, precision, scale, and code page that is independently determined for each of those references must be identical or an error is returned.

Table 29. Untyped Expression Usage in Predicates

Untyped Expression Location	Data Type
Both operands of a comparison operator	VARCHAR(254)
One operand of a comparison operator, when the other operand is other than an untyped expression	The data type of the other operand
All operands of a BETWEEN predicate	VARCHAR(254)
Two operands of a BETWEEEN predicate	Same as that of the only typed expression
Only one operand of a BETWEEN predicate	Result of applying the "Rules for result data types" on all operands that are other than untyped expressions
All operands of an IN predicate, for example, ? IN (?,?,?)	VARCHAR(254)
The first operand of an IN predicate, when the right side is a fullselect, for example, IN (fullselect)	Data type of the selected column
The first operand of an IN predicate, when the right side is not a subselect;, for example, ? IN (?,A,B), or ? IN (A,?,B,?)	Result of applying the "Rules for result data types" on all operands of the IN list (operands to the right of the IN keyword) that are other than untyped expressions
Any or all operands of the IN list of the IN predicate, for example, A IN (?,B, ?)	Result of applying the "Rules for result data types" on all operands of the IN predicate (operands to the left and right of the IN keyword) that are other than untyped expressions
Both the operand in a row-value-expression of an IN predicate, and the corresponding result column of the fullselect, for example, (c1, ?) IN (SELECT c1, ? FROM ...)	VARCHAR(254)
Any operands in a row-value-expression of an IN predicate, for example, (c1,?) IN fullselect	Data type of the corresponding result column of the fullselect
Any select list items in a subquery if a row-value-expression is specified in an IN predicate, for example, (c1,c2) IN (SELECT?, c1, FROM ...)	Data type of the corresponding operand in the row-value-expression

Table 29. Untyped Expression Usage in Predicates (continued)

Untyped Expression Location	Data Type
All three operands of the LIKE predicate	Match expression (operand 1) and pattern expression (operand 2) are VARCHAR(32672); escape expression (operand 3) is VARCHAR(2)
The match expression of the LIKE predicate when either the pattern expression or the escape expression is other than an untyped expression	VARCHAR(32672), VARBINARY(32672), or VARGRAPHIC(16336), depending on the data type of the first operand that is not an untyped expression
The pattern expression of the LIKE predicate when either the match expression or the escape expression is other than an untyped expression	VARCHAR(32672), VARBINARY(32672), or VARGRAPHIC(16336), depending on the data type of the first operand that is not an untyped expression.
The escape expression of the LIKE predicate when either the match expression or the pattern expression is other than an untyped expression	VARCHAR(2), VARBINARY(1), or VARGRAPHIC(1), depending on the data type of the first operand that is not an untyped expression.
Operand of the NULL predicate	VARCHAR(254)

Table 30. Untyped Expression Usage in Built-in Functions

Untyped Parameter Marker Location	Data Type
Array index of an ARRAY	BIGINT
All arguments of COALESCE, when all arguments are untyped parameter markers	Error
Any argument of COALESCE, when at least one argument is not an untyped parameter marker	Result of applying the "Rules for result data types" on all arguments that are other than untyped parameter markers
First argument of DAYNAME	TIMESTAMP(12)
The argument of DIGITS	DECIMAL(31,6)
First argument of FROM_UTC_TIMESTAMP	TIMESTAMP(6)
All arguments of MAX, MIN, or NULLIF, when all arguments are untyped parameter markers	Error
Any argument of MAX, MIN, or NULLIF, when at least one argument is not an untyped parameter marker	Result of applying the "Rules for result data types" on all arguments that are other than untyped parameter markers
First argument of MONTHNAME	TIMESTAMP(12)
POSSTR (both arguments)	Both arguments are VARCHAR(32672)
POSSTR (one argument, when the other argument is a character data type)	VARCHAR(32672)
POSSTR (one argument, when the other argument is a graphic data type)	VARGRAPHIC(16336)
POSSTR (the search-string argument, when the other argument is a binary data type)	VARBINARY(32672)

Table 30. Untyped Expression Usage in Built-in Functions (continued)

Untyped Parameter Marker Location	Data Type
First and second arguments of REGEXP_LIKE, REGEXP_INSTR, REGEXP_SUBSTR, REGEXP_COUNT	VARCHAR(32672)
First argument of REGEXP_LIKE, REGEXP_INSTR, REGEXP_SUBSTR, or REGEXP_COUNT when second argument is not an untyped expression	VARGRAPHIC(16336) if second argument is a graphic data type; VARCHAR(32672) otherwise
Second argument of REGEXP_LIKE, REGEXP_INSTR, REGEXP_SUBSTR, or REGEXP_COUNT when first argument is not an untyped expression	VARGRAPHIC(16336) if first argument is a graphic data type in a Unicode database; VARCHAR(32672) otherwise
First three arguments of REGEXP_REPLACE	VARCHAR(32672)
First argument of REGEXP_REPLACE when second or third argument is not an untyped expression	VARGRAPHIC(16336) if third argument is a graphic data type or third argument is an untyped expression and second argument is a graphic data type; VARCHAR(32672) otherwise
Second argument of REGEXP_REPLACE when first or third argument is not an untyped expression	VARGRAPHIC(16336) if first argument is a graphic data type in a Unicode database; VARCHAR(32672) otherwise
Third argument of REGEXP_REPLACE when first or second argument is not an untyped expression	VARGRAPHIC(16336) if first argument is a graphic data type or first argument is an untyped expression and second argument is a graphic data type; VARCHAR(32672) otherwise
First argument of SUBSTR	VARCHAR(32672)
Second and third argument of SUBSTR	INTEGER
First argument of SUBSTRB	VARCHAR(32672)
First argument of SUBSTR2	VARGRAPHIC(16336) if database supports graphic types; otherwise VARCHAR(32672)
First argument of SUBSTR4	VARCHAR(32672)
Second argument of TIMESTAMP	TIME
First argument of TIMESTAMP_FORMAT	VARCHAR(254)
First argument of TIMEZONE	TIMESTAMP(6)
First argument of TO_UTC_TIMESTAMP	TIMESTAMP(6)
Second and third arguments of TRANSLATE	VARCHAR(32672) if the first argument is a character type; VARGRAPHIC(16336) if the first argument is a graphic type
Fourth argument of TRANSLATE	VARCHAR(1) if the first argument is a character type; VARGRAPHIC(1) if the first argument is a graphic type
Second argument of TRIM_ARRAY	BIGINT
Unary minus	DECFLOAT(34)
Unary plus	DECFLOAT(34)

Table 30. Untyped Expression Usage in Built-in Functions (continued)

Untyped Parameter Marker Location	Data Type
All arguments of VALUE, when all arguments are untyped parameter markers	Error
Any argument of VALUE, when at least one argument is not an untyped parameter marker	Result of applying the "Rules for result data types" on all arguments that are other than untyped parameter markers
First argument of VARCHAR_FORMAT	TIMESTAMP(12)
First argument of XMLCOMMENT	VARCHAR(32672)
All arguments of XMLCONCAT	XML
First argument of XMLDOCUMENT	XML
Arguments of XMLELEMENT or XMLEXISTS	Error
Second argument of XMLPI	VARCHAR(32672)
Arguments of XMLQUERY	Error
First argument of XMLSERIALIZE	XML
Arguments of XMLTABLE	Error
First argument of XMLTEXT	VARCHAR(32672)
First argument of XMLVALIDATE	XML
First argument of XMLXSROBJECTID	XML
Arguments of an aggregate function	Error
All other arguments of all other scalar functions	The data type of the parameter of the function definition as determined by function resolution. The length of the argument is derived based on Table 20 on page 121 in Function Resolution section.

Table 31. Untyped Expression Usage in User-defined Routines

Untyped Parameter Marker Location	Data Type
Argument of a function	The data type and length of the parameter, as defined when the function was created.
Argument of a method	Error
Argument of a procedure	The data type of the parameter, as defined when the procedure was created

Row expression

A row expression specifies a row of data that could have a specific user-defined row type or the built-in data type ROW.

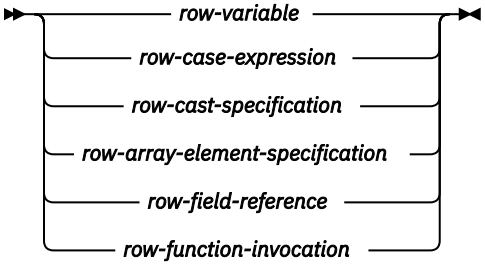
Authorization

The use of some of the row expressions may require having the appropriate authorization. For these row expressions, the privileges held by the authorization ID of the statement must include the following authorization:

- *row-variable.* For information about authorization considerations when *row-variable* is a global variable, see "Global variables".
- *row-function-invocation.* The authorization to execute the function. For information about authorization considerations, see "Function invocation" in the "Functions" topic
- *expression*. Authorizations might be required for the use of certain expressions referenced in a *row-expression*. For information about authorization considerations, see "Expressions".

Syntax

row-expression



Description

row-variable
A variable that is defined with row type.

row-case-expression
A case-expression that returns a row type.

row-cast-specification
A CAST that returns a row type.

row-array-element-specification
An array-element-specification of an array with row type elements.

row-field-reference
A field-reference of a row where the field is also a row type

row-function-invocation
A *function-invocation* of a user-defined function that has a return type that is a row type. The function could return a user-defined row type or the data type ROW with defined field names and field types.

Notes

- Row expressions can be used to generate a row within SQL PL contexts.

Predicates

A *predicate* specifies a condition that is true, false, or unknown about a given value, row, or group.

The following rules apply to all types of predicates:

- All values specified in a predicate must be compatible.
- An expression used in a basic, quantified, IN, or BETWEEN predicate must not result in a character string with a length attribute greater than 4000, a graphic string with a length attribute greater than 2000, or a LOB string of any size.
- The value of a host variable can be null (that is, the variable may have a negative indicator variable).
- The code page conversion of operands of predicates involving two or more operands, with the exception of LIKE, is done according to the rules for string conversions.
- Use of a structured type value is limited to the NULL predicate and the TYPE predicate.

- In a Unicode database, all predicates that accept a character or graphic string will accept any string type for which conversion is supported.

A fullselect is a form of the SELECT statement that, when used in a predicate, is also called a *subquery*.

Row-value-expression

The operand of several predicates (basic, quantified, and IN) can be expressed as a row value expression:

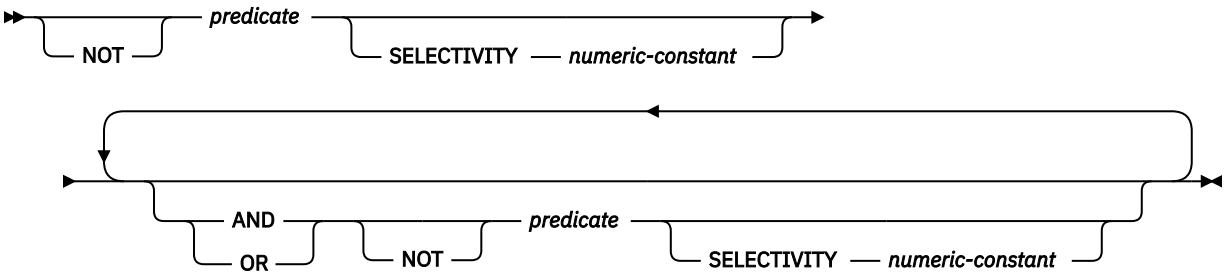
row-value-expression

A row value expression returns a single row that consists of one or more fields. The field values can be specified as a list of expressions. The number of fields that are returned by the row value expression is equal to the number of expressions that are specified in the list.

Search conditions

A *search condition* specifies a condition that is "true," "false," or "unknown" about a given value, row, or group. A search condition can also be a Boolean column, value, or literal.

search-condition



The result of a search condition is derived by application of the specified *logical operators* (AND, OR, NOT) to the result of each specified predicate. If logical operators are not specified, the result of the search condition is the result of the specified predicate.

AND and OR are defined in Table 32 on page 191, in which P and Q are any predicates:

Table 32. Truth Tables for AND and OR

P	Q	P AND Q	P OR Q
True	True	True	True
True	False	False	True
True	Unknown	Unknown	True
False	True	False	True
False	False	False	False
False	Unknown	False	Unknown
Unknown	True	Unknown	True
Unknown	False	False	Unknown
Unknown	Unknown	Unknown	Unknown

NOT(true) is false, NOT(false) is true, and NOT(unknown) is unknown.

Search conditions within parentheses are evaluated first. If the order of evaluation is not specified by parentheses, NOT is applied before AND, and AND is applied before OR. The order in which operators at the same precedence level are evaluated is undefined to allow for optimization of search conditions.

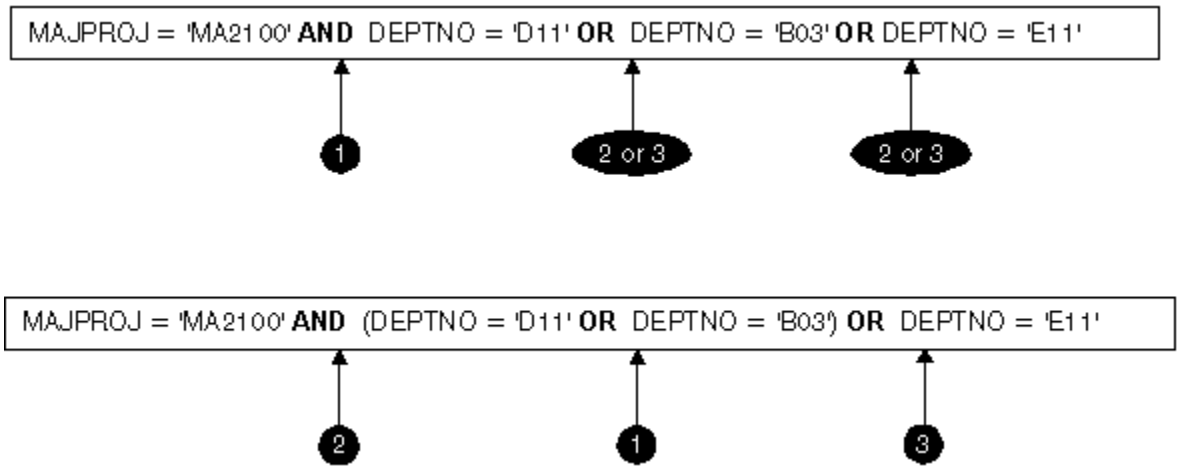


Figure 3. Search Conditions Evaluation Order

SELECTIVITY *numeric-constant*

The SELECTIVITY clause is used to indicate what the expected selectivity percentage is for the predicate. SELECTIVITY can be specified for the following predicates:

- A user-defined predicate, regardless of the *DB2_SELECTIVITY* query compiler registry variable setting
- A basic predicate in which at least one expression contains host variables or parameter markers. Specifying SELECTIVITY for this type of predicate applies only when the *DB2_SELECTIVITY* query compiler registry variable is set to YES.
- Any predicate when the DB2_SELECTIVITY query compiler registry variable is set to ALL.

A user-defined predicate is a predicate that consists of a user-defined function invocation, in the context of a predicate specification that matches the predicate specification on the PREDICATES clause of CREATE FUNCTION. For example, if the function `myfunction` is defined with PREDICATES WHEN=1..., then the following use of SELECTIVITY is valid:

```
SELECT *
  FROM STORES
  WHERE myfunction(parm,parm) = 1 SELECTIVITY 0.004
```

The selectivity value must be a numeric literal value in the inclusive range from 0 to 1 (SQLSTATE 42615). If SELECTIVITY is not specified, the default value is 0.01 (that is, the user-defined predicate is expected to filter out all but one percent of all the rows in the table). The SELECTIVITY default can be changed for any given function by updating its SELECTIVITY column in the SYSSTAT.ROUTINES view. An error will be returned if the SELECTIVITY clause is specified for a non user-defined predicate (SQLSTATE 428E5).

A user-defined function (UDF) can be applied as a user-defined predicate and, hence, is potentially applicable for index exploitation if:

- The predicate specification is present in the CREATE FUNCTION statement
- The UDF is invoked in a WHERE clause being compared (syntactically) in the same way as specified in the predicate specification
- There is no negation (NOT operator)

Examples

In the following query, the `within` UDF specification in the WHERE clause satisfies all three conditions and is considered a user-defined predicate.

```
SELECT *
  FROM customers
  WHERE within(location, :sanJose) = 1 SELECTIVITY  0.2
```

However, the presence of `within` in the following query is not index-exploitable due to negation and is not considered a user-defined predicate.

```
SELECT *
  FROM customers
  WHERE NOT(within(location, :sanJose) = 1) SELECTIVITY  0.3
```

In the next example, consider identifying customers and stores that are within a certain distance of each other. The distance from one store to another is computed by the radius of the city in which the customers live.

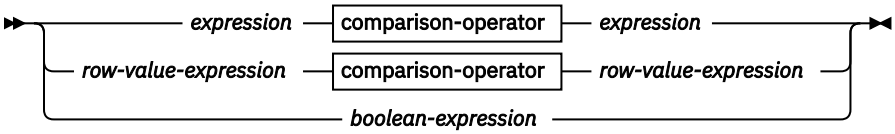
```
SELECT *
  FROM customers, stores
  WHERE distance(customers.loc, stores.loc) <
    CityRadius(stores.loc) SELECTIVITY 0.02
```

In the preceding query, the predicate in the WHERE clause is considered a user-defined predicate. The result produced by CityRadius is used as a search argument to the range producer function.

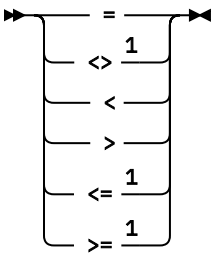
However, since the result produced by CityRadius is used as a range producer function, the user-defined predicate shown previously will not be able to make use of the index extension defined on the stores.loc column. Therefore, the UDF will make use of only the index defined on the customers.loc column.

Basic predicate

A *basic predicate* compares two values or compares a set of values with another set of values.



comparison-operator



Notes:

[1] See "Alternative forms" on page 195.

The six comparison operators can effectively be expressed based on just two of the comparison operators. If the predicate operands are *x* and *y*, then the other four comparison operators can be expressed using the following alternative predicates.

Table 33. Predicates and alternative predicates

Predicate	Alternative predicate
x <> *y*	NOT (*x* = *y*)

Table 33. Predicates and alternative predicates (continued)	
Predicate	**Alternative predicate**
x > *y*	*y* < *x*
x <= *y*	*x* < *y* OR *x* = *y*
x >= *y*	*y* < *x* OR *x* = *y*

When the predicate operands are specified as an *expression*, the data types of the expressions must be comparable. If the value of either operand is null, the result of the predicate is unknown. Otherwise the result is either true or false.

Table 34. Predicate evaluation with scalar operands		
Predicate (with operand values *x* and *y*)	**Boolean value**	**If and Only If...**
x = *y*	is True	*x* is equal to *y*
x < *y*	is True	*x* is less than *y*
x = *y*	is False	*x* is not equal to *y*
x < *y*	is False	*x* = *y* is True or *y* < *x* is True

When the predicate operands are specified as a *row-value-expression*, they must have the same number of fields and the data types of the corresponding fields of the operands must be comparable. The result of the comparison is based on comparisons of the corresponding fields in the *row-value-expression* operands.

Table 35. Predicate evaluation with row operands		
Predicate (with operand values *Rx* and *Ry* that have fields *Rxi* and *Ryi* where 0 < *i* < *number of fields*)	**Boolean value**	**If and Only If...**
Rx = *Ry*	is True	All pairs of corresponding value expressions are equal (*Rxi* = *Ryi* is True for all values of *i*).
Rx < *Ry*	is True	The first *N* pairs of corresponding value expressions are equal and the next pair has the left value expression less than the right value expression for some value of *N* (*Rxi* = *Ryi* is True for all values of *i* < *n* and *Rxn* < *Ryn* is True for some value of *n*).
Rx = *Ry*	is False	At least one pair of corresponding value expressions are not equal (NOT (*Rxi* = *Ryi*) is True for some value of *i*).

Table 35. Predicate evaluation with row operands (continued)		
Predicate (with operand values *Rx* and *Ry* that have fields *Rxi* and *Ryi* where 0 < *i* < *number of fields*)	**Boolean value**	**If and Only If...**
Rx < *Ry*	is False	All pairs of corresponding value expressions are equal (*Rx* = *Ry* is True) or the first *N* pairs of corresponding value expressions are equal and the next pair has the right value expression less than the left value expression for some value of *N* (*Rxi* = *Ryi* is True for all values of *i* < *n* and *Ryn* < *Rxn* is True for some value of *n*).
Rx comparison operator Ry	is Unknown	The comparison is neither True nor False.

Boolean values

You can use a basic predicate to compare a Boolean value with another Boolean value or with a value of a data type that can be cast to a Boolean value. A value of TRUE is greater than a value of FALSE. For example:

- `TRUE = 'on'` is TRUE
- `DECFLOAT(4.3) = TRUE` is TRUE
- `'0' <= FALSE` is TRUE
- `'yes' <= FALSE` is FALSE

Alternative forms

For the comparison operators <>, <=, and >=, alternative forms are also supported. (The forms ¬=, ¬<, and ¬> are supported only in code pages 437, 819, and 850.) Support for these alternative forms is intended only to accommodate existing SQL statements; these forms are not recommended for new SQL statements.

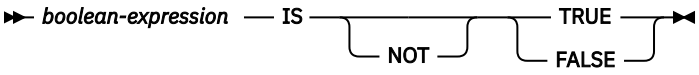
Comparison Operator	Alternative Forms
<>	^= != ¬=
<=	^> !> ¬>
>=	^< !< ¬<

Examples:

```
  EMPNO='528671'
  SALARY < 20000
  PRSTAFF <> :VAR1
  SALARY > (SELECT AVG(SALARY) FROM EMPLOYEE)
(YEARVAL, MONTHVAL) >= (2009, 10)
```

Boolean predicate

A *Boolean predicate* returns the truth value of a Boolean expression.



boolean-expression
An expression that returns the Boolean value (or representation of a Boolean value) that is to be evaluated by the function. The data type of the returned value must be of one of the following data types (SQLSTATE 42884):

- BOOLEAN
- SMALLINT, INTEGER, BIGINT (binary integer)
- DECFLOAT (floating decimal)
- CHAR or VARCHAR (string)
- GRAPHIC or VARGRAPHIC (graphic; applies only to a Unicode database)

Non-Boolean values are implicitly cast to Boolean values as described in "Boolean values" on page 41.

If the returned value is a string, it must be a valid representation of a Boolean value as described in "Boolean values" on page 41 (SQLSTATE 22018).

Null values are treated differently by different predicates:

- The IS TRUE predicate returns:
  - TRUE when the truth value of the value returned by input expression is TRUE
  - FALSE when the truth value is FALSE or NULL
- The IS FALSE predicate returns:
  - TRUE when the truth value of the value returned by input expression is FALSE
  - FALSE when the truth value is TRUE or NULL
- The IS NOT TRUE predicate returns:
  - TRUE when the truth value of the value returned by input expression is FALSE or NULL
  - FALSE when the truth value is TRUE
- The IS NOT FALSE predicate returns
  - TRUE when the truth value of the value returned by input expression is TRUE or NULL
  - FALSE when the truth value is FALSE

Alternative syntax: The keyword ON can be used instead of TRUE; OFF can be used instead of FALSE.

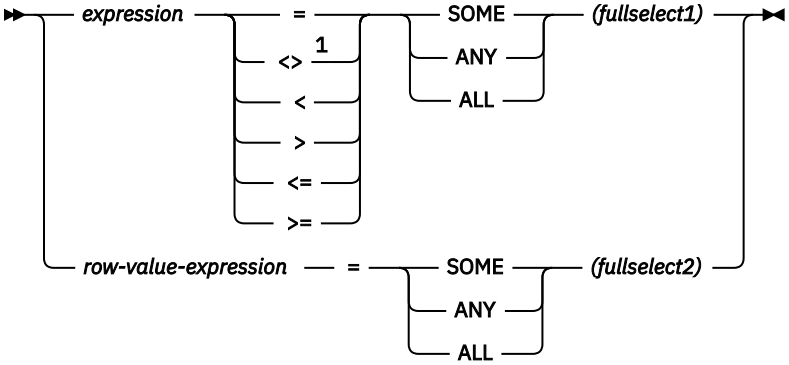
Example

The following statement returns those values from COLA that are not equal to the corresponding value in COLX.

```
SELECT COLA FROM TBLAB WHERE COLA = COLX IS NOT TRUE
```

Quantified predicate

A *quantified predicate* compares a value or values with a collection of values.



Notes:

[1] The following forms of the comparison operators are also supported in basic and quantified predicates: ^=, ^<, ^>, !=, !<, and !>. In code pages 437, 819, and 850, the forms ¬=, ¬<, and ¬> are supported. All of these product-specific forms of the comparison operators are intended only to support existing SQL statements that use these operators, and are not recommended for use when writing new SQL statements.

The fullselect must identify a number of columns that is the same as the number of expressions specified to the left of the predicate operator (SQLSTATE 428C4). The fullselect may return any number of rows.

When ALL is specified:

- The result of the predicate is true if the fullselect returns no values or if the specified relationship is true for every value returned by the fullselect.
- The result is false if the specified relationship is false for at least one value returned by the fullselect.
- The result is unknown if the specified relationship is not false for any values returned by the fullselect and at least one comparison is unknown because of the null value.

When SOME or ANY is specified:

- The result of the predicate is true if the specified relationship is true for each value of at least one row returned by the fullselect.
- The result is false if the fullselect returns no rows or if the specified relationship is false for at least one value of every row returned by the fullselect.
- The result is unknown if the specified relationship is not true for any of the rows and at least one comparison is unknown because of a null value.

Examples: Use the following tables when referring to the following examples.

TBL***AB***:

COL***A***	COL***B***
1	12
2	12
3	13
4	14
-	-

TBL***XY***:

COL***X***	COL***Y***
2	22
3	23

Figure 4. Tables for quantified predicate examples

Example 1

```
SELECT COLA FROM TBLAB
   WHERE COLA = ANY(SELECT COLX FROM TBLXY)
```

Results in 2,3. The subselect returns (2,3). COLA in rows 2 and 3 equals at least one of these values.

Example 2

```
SELECT COLA FROM TBLAB
   WHERE COLA > ANY(SELECT COLX FROM TBLXY)
```

Results in 3,4. The subselect returns (2,3). COLA in rows 3 and 4 is greater than at least one of these values.

Example 3

```
SELECT COLA FROM TBLAB
   WHERE COLA > ALL(SELECT COLX FROM TBLXY)
```

Results in 4. The subselect returns (2,3). COLA in row 4 is the only one that is greater than both these values.

Example 4

```
SELECT COLA FROM TBLAB
   WHERE COLA > ALL(SELECT COLX FROM TBLXY
                    WHERE COLX<0)
```

Results in 1,2,3,4, null. The subselect returns no values. Thus, the predicate is true for all rows in TBLAB.

Example 5

```
SELECT * FROM TBLAB
   WHERE (COLA,COLB+10) = SOME (SELECT COLX, COLY FROM TBLXY)
```

The subselect returns all entries from TBLXY. The predicate is true for the subselect, hence the result is as follows:

```
COLA        COLB
----------- -----------
          2          12
          3          13
```

Example 6

```
SELECT * FROM TBLAB
   WHERE (COLA,COLB) = ANY (SELECT COLX,COLY-10 FROM TBLXY)
```

The subselect returns COLX and COLY-10 from TBLXY. The predicate is true for the subselect, hence the result is as follows:

```
COLA        COLB
----------- -----------
          2          12
          3          13
```

ARRAY_EXISTS

The ARRAY_EXISTS predicate tests for the existence of an array index in an array.

array-expression
An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

array-index
The data type of *array-index* must be assignable to the data type of the array index of the array. If *array-expression* is an ordinary array, then *array-index* must be assignable to INTEGER (SQLSTATE 428H1).

The result is true if *array-expression* includes an array index that is equal to *array-index*cast to the data type of the array index of *array-expression*; otherwise the result is false.

The result cannot be unknown; if either argument is null, the result is false.

Example

1. Assume that array variableRECENT_CALLS is defined as an ordinary array of array type PHONENUMBERS. The following IF statement tests if the recent calls list has reached the 40th saved call yet. If it has, the local Boolean variable EIGHTY_PERCENT is set to true:

```
IF (ARRAY_EXISTS(RECENT_CALLS, 40)) THEN
  SET EIGHTY_PERCENT = TRUE;
END IF
```

BETWEEN predicate

The BETWEEN predicate compares a value with a range of values.

If the data types of the operands are not the same, all values are converted to the data type that would result by applying the "Rules for result data types", except if the data types of all the operands are numeric, in which case no values are converted.

The BETWEEN predicate:

```
value1 BETWEEN value2 AND value3
```

is equivalent to the search condition:

```
value1 >= value2 AND value1 <= value3
```

The BETWEEN predicate:

```
value1 NOT BETWEEN value2 AND value3
```

is equivalent to the search condition:

```
NOT(value1 BETWEEN value2 AND value3); that is,
value1 < value2 OR value1 > value3.
```

The first operand (expression) cannot include a function that is not deterministic or has an external action (SQLSTATE 42845).

Examples

Example 1

```
EMPLOYEE.SALARY BETWEEN 20000 AND 40000
```

Results in all salaries between $20,000.00 and $40,000.00.

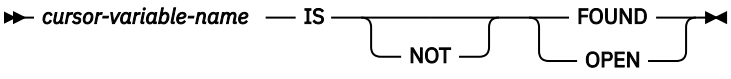
Example 2

```
SALARY NOT BETWEEN 20000 + :HV1 AND 40000
```

Assuming :HV1 is 5000, results in all salaries below $25,000.00 and above $40,000.00.

Cursor predicates

Cursor predicates are SQL keywords that can be used to determine the state of a cursor defined within the current scope. They provide a means for easily referencing whether a cursor is open, closed or if there are rows associated with the cursor.



cursor-variable-name
The name of a SQL variable or SQL parameter of a cursor type.

IS
Specifies that a cursor predicate property is to be tested.

NOT
Specifies that the opposite value of testing the cursor predicate property is to be returned.

FOUND
Specifies to check if the cursor contains rows after the execution of a FETCH statement. If the last FETCH statement executed was successful, and if the IS FOUND predicate syntax is used, the returned value is TRUE. If the last FETCH statement executed resulted in a condition where rows were not found, the result is false. The result is unknown when:

- the value of cursor-variable-name is null
- the underlying cursor of cursor-variable-name is not open
- the predicate is evaluated before the first FETCH action was performed on the underlying cursor
- the last FETCH action returned an error

The IS FOUND predicate can be useful within a portion of SQL PL logic that loops and performs a fetch with each iteration. The predicate can be used to determine if rows remain to be fetched. It provides an efficient alternative to using a condition handler that checks for the error condition that is raised when no more rows remain to be fetched.

When the NOT keyword is specified, so that the syntax is IS NOT FOUND, the result value is the opposite.

OPEN
Specifies to check if the cursor is in an open state. If the cursor is in open and if the IS OPEN predicate syntax is used, the returned value is TRUE. This can be a useful predicate in cases where cursors are passed as parameters to functions and procedures. Before attempting to open the cursor, this predicate can be used to determine if the cursor is already open.

When the NOT keyword is specified, so that the syntax is IS NOT OPEN, the result value is the opposite.

Notes

- A cursor predicate can only be used in statements within a compound SQL (compiled) statement (SQLSTATE 42818).

Example

The following script defines an SQL procedure that contains references to these predicates as well as the prerequisite objects required to successfully compile and call the procedure:

```
CREATE TABLE T1 (c1 INT, c2 INT, c3 INT)@

INSERT INTO T1 VALUES (1,1,1),(2,2,2),(3,3,3) @

CREATE TYPE myRowType AS ROW(c1 INT, c2 INT, c3 INT)@

CREATE TYPE myCursorType AS myRowType CURSOR@


CREATE PROCEDURE p(OUT count INT)
LANGUAGE SQL
BEGIN
  DECLARE C1 CURSOR;
  DECLARE lvarInt INT;

  SET count = -1;
  SET c1 = CURSOR FOR SELECT c1 FROM t1;

  IF (c1 IS NOT OPEN) THEN
     OPEN c1;
  ELSE
     set count = -2;
  END IF;

  SET count = 0;
  IF (c1 IS OPEN) THEN

    FETCH c1 INTO lvarInt;

    WHILE (c1 IS FOUND) DO
      SET count = count + 1;
      FETCH c1 INTO lvarInt;
    END WHILE;
  ELSE
     SET COUNT = 0;
  END IF;

END@

CALL p()@
```

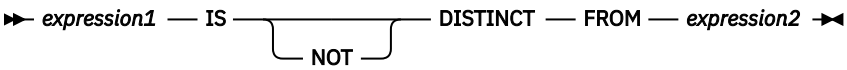
DISTINCT predicate

The DISTINCT predicate compares two expressions and evaluates to TRUE if their values are not identical.

The result of a DISTINCT predicate depends on whether either or both of its input expressions are null:

Input expressions	IS DISTINCT FROM	IS NOT DISTINCT FROM
Both inputs are non-null.	Evaluates to TRUE if the inputs are not identical and FALSE if they are. Equivalent to the <> operator.	Evaluates to FALSE if the inputs are not identical and TRUE if they are. Equivalent to the = operator.
One input is null.	Evaluates to TRUE.	Evaluates to FALSE.
Both inputs are null.	Evaluates to FALSE.	Evaluates to TRUE.

The result of a DISTINCT predicate cannot be null.



expression1* and *expression2
The expressions that are to be compared.

Examples

Assume that HV is a host variable and T1 is a table with one column (C1) and three rows:

```
C1
----
1
2
NULL
```

- If HV=2, the statement

  ```
  SELECT * FROM T1 WHERE C1 IS DISTINCT FROM :HV;
  ```

  returns rows 1 and 3.

  If HV=2, the statement

  ```
  SELECT * FROM T1 WHERE C1 IS NOT DISTINCT FROM :HV;
  ```

  returns row 2.
- If HV=NULL, the statement

  ```
  SELECT * FROM T1 WHERE C1 IS DISTINCT FROM :HV;
  ```

  returns rows 1 and 2.

  If HV=NULL, the statement

  ```
  SELECT * FROM T1 WHERE C1 IS NOT DISTINCT FROM :HV;
  ```

  returns row 3.

EXISTS predicate

The EXISTS predicate tests for the existence of certain rows.

The fullselect may specify any number of columns, and

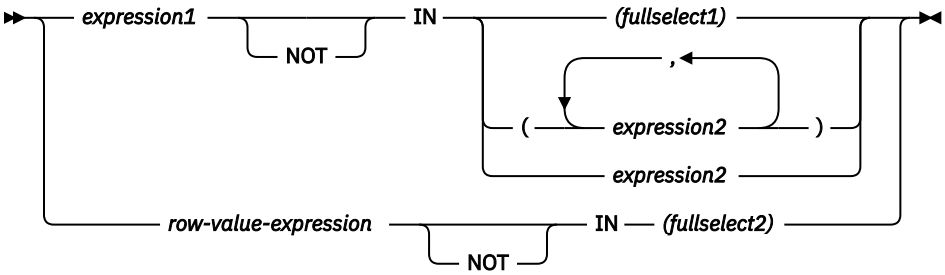
- The result is true only if the number of rows specified by the fullselect is not zero.
- The result is false only if the number of rows specified is zero
- The result cannot be unknown.

Example

```
   EXISTS (SELECT * FROM TEMPL WHERE SALARY < 10000)
```

IN predicate

The IN predicate compares a value or values with a collection of values.



The fullselect must identify a number of columns that is the same as the number of expressions specified to the left of the IN keyword (SQLSTATE 428C4). The fullselect may return any number of rows.

- An IN predicate of the form:

  ```
  expression IN expression
  ```

  is equivalent to a basic predicate of the form:

  ```
  expression = expression
  ```

- An IN predicate of the form:

  ```
  expression IN (fullselect)
  ```

  is equivalent to a quantified predicate of the form:

  ```
  expression = ANY (fullselect)
  ```

- An IN predicate of the form:

  ```
  expression NOT IN (fullselect)
  ```

  is equivalent to a quantified predicate of the form:

  ```
  expression <> ALL (fullselect)
  ```

- An IN predicate of the form:

  ```
  expression IN (expressiona, expressionb, ..., expressionk)
  ```

  is equivalent to:

  ```
  expression = ANY (fullselect)
  ```

  where fullselect in the values-clause form is:

  ```
  VALUES (expressiona), (expressionb), ..., (expressionk)
  ```

- An IN predicate of the form:

  ```
  (expressiona, expressionb,..., expressionk) IN (fullselect)
  ```

  is equivalent to a quantified predicate of the form:

  ```
  (expressiona, expressionb,..., expressionk) = ANY (fullselect)
  ```

  Note that the operand on the left side of this form of these predicates is referred to as a *row-value-expression*.

The values for *expression1* and *expression2* or the column of *fullselect1* in the IN predicate must be compatible. Each field of the *row-value-expression* and its corresponding column of *fullselect2* in the IN predicate must be compatible. The rules for result data types can be used to determine the attributes of the result used in the comparison.

The values for the expressions in the IN predicate (including corresponding columns of a fullselect) can have different code pages. If a conversion is necessary, the code page is determined by applying rules for string conversions to the IN list first, and then to the predicate, using the derived code page for the IN list as the second operand.

Examples

Example 1: The following condition evaluates to true if the value in the row under evaluation in the DEPTNO column contains D01, B01, or C01:

```
DEPTNO IN ('D01', 'B01', 'C01')
```

Example 2: The following condition evaluates to true only if the EMPNO (employee number) on the left side matches the EMPNO of an employee in department E11:

```
EMPNO IN (SELECT EMPNO FROM EMPLOYEE WHERE WORKDEPT = 'E11')
```

Example 3: Given the following information, this example evaluates to true if the specific value in the row of the COL_1 column matches any of the values in the list:

Table 36. IN Predicate example

Expressions	Type	Code Page
COL_1	column	850
HV_2	host variable	437
HV_3	host variable	437
CON_1	constant	850

When evaluating the predicate:

```
COL_1 IN (:HV_2, :HV_3, CON_4)
```

the two host variables will be converted to code page 850, based on the rules for string conversions.

Example 4: The following condition evaluates to true if the specified year in EMENDATE (the date an employee activity on a project ended) matches any of the values specified in the list (the current year or the two previous years):

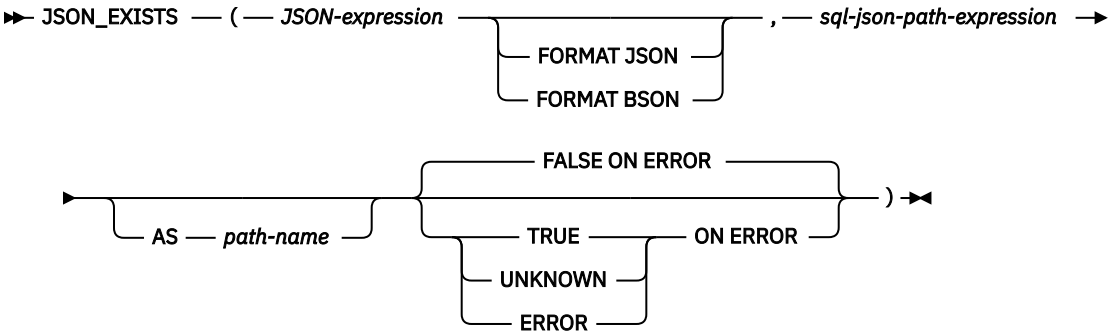
```
YEAR(EMENDATE) IN (YEAR(CURRENT DATE),
                   YEAR(CURRENT DATE - 1 YEAR),
                   YEAR(CURRENT DATE - 2 YEARS))
```

Example 5: The following condition evaluates to true if both ID and DEPT on the left side match MANAGER and DEPTNUMB respectively for any row of the ORG table.

```
(ID, DEPT) IN (SELECT MANAGER, DEPTNUMB FROM ORG)
```

JSON_EXISTS predicate

The JSON_EXISTS predicate determines whether JSON data contains a JSON value that can be located by using the specified *sql-json-path-expression*.



The result of the JSON_EXISTS predicate is true if at least one value can be located in *JSON-expression* by using *sql-json-path-expression*.

If *sql-json-path-expression* uses strict mode and an error occurs, the result of the predicate is determined by the ON ERROR clause.

The result of the JSON_EXISTS predicate is UNKNOWN if *JSON-expression* is the null value.

JSON-expression
An expression that returns a value that is a built-in string data type, except the following data types (SQLSTATE 42815):

- GRAPHIC
- VARGRAPHIC
- DBCLOB
- BINARY
- CHAR FOR BIT DATA
- VARCHAR FOR BIT DATA
- A user-defined type that is sourced on any of the previously listed data types

If a character value is returned, it must contain correctly formatted JSON data (SQLSTATE 22032). If a binary data type is returned, it is interpreted according to the explicit or implicit FORMAT clause.

FORMAT JSON
JSON-expression is formatted as JSON data.

If *JSON-expression* is a character string data type, it is treated as JSON data.

If *JSON-expression* is a binary string data type, it is interpreted as UTF-8 data.

FORMAT BSON
Specifies that *JSON-expression* is formatted as the BSON representation of JSON data (SQLSTATE 22032). *JSON-expression* must be a binary string data type (SQLSTATE 42815).

sql-json-path-expression
An expression that returns a value that is a built-in character string data type. The string is interpreted as an SQL/JSON path expression and is used to locate a JSON value within the JSON data that is specified by *JSON-expression*. For more information about the SQL/JSON path expression, see “sql-json-path-expression” on page 179.

AS *path-name*
Specifies a name to be used to identify *sql-json-path-expression*.

ON ERROR
Specifies the behavior when an error is encountered by JSON_EXISTS.

FALSE ON ERROR
The result is false if an error is encountered. This clause is the default.

TRUE ON ERROR
The result is true if an error is encountered.

UNKNOWN ON ERROR
The result is unknown if an error is encountered.

ERROR ON ERROR
An error is returned if an error is encountered.

Notes

- If parameter markers are not explicitly cast to a supported data type, an error is returned (SQLSTATE 42815)

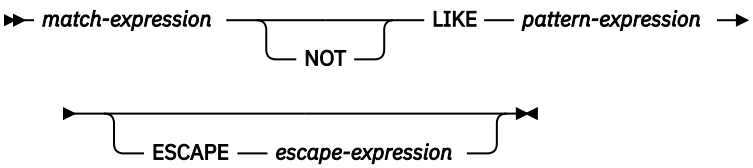
Example

1. Return rows for employees who do not have an emergency contact in their JSON_DATA column. COALESCE causes null values to be treated as an empty string. The FALSE ON ERROR clause is used so all rows that do not contain an emergency value are returned.

```
SELECT empno, lastname FROM employee
  WHERE NOT JSON_EXISTS(COALESCE(JSON_DATA, ''), 'strict $.emergency' FALSE ON ERROR);
```

LIKE predicate

The LIKE predicate searches for strings that have a certain pattern. The pattern is specified by a string in which the underscore and the percent sign may have special meanings. Trailing blanks in a pattern are part of the pattern.



If the value of any of the arguments is null, the result of the LIKE predicate is unknown.

The values for *match-expression*, *pattern-expression*, and *escape-expression* are compatible string expressions. There are slight differences in the types of string expressions supported for each of the arguments. The valid types of expressions are listed under the description of each argument.

None of the expressions can yield a distinct type. However, it can be a function that casts a distinct type to its source type.

match-expression
An expression that specifies the string that is to be examined to see if it conforms to a certain pattern of characters.

LIKE *pattern-expression*
An expression that specifies the string that is to be matched.

The expression can be specified in the same way as *match-expression* with the following restrictions:

- No element in the expression can be of type CLOB or DBCLOB. In addition it cannot be a BLOB file reference variable.
- The actual length of *pattern-expression* cannot be more than 32 672 bytes.

A **simple description** of the use of the LIKE predicate is that the pattern is used to specify the conformance criteria for values in the *match-expression*, where:

- The underscore character (_) represents any single character.

- The percent sign (%) represents a string of zero or more characters.
- Any other character represents itself.

If the *pattern-expression* needs to include either the underscore or the percent character, the *escape-expression* is used to specify a character to precede either the underscore or the percent character in the pattern.

A **rigorous description** of the use of the LIKE predicate follows. Note that this description ignores the use of the *escape-expression*; its use is covered later.

- Let *m* denote the value of *match-expression* and let *p* denote the value of *pattern-expression*. The string *p* is interpreted as a sequence of the minimum number of substring specifiers so each character of *p* is part of exactly one substring specifier. A substring specifier is an underscore, a percent sign, or any non-empty sequence of characters other than an underscore or a percent sign.

  The result of the predicate is unknown if *m* or *p* is the null value. Otherwise, the result is either true or false. The result is true if *m* and *p* are both empty strings or there exists a partitioning of *m* into substrings such that:

  – A substring of *m* is a sequence of zero or more contiguous characters and each character of *m* is part of exactly one substring.
  – If the *n*th substring specifier is an underscore, the *n*th substring of *m* is any single character.
  – If the *n*th substring specifier is a percent sign, the *n*th substring of *m* is any sequence of zero or more characters.
  – If the *n*th substring specifier is neither an underscore nor a percent sign, the *n*th substring of *m* is equal to that substring specifier and has the same length as that substring specifier.
  – The number of substrings of *m* is the same as the number of substring specifiers.

  Thus, if *p* is an empty string and *m* is not an empty string, the result is false. Similarly, it follows that if *m* is an empty string and *p* is not an empty string (except for a string containing only percent signs), the result is false.

  The predicate *m* NOT LIKE *p* is equivalent to the search condition NOT (*m* LIKE *p*).

When the *escape-expression* is specified, the *pattern-expression* must not contain the escape character identified by the *escape-expression*, except when immediately followed by the escape character, the underscore character, or the percent sign character (SQLSTATE 22025).

If the *match-expression* is a character string in an MBCS database, it can contain mixed data. In this case, the pattern can include both SBCS and non-SBCS characters. For non-Unicode databases, the special characters in the pattern are interpreted as follows:

- An SBCS halfwidth underscore refers to one SBCS character.
- A non-SBCS fullwidth underscore refers to one non-SBCS character.
- An SBCS halfwidth or non-SBCS fullwidth percent sign refers to zero or more SBCS or non-SBCS characters.

In a Unicode database, there is really no distinction between "single-byte" and "non-single-byte" characters. Although the UTF-8 format is a "mixed-byte" encoding of Unicode characters, there is no real distinction between SBCS and non-SBCS characters in UTF-8. Every character is a Unicode character, regardless of the number of bytes in UTF-8 format.

In a Unicode graphic column, every non-supplementary character, including the halfwidth underscore character (U&'\005F') and the halfwidth percent sign character (U&'\0025'), is two bytes in width. In a Unicode database, special characters in a pattern are interpreted as follows:

- For character strings, a halfwidth underscore character (X'5F') or a fullwidth underscore character (X'EFBCBF') refers to one Unicode character, and a halfwidth percent sign character (X'25') or a fullwidth percent sign character (X'EFBC85') refers to zero or more Unicode characters.
- For graphic strings, a halfwidth underscore character (U&'\005F') or a fullwidth underscore character (U&'\FF3F') refers to one Unicode character, and a halfwidth percent sign character

(U&'\0025') or a fullwidth percent sign character (U&'\FF05') refers to zero or more Unicode characters.

- To be recognized as special characters when a locale-sensitive UCA-based collation is in effect, the underscore character and the percent sign character must not be followed by non-spacing combining marks (diacritics). For example, the pattern U&'%\0300' (percent sign character followed by non-spacing combining grave accent) will be interpreted as a search for % and not as a search for zero or more Unicode characters followed by a letter with a grave accent.

A Unicode supplementary character is stored as two graphic code points in a Unicode graphic column. To match a Unicode supplementary character in a Unicode graphic column, use one underscore if the database uses locale-sensitive UCA-based collation, and two underscores otherwise. To match a Unicode supplementary character in a Unicode character column, use one underscore for all collations. To match a base character with one or more trailing non-spacing combining characters, use one underscore if the database uses locale-sensitive UCA-based collation. Otherwise, use as many underscore characters as the number of non-spacing combining characters plus the base character.

escape-expression

This optional argument is an expression that specifies a character to be used to modify the special meaning of the underscore (_) and percent (%) characters in the *pattern-expression*. This allows the LIKE predicate to be used to match values that contain the actual percent and underscore characters.

The expression can be specified by any one of:

- A constant
- A special register
- A global variable
- A host variable
- A scalar function with any of the previously mentioned operands
- An expression concatenating any of the previously listed items

with the restrictions that:

- No element in the expression can be of type CLOB or DBCLOB. In addition, it cannot be a BLOB file reference variable.
- For character columns, the result of the expression must be one character, or a binary string containing exactly one byte (SQLSTATE 22019).
- For graphic columns, the result of the expression must be one character (SQLSTATE 22019).
- The result of the expression must not be a non-spacing combining character sequence (for example, U&'\0301', Combining Acute Accent).

When escape characters are present in the pattern string, an underscore, percent sign, or escape character can represent a literal occurrence of itself. This is true if the character in question is preceded by an odd number of successive escape characters. It is not true otherwise.

In a pattern, a sequence of successive escape characters is treated as follows:

- Let S be such a sequence, and suppose that S is not part of a larger sequence of successive escape characters. Suppose also that S contains a total of n characters. Then the rules governing S depend on the value of n:
  - If n is odd, S must be followed by an underscore or percent sign (SQLSTATE 22025). S and the character that follows it represent (n-1)/2 literal occurrences of the escape character followed by a literal occurrence of the underscore or percent sign.
  - If n is even, S represents n/2 literal occurrences of the escape character. Unlike the case where n is odd, S could end the pattern. If it does not end the pattern, it can be followed by any character (except, of course, an escape character, which would violate the assumption that S is not part of a larger sequence of successive escape characters). If S is followed by an underscore or percent sign, that character has its special meaning.

Following is an illustration of the effect of successive occurrences of the escape character which, in this case, is the back slash (\).

Pattern string
Actual Pattern

\%
A percent sign

\\%
A back slash followed by zero or more arbitrary characters

\\\%
A back slash followed by a percent sign

The code page used in the comparison is based on the code page of the *match-expression* value.

- The *match-expression* value is never converted.
- If the code page of *pattern-expression* is different from the code page of *match-expression*, the value of *pattern-expression* is converted to the code page of *match-expression*, unless either operand is defined as FOR BIT DATA or is a binary type (in which case there is no conversion).
- If the code page of *escape-expression* is different from the code page of *match-expression*, the value of *escape-expression* is converted to the code page of *match-expression*, unless either operand is defined as FOR BIT DATA or is a binary type (in which case there is no conversion).

Notes

- The number of trailing blanks is significant in both the *match-expression* and the *pattern-expression*. If the strings are not the same length, the shorter string is not padded with blank spaces. For example, the expression `'PADDED ' LIKE 'PADDED'` would not result in a match.
- If the pattern specified in a LIKE predicate is a parameter marker, and a fixed-length character host variable is used to replace the parameter marker, the value specified for the host variable must have the correct length. If the correct length is not specified, the select operation will not return the intended results.

  For example, if the host variable is defined as CHAR(10), and the value WYSE% is assigned to that host variable, the host variable is padded with blanks on assignment. The pattern used is:

  ```
  'WYSE%     '
  ```

  The database manager searches for all values that start with WYSE and that end with five blank spaces. If you want to search only for values that start with "WYSE", assign a value of `WSYE%%%%%` to the host variable.
- The pattern is matched using the collation of the database, unless either operand is defined as FOR BIT DATA, in which case the pattern is matched using a binary comparison.

Examples

- Search for the string "SYSTEMS" appearing anywhere within the PROJNAME column in the PROJECT table.

  ```
  SELECT PROJNAME FROM PROJECT
    WHERE PROJECT.PROJNAME LIKE '%SYSTEMS%'
  ```

- Search for a string with a first character of "J" that is exactly two characters long in the FIRSTNME column of the EMPLOYEE table.

  ```
  SELECT FIRSTNME FROM EMPLOYEE
    WHERE EMPLOYEE.FIRSTNME LIKE 'J_'
  ```

- Search for a string of any length, with a first character of "J", in the FIRSTNME column of the EMPLOYEE table.

```
SELECT FIRSTNME FROM EMPLOYEE
  WHERE EMPLOYEE.FIRSTNME LIKE 'J%'
```

- In the CORP_SERVERS table, search for a string in the LA_SERVERS column that matches the value in the CURRENT SERVER special register.

```
SELECT LA_SERVERS FROM CORP_SERVERS
  WHERE CORP_SERVERS.LA_SERVERS LIKE CURRENT SERVER
```

- Retrieve all strings that begin with the character sequence "_\" in column A of table T.

```
SELECT A FROM T
  WHERE T.A LIKE '\_\\%' ESCAPE '\'
```

- Use a binary string constant to specify a one-byte escape character that is compatible with the match and pattern data types (both BLOBs).

```
SELECT COLBLOB FROM TABLET
  WHERE COLBLOB LIKE :pattern_var ESCAPE BX'0E'
```

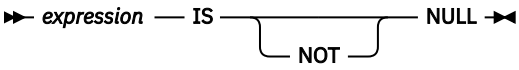
- In a Unicode database defined with the case insensitive collation CLDR181_LEN_S1, find all names that start with "Bill".

```
SELECT NAME FROM CUSTDATA WHERE NAME LIKE 'Bill%'
```

  This query returns the names "Bill Smith", "billy simon", and "BILL JONES".

NULL predicate

The NULL predicate tests for null values.



The result of a NULL predicate cannot be unknown. If the value of the expression is null, the result is true. If the value is not null, the result is false. If NOT is specified, the result is reversed.

A row type value cannot be used as the operand in a NULL predicate, except as the qualifier of a field name. If *expression* is a row type, an error is returned (SQLSTATE 428H2).

For compatibility with other SQL dialects, you can use ISNULL as a synonym for IS NULL and NOTNULL as a synonym for IS NOT NULL.

Examples

```
PHONENO IS NULL

SALARY IS NOT NULL
```

OVERLAPS predicate

The OVERLAPS predicate determines whether two chronological periods overlap. A chronological period is specified by a pair of date-time expressions (the first expression specifies the start of a period; the second specifies its end).

Each of the date-time expressions (*start1*, *end1*, *start2*, and *end2*) must return a value that is a DATE, TIME, or TIMESTAMP:

- If *start1* returns a DATE or TIMESTAMP value, the other expressions must all return either a DATE or TIMESTAMP value; otherwise, error SQL0401N is returned. The default time associated with a returned DATE value is 00:00.
- If *start1* returns a TIME value, the other expressions must all return a TIME value; otherwise, error SQL0401N is returned.

The begin and end values are not included in the periods. For example, the periods **2016-10-19 to 2016-10-20** and **2016-10-20 to 2016-10-21** do not overlap.

If none of the expressions return a null value, the OVERLAPS predicate returns **true** if the time periods overlap and **false** if they do not. If one or more of the date-time expressions returns a null value, that influences the result:

- If one of the expressions returns a null value, the result is true if the other value for that period falls within the other period; otherwise, the result is NULL.
- If two or more of the expressions return a null value, the result is NULL.

Examples

The following statement returns rows because the OVERLAPS predicate is true (the period 17-21 March 2016 overlaps the period 20-22 March 2016):

```
SELECT * from T1 where (cast('2016-03-17' as DATE),
      cast('2016-03-21' as DATE)) OVERLAPS
      (cast('2016-03-20' as DATE), cast('2016-03-22' as DATE));
```

The following statement does not return rows because the OVERLAPS predicate is false (the begin and end values are not included in the periods, so the period 19-20 October 2016 does not overlap the period 20-21 October 2016):

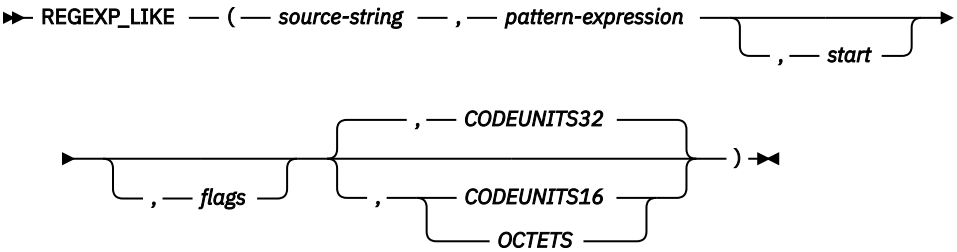
```
SELECT * from T1 where (cast('2016-10-19' as DATE),
      cast('2016-10-20' as DATE)) OVERLAPS
      (cast('2016-10-20' as DATE), cast('2016-10-21' as DATE));
```

The following statement returns rows because the OVERLAPS predicate is true (the date 22 March 2016 is within the period 20-23 March 2016):

```
SELECT * from T1 where (cast('2016-10-22' as DATE),NULL) OVERLAPS
      (cast('2016-10-20' as DATE), cast('2016-10-23' as DATE));
```

REGEXP_LIKE predicate

The REGEXP_LIKE predicate searches for a regular expression pattern in a string.



If the *pattern-expression* is found, the result is true. If the pattern-expression is not found, the result is false. If the value of any of the arguments is null, the result of the REGEXP_LIKE predicate is unknown.

source-string
An expression that specifies the string in which the search is to take place. This expression must return a built-in character string, graphic string, numeric value, Boolean value, or datetime value. A numeric, Boolean, or datetime value is implicitly cast to VARCHAR before the function is evaluated. A character string cannot specify the FOR BIT DATA attribute (SQLSTATE 42815).

pattern-expression
An expression that specifies the regular expression string that is the pattern for the search. This expression must return a built-in character string, graphic string, numeric value, Boolean value, or datetime value. A numeric, Boolean, or datetime value is implicitly cast to VARCHAR before the function is evaluated. The length of a CLOB or DBCLOB expression must not be greater than the maximum length of a VARCHAR or VARGRAPHIC data type. A character string cannot specify the FOR BIT DATA attribute (SQLSTATE 42815).

start
An expression that specifies the position within *source-string* at which the search is to start. The expression must return a built-in character string, graphic string, Boolean, or numeric value. If the value is not of type INTEGER, it is implicitly cast to INTEGER before the function is evaluated. The value of the integer must be greater than or equal to 1. If OCTETS is specified and the source string is graphic data, the value of the integer must be odd (SQLSTATE 428GC). The default start value is 1. See parameter description for CODEUNITS16, CODEUNITS32, or OCTETS for the string unit that applies to the start position.

flags
An expression that specifies flags that controls aspects of the pattern matching. The expression must return a built-in character string that does not specify the FOR BIT DATA attribute (SQLSTATE 42815). The string can include one or more valid flag values and the combination of flag values must be valid (SQLSTATE 2201T). An empty string is the same as the value 'c'. The default flag value is 'c'.

Table 37. Supported flag values

Flag value	Description
c	Specifies that matching is case-sensitive. This flag is the default value if 'c' or 'i' is not specified. This value must not be specified with a value of 'i'.
i	Specifies that matching is case insensitive. This value must not be specified with a value of 'c'.
m	Specifies that the input data can contain more than one line. By default, the '^' in a pattern matches only the start of the input string; the '$' in a pattern matches only the end of the input string. If this flag is set, "^" and "$" also matches at the start and end of each line within the input string.
n	Specifies that the '.' character in a pattern matches a line terminator in the input string. By default, the '.' character in a pattern does not match a line terminator. A carriage-return and line-feed pair in the input string behaves as a single-line terminator, and matches a single "." in a pattern.
s	Specifies that the '.' character in a pattern matches a line terminator in the input string. This value is a synonym for the 'n' value.
x	Specifies that white space characters in a pattern are ignored, unless escaped.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of the start value:

- CODEUNITS16 specifies that the start value is expressed in 16-bit UTF-16 code units.
- CODEUNITS32 specifies that the start value is expressed in 32-bit UTF-32 code units. This is the default.
- OCTETS specifies that the start value is expressed in bytes.

If the string unit is specified as CODEUNITS16 or OCTETS, and if the string unit of the source string is CODEUNITS32, an error is returned (SQLSTATE 428GC).

For more information, see "String units in built-in functions" in “Character strings” on page 31.

Notes

- The regular expression processing is done by using the International Components for Unicode (ICU) regular expression interface.
- **Considerations for non-Unicode databases**:
  - A regular expression pattern supports only half-width control characters; use a character string data type for the pattern expression argument. A character string data type can be used for the pattern expression argument even when a graphic string data type is used for the source string argument.
  - The source string argument must be a graphic string data type if the pattern expression argument is a graphic string data type.

Examples

1. Select the employee number where the last name is spelled LUCCHESSI, LUCHESSI, or LUCHESI from the EMPLOYEE table without considering upper or lower case letters.

   ```
   SELECT EMPNO FROM EMPLOYEE
      WHERE REGEXP_LIKE(LASTNAME,'luc+?hes+?i','i')
   ```

   The result is 1 row with EMPNO value '000110'.
2. Select any invalid product identifier values from the PRODUCT table. The expected format is 'nnn-nnn-nn' where 'n' is a digit 0 - 9.

   ```
   SELECT PID FROM PRODUCT
      WHERE NOT REGEXP_LIKE(pid,'[0-9]{3}-[0-9]{3}-[0-9]{2}')
   ```

   The result is 0 rows because all the product identifiers match the pattern.

Trigger event predicates

A trigger event predicate is used in a triggered action to test the event that activated the trigger. It is only valid in the triggered action of a compiled trigger definition (SQLSTATE 42601).

DELETING
True if the trigger was activated by a delete operation. False otherwise.

INSERTING
True if the trigger was activated by an insert operation. False otherwise.

UPDATING
True if the trigger was activated by an update operation. False otherwise.

Notes

- Trigger event predicates can be used anywhere in the triggered action of a CREATE TRIGGER statement that uses a compound SQL (compiled) statement as the *SQL-procedure-statement*. In other contexts the keywords will not be recognized and will attempt to resolve as column or variable names.

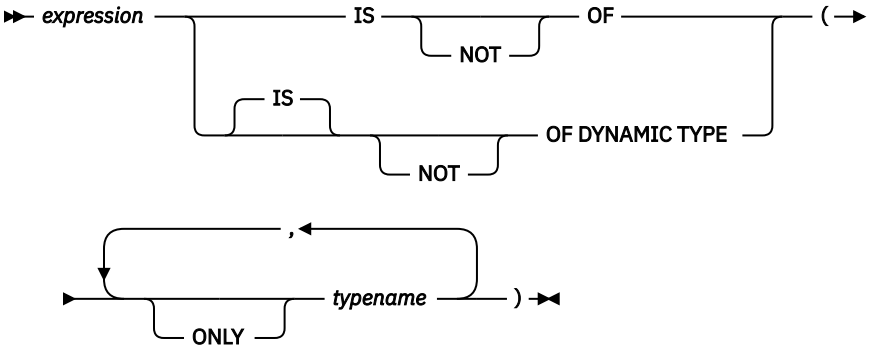
Example

The following trigger increments the number of employees each time a new person is hired (that is, each time a new row is inserted into the EMPLOYEE table); decrements the number of employees each time an employee leaves the company; and raises an error when an update occurs that would result in a salary increase greater than ten percent of the current salary, by using trigger event predicates in its conditions:

```
CREATE TRIGGER HIRED
  AFTER INSERT OR DELETE OR UPDATE OF SALARY ON EMPLOYEE
  REFERENCING NEW AS N OLD AS O FOR EACH ROW
    BEGIN
      IF INSERTING
      THEN UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
      END IF;

      IF DELETING
      THEN UPDATE COMPANY_STATS SET NBEMP = NBEMP - 1;
      END IF;

      IF (UPDATING AND (N.SALARY > 1.1 * O.SALARY))
      THEN SIGNAL SQLSTATE '75000' SET MESSAGE_TEXT = 'Salary increase>10%'
      END IF;
    END;
```

TYPE predicate

A *TYPE predicate* compares the type of an expression with one or more user-defined structured types.



The dynamic type of an expression involving the dereferencing of a reference type is the actual type of the referenced row from the target typed table or view. This might differ from the target type of an expression involving the reference, which is called the static type of the expression.

If the value of *expression* is null, the result of the predicate is unknown. The result of the predicate is true if the dynamic type of the *expression* is a subtype of one of the structured types specified by *typename*, otherwise the result is false. If ONLY precedes any *typename* the proper subtypes of that type are not considered.

If *typename* is not qualified, it is resolved using the SQL path. Each *typename* must identify a user-defined type that is in the type hierarchy of the static type of *expression* (SQLSTATE 428DU).

The DEREF function should be used whenever the TYPE predicate has an expression involving a reference type value. The static type for this form of *expression* is the target type of the reference.

The syntax IS OF and OF DYNAMIC TYPE are equivalent alternatives for the TYPE predicate. Similarly, IS NOT OF and NOT OF DYNAMIC TYPE are equivalent alternatives.

Examples

A table hierarchy exists with root table EMPLOYEE of type EMP and subtable MANAGER of type MGR. Another table, ACTIVITIES, includes a column called WHO_RESPONSIBLE that is defined as REF(EMP) SCOPE EMPLOYEE. The following example shows a type predicate that evaluates to true when a row corresponding to WHO_RESPONSIBLE is a manager:

```
DEREF (WHO_RESPONSIBLE) IS OF (MGR)
```

If a table contains a column EMPLOYEE of type EMP, EMPLOYEE may contain values of type EMP as well as values of its subtypes like MGR. The following predicate

```
EMPL IS OF (MGR)
```

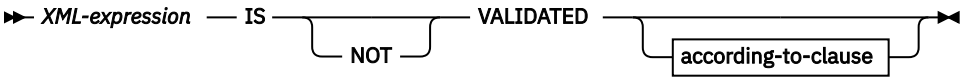
returns true when EMPL is not null and is actually a manager.

VALIDATED predicate

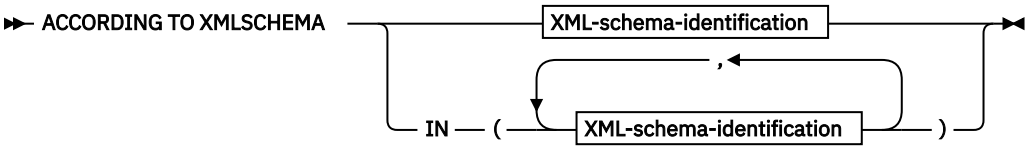
The VALIDATED predicate tests whether or not the value specified by *XML-expression* has been validated using the XMLVALIDATE function.

If the value specified is null, the result of the validation constraint is unknown; otherwise, the result of the validation constraint is either true or false. The value you specify must be of type XML.

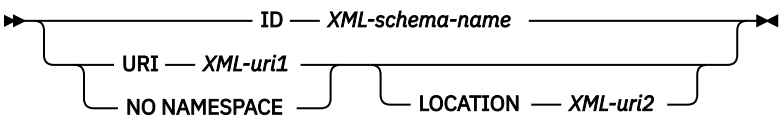
If the ACCORDING TO XMLSCHEMA clause is not specified, then XML schemas used for validation do not impact the result of the validation constraint.



according-to-clause



XML-schema-identification



Description

XML-expression

Specifies the XML value tested, where *XML-expression* can consist of an XML document, XML content, a sequence of XML nodes, an XML *column-name*, or an XML *correlation-name*.

If an XML *column-name* is specified, the predicate evaluates whether or not XML documents associated with the specified column name have been validated.

See "CREATE TRIGGER" for information about specifying correlation names of type XML as part of triggers.

IS VALIDATED or IS NOT VALIDATED

Specifies the required validation state for the *XML-expression* operand.

For a constraint that specifies IS VALIDATED to evaluate as true, the operand must have been validated. If an optional ACCORDING TO XMLSCHEMA clause includes one or several XML schemas, the operand must have been validated using one of the identified XML schemas.

For a constraint that specifies IS NOT VALIDATED to evaluate as false, the operand must be in an validated state. If an optional ACCORDING TO XMLSCHEMA clause includes one or several XML schemas, the operand must have been validated using one of the identified XML schemas.

according-to-clause

Specifies one or several XML schemas against which the operand must or must not have been validated. Only XML schemas previously registered with the XML schema repository may be specified.

ACCORDING TO XMLSCHEMA

ID *XML-schema-name*

Specifies an SQL identifier for the XML schema. The name, including the implicit or explicit SQL schema qualifier, must uniquely identify an existing XML schema in the XML schema repository at the current server. If no XML schema by this name exists in the implicitly or explicitly specified SQL schema, an error is returned (SQLSTATE 42704).

URI *XML-uri1*
Specifies the target namespace URI of the XML schema. The value of *XML-uri1* specifies a URI as a character string constant that is not empty. The URI must be the target namespace of a registered XML schema (SQLSTATE 4274A) and, if no LOCATION clause is specified, it must uniquely identify the registered XML schema (SQLSTATE 4274B).

NO NAMESPACE
Specifies that the XML schema has no target namespace. The target namespace URI is equivalent to an empty character string that cannot be specified as an explicit target namespace URI.

LOCATION *XML-uri2*
Specifies the XML schema location URI of the XML schema. The value of *XML-uri2* specifies a URI as a character string constant that is not empty. The XML schema location URI, combined with the target namespace URI, must identify a registered XML schema (SQLSTATE 4274A), and there must be only one such XML schema registered (SQLSTATE 4274B).

Examples

Example 1: Assume that column XMLCOL is defined in table T1. Retrieve only the XML values that have been validated by any XML schema.

```
SELECT XMLCOL FROM T1
  WHERE XMLCOL IS VALIDATED
```

Example 2: Assume that column XMLCOL is defined in table T1. Enforce the rule that values cannot be inserted or updated unless they have been validated.

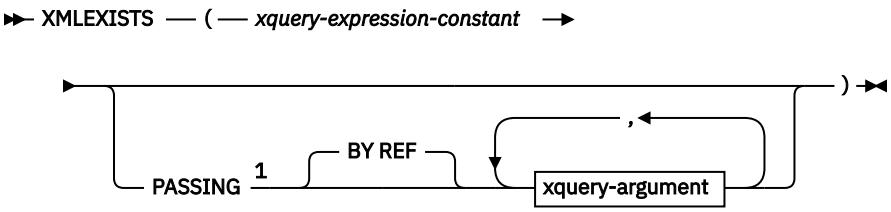
```
ALTER TABLE T1 ADD CONSTRAINT CK_VALIDATED
  CHECK (XMLCOL IS VALIDATED)
```

Example 3: Assume that you want to select only those rows from table T1 with XML column XMLCOL that have been validated with the XML schema URI http://www.posample.org.

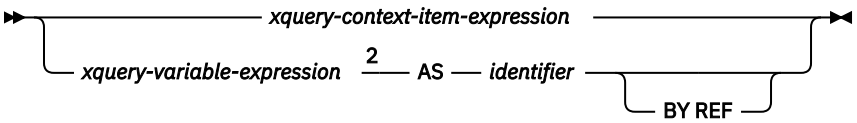
```
SELECT XMLCOL FROM T1
  WHERE XMLCOL IS VALIDATED
   ACCORDING TO XMLSCHEMA URI
   'http://www.posample.org'
```

XMLEXISTS predicate

The XMLEXISTS predicate tests whether an XQuery expression returns a sequence of one or more items.



xquery-argument



Notes:

[1] The data type cannot be DECFLOAT.
[2] The data type of the expression cannot be DECFLOAT.

xquery-expression-constant
Specifies an SQL character string constant that is interpreted as an XQuery expression. The constant string is converted directly to UTF-8 without conversion to the database or section code page. The XQuery expression executes using an optional set of input XML values, and returns an output sequence that is tested to determine the result of the XMLEXISTS predicate. The value for *xquery-expression-constant* must not be an empty string or a string of blank characters (SQLSTATE 10505).

PASSING
Specifies input values and the manner in which these values are passed to the XQuery expression specified by *xquery-expression-constant*. By default, every unique column name that is in the scope where the function is invoked is implicitly passed to the XQuery expression using the name of the column as the variable name. If an *identifier* in a specified xquery-argument matches an in-scope column name, then the explicit xquery-argument is passed to the XQuery expression overriding that implicit column.

BY REF
Specifies that the default passing mechanism is by reference for any *xquery-variable-expression* of data type XML. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any, directly from the specified input expressions, preserving all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons involving some nodes contained between the two input arguments might refer to nodes within the same XML node tree.

This clause has no impact on how non-XML values are passed. The non-XML values create a new copy of the value during the cast to XML.

xquery-argument
Specifies an argument that is to be passed to the XQuery expression specified by *row-xquery-expression-constant*. The method through which *row-xquery-argument* is used in the XQuery expression depends on whether the argument is specified as an *xquery-context-item-expression* or *an xquery-variable-expression*.

- If the resulting value is of type XML, it becomes an *input-xml-value*. A null XML value is converted to an XML empty sequence.
- If the resulting value is not of type XML, it must be castable to the XML data type. A null value is converted to an XML empty sequence. The converted value becomes an *input-xml-value*.

When the *xquery-expression-constant* is evaluated, an XQuery variable is presented with a value equal to *input-xml-value* and a name specified by the AS clause.

xquery-context-item-expression
xquery-context-item-expression specifies the initial context item in the XQuery expression specified by *xquery-expression-constant*. The value of the initial context item is the result of *xquery-context-item-expression* cast to XML. *xquery-context-item-expression* must not be specified more than one time.

xquery-context-item-expression must not be a sequence of more than one item. If the result of *xquery-context-item-expression* is an empty string, the XQuery expression is evaluated with the initial context item set to an XML empty string.

If the *xquery-context-item-expression* is not specified or is an empty string, the initial context item in the XQuery expression is undefined, and the XQuery expression must not reference the initial context item. An XQuery variable is not created for the context item expression.

If the *xquery-context-expression* is not specified or the *input-xml-value* that results from the *xquery-context-expression* is an XML empty sequence, the initial context item is undefined. If the XQuery expression refers to the initial context item, it must be specified with a value that is not an XML empty sequence.

xquery-variable-expression
Specifies an SQL expression whose value is available to the XQuery expression specified by *xquery-expression-constant* during execution. The expression cannot contain a sequence reference (SQLSTATE 428F9) or an OLAP function (SQLSTATE 42903). The data type of the expression cannot be DECFLOAT.

AS *identifier*
Specifies that the value generated by *xquery-variable-expression* will be passed to *xquery-expression-constant* as an XQuery variable. The variable name will be *identifier*. The leading dollar sign ($) that precedes variable names in the XQuery language is not included in *identifier*. The identifier must be a valid XQuery variable name and is restricted to an XML NCName. The identifier must not be greater than 128 bytes in length. Two arguments within the same PASSING clause cannot use the same identifier (SQLSTATE 42711).

BY REF
Indicates that an XML input value is to be passed by reference. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any, directly from the specified input expressions, preserving all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons involving some nodes contained between the two input arguments might refer to nodes within the same XML node tree. If BY REF is not specified following an *xquery-variable-expression*, XML arguments are passed by way of the default passing mechanism that is provided through the syntax that follows the PASSING keyword. This option cannot be specified for non-XML values. When a non-XML value is passed, the value is converted to XML; this process creates a copy.

Notes

The XMLEXISTS predicate cannot be:

- Part of the ON clause that is associated with a JOIN operator or a MERGE statement (SQLSTATE 42972)
- Part of the GENERATE KEY USING or RANGE THROUGH clause in the CREATE INDEX EXTENSION statement (SQLSTATE 428E3)
- Part of the FILTER USING clause in the CREATE FUNCTION (External Scalar) statement, or the FILTER USING clause in the CREATE INDEX EXTENSION statement (SQLSTATE 428E4)
- Part of a check constraint or a column generation expression (SQLSTATE 42621)
- Part of a group-by-clause (SQLSTATE 42822)
- Part of an argument for an aggregate function (SQLSTATE 42607)

An XMLEXISTS predicate that involves a subquery might be restricted by statements that restrict subqueries.

Example

```
SELECT c.cid FROM customer c
  WHERE XMLEXISTS('$d/*:customerinfo/*:addr[ *:city = "Aurora" ]'
    PASSING info AS "d")
```

Built-in global variables

Built-in global variables are provided with the database manager and are used in SQL statements to retrieve scalar values associated with the variables.

As an example, the ROUTINE_TYPE global variable is referenced in an SQL statement to retrieve the current routine type.

For most built-in global variables, the authorization ID of any statement that retrieves the value of the global variable is required to have the READ privilege on the global variable or DATAACCESS authority. However, there are exceptions where an authorization ID with other database authorities also has access to the global variable. Exceptions to the authorization required to retrieve the value of a global variable are specified within the descriptions of the built-in global variables.

In a non-restrictive database, the READ privilege is granted to PUBLIC for most built-in global variables at creation time. Exceptions to granting this privilege are specified within the descriptions of the built-in global variables.

Examples

- To access the global variable CLIENT_HOST, run the following query:

```
VALUES SYSIBM.CLIENT_HOST
```

  This query returns the host name of the current client:

```
1
----------
hotellnx93
```

  An alternative way to call the global variable is to use it in a SELECT statement:

```
SELECT C1, C2
  FROM T1
  WHERE C3 = CLIENT_HOST
```

- For read/write global variables, set the value with the "SET variable" statement:

```
SET NLS_STRING_UNITS = 'CODEUNITS32'
```

CLIENT_HOST global variable

This built-in global variable contains the host name of the current client, as returned by the operating system.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.
- The type is VARCHAR(255).
- The schema is SYSIBM.
- The scope of this global variable is session.

If the client connection originated from an application running on the local system, the value of the variable is NULL. The database manager obtains the client IP address from the network when the connection is accepted. The client host name is obtained from the client IP address by invoking the TCP/IP `GetAddrInfo` function. If the processes did not originate from a remote system using TCP/IP, the value of the variable is NULL.

CLIENT_IPADDR global variable

This built-in global variable contains the IP address of the current client, as returned by the operating system.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.
- The type is VARCHAR(128).
- The schema is SYSIBM.
- The scope of this global variable is session.

The value of the CLIENT_IPADDR global variable is NULL if the client did not connect by using the TCP/IP or SSL protocol.

CLIENT_ORIGUSERID global variable

This built-in global variable contains the original user identifier, as supplied by an application, usually from a multiple-tier server environment.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.

- The type is VARCHAR(1024).
- The schema is SYSIBM.
- The scope of this global variable is session.

If an application does not supply a value, the value of the CLIENT_ORIGUSERID global variable is NULL.

The value that is used for CLIENT_ORIGUSERID can be set with the following APIs:

- the originalUser variable of the getDB2Connection (trusted reuse) and reuseDB2Connection (trusted connection reuse) APIs of the IBM® Data Server Driver for JDBC and SQLJ.

Notes

No privileges are granted to PUBLIC when the CLIENT_ORIGUSERID global variable is created.

CLIENT_USRSECTOKEN global variable

This built-in global variable contains a security token, as supplied by an application, usually from a multiple-tier server environment.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.
- The type is BLOB(4K).
- The schema is SYSIBM.
- The scope of this global variable is session.

If an application does not supply a value, the value of the CLIENT_USRSECTOKEN global variable is NULL.

The value that is used for CLIENT_USERSECTOKEN can be set with the following APIs:

- the userSecToken variable of the getDB2Connection (trusted reuse) and reuseDB2Connection (trusted connection reuse) APIs of the IBM® Data Server Driver for JDBC and SQLJ.

Notes

No privileges are granted to PUBLIC when the CLIENT_USRSECTOKEN global variable is created.

MON_INTERVAL_ID global variable

This built-in global variable contains the identifier for the current monitoring interval.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.
- The type is BIGINT.
- The schema is SYSIBM.
- The scope of this global variable is database.

The MON_INTERVAL_ID global variable facilitates the collection and aggregation of monitoring data by external monitoring applications. The value of this global variable is intended to be set by monitoring tools. However, you can increment the value by using the MON_INCREMENT_INTERVAL_ID procedure.

The value of the MON_INTERVAL_ID global variable is 0 if there is no current monitoring interval.

The **mon_interval_id** monitor element contains the value of the monitor interval ID at the time that the monitoring data was captured. You can use the value of the **mon_interval_id** monitor element to correlate data that you gather over a specific monitoring interval.

Notes

In addition to the typical authorization groups or IDs that can read built-in global variables, an authorization ID with either DBADM or SQLADM authority has read access to the MON_INTERVAL_ID global variable.

NLS_STRING_UNITS global variable

This built-in global variable specifies the default string units that are used when defining character and graphic data types in a Unicode database.

This global variable has the following characteristics:

- It is a read/write variable, with values maintained by the user.
- The type is VARCHAR(11 OCTETS).
- The schema is SYSIBM.
- The scope of this global variable is session.
- It has a default value of NULL.

This global variable is only applicable to a Unicode database and must have a value of NULL, 'SYSTEM', or 'CODEUNITS32' (SQLSTATE 42815).

NULL
The STRING_UNITS database configuration parameter is used to determine the default string units

SYSTEM
CHAR, VARCHAR, and CLOB data types defined without specifying the CODEUNITS32 keyword will default to OCTETS.

GRAPHIC, VARGRAPHIC, and DBCLOB data types defined without specifying the CODEUNITS32 keyword will default to CODEUNITS16.

CODEUNITS32
CHAR, VARCHAR, and CLOB data types defined without specifying the OCTETS keyword will default to CODEUNITS32.

GRAPHIC, VARGRAPHIC, and DBCLOB data types defined without specifying the CODEUNITS16 keyword will default to CODEUNITS32.

PACKAGE_NAME global variable

This built-in global variable contains the name of the currently executing package.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.
- The type is VARCHAR(128).
- The schema is SYSIBM.
- The scope of this global variable is session.

In a nested execution scenario, where one package invokes another, the PACKAGE_NAME global variable contains the name of the immediate package context. For example, if package A checks the value of the PACKAGE_NAME variable, the value is A. If package A invokes package B and package B checks the value of the PACKAGE_NAME variable, the value is B.

PACKAGE_SCHEMA global variable

This built-in global variable contains the schema name of the currently executing package.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.

- The type is VARCHAR(128).
- The schema is SYSIBM.
- The scope of this global variable is session.

In a nested execution scenario, where one package invokes another, the PACKAGE_SCHEMA global variable contains the schema name of the immediate package context. For example, if package X.A checks the value of the PACKAGE_SCHEMA variable, the schema value is X. If package X.A invokes package Y.B and package B checks the value of the PACKAGE_SCHEMA variable, the schema value is Y.

PACKAGE_VERSION global variable

This built-in global variable contains the version identifier of the currently executing package.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.
- The type is VARCHAR(64).
- The schema is SYSIBM.
- The scope is session.

If the current executing package does not have a version identifier, the value is NULL.

In a nested execution scenario, where one package invokes another, the PACKAGE_VERSION global variable contains the version identifier of the immediate package context. For example, if package A checks the value of the PACKAGE_VERSION variable, then the value is 1.0. If package A invokes package B and package B checks the value of the PACKAGE_VERSION variable, then the value is 1.8.

ROUTINE_MODULE global variable

This built-in global variable contains the module name of the currently executing routine.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.
- The type is VARCHAR(128).
- The schema is SYSIBM.
- The scope of this global variable is session.

If the currently executing routine does not belong to a module or if the variable is referenced outside a routine execution context, the value of the ROUTINE_MODULE global variable is NULL.

Notes

The value of the ROUTINE_MODULE global variable is set only for procedures and compiled functions: the value always reflects the name of the currently executing routine.

The value does not change for inline functions or methods: the value remains the same as it was when the inline function or method was invoked.

ROUTINE_SCHEMA global variable

This built-in global variable contains the schema name of the currently executing routine.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.
- The type is VARCHAR(128).
- The schema is SYSIBM.
- The scope of this global variable is session.

If the ROUTINE_SCHEMA global variable is referenced outside a routine execution context, the value of the variable is NULL.

Notes

The value of the ROUTINE_SCHEMA global variable is set only for procedures and compiled functions: the value always reflects the schema name of the currently executing routine.

The value does not change for inline functions or methods: the value remains the same as it was when the inline function or method was invoked.

ROUTINE_SPECIFIC_NAME global variable

This built-in global variable contains the specific name of the currently executing routine.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.
- The type is VARCHAR(128).
- The schema is SYSIBM.
- The scope of this global variable is session.

If the ROUTINE_SPECIFIC_NAME global variable is referenced outside a routine execution context, the value of the variable is NULL.

Notes

The value of the ROUTINE_SPECIFIC_NAME global variable is set only for procedures and compiled functions: the value always reflects the specific name of the currently executing routine.

The value is not changed for inline functions or methods: the value remains the same as it was when the inline function or method was invoked.

ROUTINE_TYPE global variable

This built-in global variable contains the type of the currently executing routine.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.
- The type is CHAR(1).
- The schema is SYSIBM.
- The scope of this global variable is session.

If the ROUTINE_TYPE global variable is referenced outside a routine execution context, the value of the variable is NULL.

The value of the ROUTINE_TYPE global variable is P for a stored procedure and F for a function.

Notes

The value of the ROUTINE_TYPE global variable is set only for procedures and compiled functions: the value always reflects the type of the currently executing routine.

The value does not change for inline functions or methods: the value remains the same as it was when the inline function or method was invoked.

SQL_COMPAT global variable

This built-in global variable specifies the SQL compatibility mode. Its value determines which set of syntax rules are applied to SQL queries.

This global variable has the following characteristics:

- It is a read/write variable, with values maintained by the user.
- The type is VARCHAR(3).
- The schema is SYSIBM.
- The scope of this global variable is session.
- It has a default value of NULL.

This global variable must have a value of NULL, 'DB2', or 'NPS' (SQLSTATE 42815).

NULL
The default setting ('DB2') is used.

'DB2'
Db2 syntax rules are applied to SQL queries.

'NPS'
Netezza syntax rules are applied to SQL queries. If you use this setting, some SQL behavior will differ from what is documented in the SQL reference information. To determine the potential effects of a compatibility feature on your SQL applications, see Compatibility features for Netezza .

TRUSTED_CONTEXT global variable

This built-in global variable contains the name of the trusted context that was matched to establish the current trusted connection.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.
- The type is VARCHAR(128).
- The schema is SYSIBM.
- The scope is session.

If no trusted connection is established, the value of the TRUSTED_CONTEXT global variable is NULL.

Built-in functions

Built-in functions are functions provided with the database manager and are classified as aggregate functions, scalar functions, or table functions.

This topic lists the supported built-in functions classified by type:

- Aggregate functions (Table 38 on page 225)
- Array functions (Table 39 on page 227)
- Cast scalar functions (Table 40 on page 227)
- Datetime scalar functions (Table 41 on page 228)
- JSON scalar functions (Table 42 on page 231)
- Miscellaneous scalar functions (Table 43 on page 232)
- Numeric scalar functions (Table 44 on page 233)
- Partitioning scalar functions (Table 45 on page 235)
- Regular expression functions (Table 46 on page 235)
- Security scalar functions (Table 47 on page 235)
- String scalar functions (Table 48 on page 236)

- Table functions (Table 49 on page 238)
- XML functions (Table 50 on page 239)

The “OLAP specification” on page 163 topic documents the following OLAP functions which are sometimes referred to as built-in functions:

- FIRST_VALUE and LAST_VALUE
- LAG and LEAD
- NTILE
- RANK and DENSE_RANK
- RATIO_TO_REPORT
- ROW_NUMBER

There are additional built-in functions documented under the following headings:

- ADMIN_CMD procedure and associated SQL routines
- Audit routines and procedures
- Configuration SQL routines and views
- Db2 pureScale instance views
- Environment views
- Explain routines
- Monitor routines
- MQSeries® SQL routines
- Security SQL routines and views
- Snapshot SQL routines and views
- SQL procedures SQL routines
- Workload management routines
- Miscellaneous SQL routines and views

For details about these additional built-in functions, see "Supported built-in SQL routines and views" in *Administrative Routines and Views*.

Table 38. Aggregate functions

Function	**Description**
“ARRAY_AGG ” on page 240	Aggregates a set of elements into an array.
“AVG ” on page 244	Returns the average of a set of numbers.
“CORRELATION ” on page 245	Returns the coefficient of correlation of a set of number pairs.
“COUNT ” on page 246	Returns the number of rows or values in a set of rows or values.
“COUNT_BIG ” on page 247	Returns the number of rows or values in a set of rows or values. The result can be greater than the maximum value of INTEGER.
“COVARIANCE ” on page 248	Returns the covariance of a set of number pairs.
“COVARIANCE_SAMP ” on page 249	Returns the sample covariance of a set of number pairs.
“CUME_DIST ” on page 250	Returns the cumulative distribution of a row that is hypothetically inserted into a group of rows.

Table 38. Aggregate functions (continued)	
Function	**Description**
“GROUPING ” on page 251	Used with grouping-sets and super-groups to indicate sub-total rows generated by a grouping set. The value returned is 0 or 1. A value of 1 means that the value of the argument in the returned row is a null value, and the row was generated for a grouping set. This generated row provides a sub-total for a grouping set.
“LISTAGG ” on page 255	Aggregates a set string elements into one string by concatenating the strings.
“MAX ” on page 257	Returns the maximum value in a set of values.
“MEDIAN ” on page 258	Returns the median value in a set of values.
“MIN ” on page 259	Returns the minimum value in a set of values.
“PERCENTILE_CONT ” on page 260	Returns the value that corresponds to the specified percentile, given a sort specification by using a continuous distribution model.
“PERCENTILE_DISC ” on page 261	Returns the value that corresponds to the specified percentile given a sort specification by using a discrete distribution model.
“PERCENT_RANK ” on page 262	Returns the relative percentile rank of a row that is hypothetically inserted into a group of rows.
“Regression functions (REGR_AVGX, REGR_AVGY, REGR_COUNT, ...)” on page 264	The regression functions fit an ordinary-least-squares regression line of the form y = a * x + b to a set of number pairs: • REGR_AVGX returns quantities used to compute diagnostic statistics. • REGR_AVGY returns quantities used to compute diagnostic statistics. • REGR_COUNT returns the number of non-null number pairs used to fit the regression line. • REGR_INTERCEPT or REGR_ICPT returns the y-intercept of the regression line. • REGR_R2 returns the coefficient of determination for the regression. • REGR_SLOPE returns the slope of the line. • REGR_SXX returns quantities used to compute diagnostic statistics. • REGR_SXY returns quantities used to compute diagnostic statistics. • REGR_SYY returns quantities used to compute diagnostic statistics.
“STDDEV ” on page 266	Returns the biased standard deviation (division by *n*) of a set of numbers.
“STDDEV_SAMP ” on page 267	Returns the sample standard deviation (division by [*n*-1]) of a set of numbers.
“SUM ” on page 268	Returns the sum of a set of numbers.
“VARIANCE ” on page 269	Returns the biased variance (division by *n*) of a set of numbers.
“VARIANCE_SAMP ” on page 270	Returns the sample variance (division by [*n*-1]) of a set of numbers.
“XMLAGG ” on page 271	Returns an XML sequence containing an item for each non-null value in a set of XML values.
“XMLGROUP ” on page 273	Returns an XML value with a single XQuery document node containing one top-level element node.

Table 39. Array functions	
Function	**Description**
“ARRAY_AGG ” on page 240	Aggregates a set of elements into an array.
“ARRAY_DELETE ” on page 284	Deletes an element or range of elements from an associative array.
“ARRAY_FIRST ” on page 285	Returns the smallest array index value of the array.
“ARRAY_LAST ” on page 286	Returns the largest array index value of the array.
“ARRAY_NEXT ” on page 286	Returns the next larger array index value for an array relative to the specified array index argument.
“ARRAY_PRIOR ” on page 287	Returns the next smaller array index value for an array relative to the specified array index argument.
“CARDINALITY ” on page 298	Returns a value of type BIGINT representing the number of elements of an array
“MAX_CARDINALITY ” on page 419	Returns a value of type BIGINT representing the maximum number of elements that an array can contain.
“TRIM_ARRAY ” on page 546	Returns a value with the same array type as *array-variable* but with the cardinality reduced by the value of *numeric-expression*.
“UNNEST ” on page 624	Returns a result table that includes a row for each element of the specified array.

Table 40. Cast scalar functions	
Function	**Description**
“BIGINT ” on page 290	Returns a 64-bit integer representation of a value in the form of an integer constant.
“BINARY ” on page 292	Returns a fixed-length binary string representation of a string of any data type.
“BLOB ” on page 295	Returns a BLOB representation of a string of any type.
“BPCHAR ” on page 296	Returns a VARCHAR representation of a value.
“CHAR ” on page 300	Returns a CHARACTER representation of a value.
“CLOB ” on page 308	Returns a CLOB representation of a value.
“DATE ” on page 315	Returns a DATE from a value.
“DATETIME ” on page 316	Returns a TIMESTAMP from a value or a pair of values.
“DBCLOB ” on page 325	Returns a DBCLOB representation of a string.
“DECFLOAT ” on page 327	Returns the decimal floating-point representation of a value.
“DECIMAL or DEC ” on page 331	Returns a DECIMAL representation of a value.
“DOUBLE_PRECISION or DOUBLE” on page 339	Returns the floating-point representation of a value.
EMPTY_BLOB, EMPTY_CLOB, and EMPTY_DBCLOB scalar functions	Return a zero-length value of the associated data type.
“FLOAT ” on page 348	Returns a DOUBLE representation of a value.
“FLOAT4 ” on page 349	Returns a REAL representation of a value.

Table 40. Cast scalar functions (continued)	
Function	**Description**
“FLOAT8 ” on page 349	Returns a DOUBLE representation of a value.
“GRAPHIC ” on page 353	Returns a GRAPHIC representation of a string.
“INT ” on page 376	Returns an INTEGER representation of a value.
“INTEGER ” on page 379	Returns an INTEGER representation of a value.
“INTERVAL ” on page 376	Returns a DECIMAL duration that corresponds to a duration specified as a string.
“INT2 ” on page 381	Returns a SMALLINT representation of a value.
“INT4 ” on page 381	Returns a INTEGER representation of a value.
“INT8 ” on page 381	Returns a BIGINT representation of a value.
“NCHAR ” on page 429	Returns a fixed-length national character string representation of a value.
“NCLOB ” on page 431	Returns an NCLOB representation of a national character string.
“NUMERIC ” on page 438	Returns a DECIMAL representation of a value.
“NVARCHAR ” on page 431	Returns a varying-length national character string representation of a value.
“REAL ” on page 455	Returns the single-precision floating-point representation of a value.
“SMALLINT ” on page 502	Returns a SMALLINT representation of a value.
“TIME ” on page 525	Returns a TIME from a value.
“TIMESTAMP ” on page 525	Returns a TIMESTAMP from a value or a pair of values.
“TO_CLOB” on page 537	Returns a CLOB representation of a character string type.
“TO_NCLOB ” on page 539	Returns an NCLOB representation of a character string.
“VARBINARY ” on page 556	Returns a VARBINARY (varying-length binary string) representation of a string of any data type.
“VARCHAR ” on page 557	Returns a VARCHAR representation of a value.
“VARGRAPHIC ” on page 573	Returns a VARGRAPHIC representation of a value.

Table 41. Datetime scalar functions	
Function	**Description**
“ADD_DAYS ” on page 277	Returns a datetime value that represents the first argument plus a specified number of days.
“ADD_HOURS ” on page 278	Returns a timestamp value that represents the first argument plus a specified number of hours.
“ADD_MINUTES ” on page 279	Returns a timestamp value that represents the first argument plus a specified number of minutes.
“ADD_MONTHS ” on page 280	Returns a datetime value that represents *expression* plus a specified number of months.
“ADD_SECONDS ” on page 281	Returns a timestamp value that represents the first argument plus a specified number of seconds and fractional seconds.

Table 41. Datetime scalar functions (continued)	
Function	**Description**
“ADD_YEARS ” on page 282	Returns a datetime value that represents the first argument plus a specified number of years.
“AGE ” on page 283	Returns a numeric value that specifies the number of full years, full months, and full days between the current timestamp and the argument.
“DATE_PART ” on page 316	Returns portion of a datetime based on its argument.
“DATE_TRUNC ” on page 318	Returns a timestamp expression rounded to the specified unit.
“DAY ” on page 320	Returns the day part of a value.
“DAYNAME ” on page 321	Returns a character string containing the name of the day (for example, Friday) for the day portion of expression, based on locale-name or the value of the special register CURRENT LOCALE LC_TIME.
“DAYOFMONTH ” on page 322	Returns an integer between 1 and 31 that represents the day of the month.
“DAYOFWEEK ” on page 322	Returns the day of the week in the first argument as an integer value. The integer value is in the range 1-7, where 1 represents the first day of the week, as specified in the second argument.
“DAYOFWEEK_ISO ” on page 323	Returns the day of the week from a value, where 1 is Monday and 7 is Sunday.
“DAYOFYEAR ” on page 323	Returns the day of the year from a value.
“DAYS ” on page 323	Returns an integer representation of a date.
“DAYS_BETWEEN ” on page 324	Returns the number of full days between the specified arguments.
“DAYS_TO_END_OF_MONTH ” on page 325	Returns the number of days to the end of the month.
“EXTRACT ” on page 344	Returns a portion of a date or timestamp based on the arguments.
“FIRST_DAY ” on page 348	Returns a date or timestamp that represents the first day of the month of the argument.
“FROM_UTC_TIMESTAMP ” on page 350	Returns a TIMESTAMP that is converted from Coordinated Universal Time to the timezone that is specified by the timezone string.
“HOUR ” on page 364	Returns the hour part of a value.
“HOURS_BETWEEN ” on page 365	Returns the number of full hours between the specified arguments.
“JULIAN_DAY ” on page 396	Returns an integer value representing the number of days from January 1, 4712 B.C. to the date specified in the argument.
“LAST_DAY ” on page 396	Returns a datetime value that represents the last day of the month of the argument.
“MICROSECOND ” on page 419	Returns the microsecond part of a value.
“MIDNIGHT_SECONDS ” on page 420	Returns an integer value representing the number of seconds between midnight and a specified time value.
“MINUTE ” on page 421	Returns the minute part of a value.
“MINUTES_BETWEEN ” on page 422	Returns the number of full minutes between the specified arguments.
“MONTH ” on page 425	Returns the month part of a value.

Table 41. Datetime scalar functions (continued)	
Function	**Description**
“MONTHNAME ” on page 425	Returns a character string containing the name of the month (for example, January) for the month portion of expression, based on locale-name or the value of the special register CURRENT LOCALE LC_TIME.
“MONTHS_BETWEEN ” on page 426	Returns an estimate of the number of months between *expression1* and *expression2*.
“NEXT_DAY ” on page 433	Returns a datetime value that represents the first weekday, named by *string-expression*, that is later than the date in *expression*.
“NEXT_MONTH ” on page 434	Returns the first day of the next month after the specified date.
“NEXT_QUARTER ” on page 435	Returns the first day of the next quarter after the specified date.
“NEXT_WEEK ” on page 435	Returns the first day of the next week after the specified date.
“NEXT_YEAR ” on page 436	Returns the first day of the next year after the specified date.
“NOW ” on page 437	Returns a timestamp based on when the SQL statement is executed at the current server.
“QUARTER ” on page 450	Returns an integer that represents the quarter of the year in which a date resides.
“ROUND ” on page 485	Returns a datetime value, rounded to the unit specified by *format-string*.
“ROUND_TIMESTAMP ” on page 490	Returns a timestamp that is the *expression* rounded to the unit specified by the *format-string*.
“SECOND ” on page 499	Returns the seconds part of a value.
“SECONDS_BETWEEN ” on page 500	Returns the number of full seconds between the specified arguments.
“THIS_MONTH ” on page 523	Returns the first day of the month in the specified date.
“THIS_QUARTER ” on page 523	Returns the first day of the quarter in the specified date.
“THIS_WEEK ” on page 524	Returns the first day of the week in the specified date.
“THIS_YEAR ” on page 524	Returns the first day of the year in the specified date.
“TIMESTAMP_FORMAT ” on page 527	Returns a timestamp from a character string (*argument1*) that has been interpreted using a format template (*argument2*).
“TIMESTAMP_ISO ” on page 532	Returns a timestamp value based on a date, time, or timestamp argument. If the argument is a date, it inserts zero for all the time elements. If the argument is a time, it inserts the value of CURRENT DATE for the date elements, and zero for the fractional time element.
“TIMESTAMPDIFF ” on page 533	Returns an estimated number of intervals of type *argument1*, based on the difference between two timestamps. The second argument is the result of subtracting two timestamp types and converting the result to CHAR.
“TIMEZONE ” on page 535	Converts a date and time in one timezone into a timestamp in another timezone.
“TO_CHAR ” on page 536	Returns a CHARACTER representation of a timestamp.
“TO_DATE ” on page 537	Returns a timestamp from a character string.

Table 41. Datetime scalar functions (continued)	
Function	**Description**
“TO_NCHAR ” on page 539	Returns a national character representation of an input expression that has been formatted using a character template.
“TO_TIMESTAMP ” on page 540	Returns a timestamp that is based on the interpretation of the input string using the specified format.
“TO_UTC_TIMESTAMP ” on page 541	Returns a TIMESTAMP that is converted from Coordinated Universal Time to the timezone specified by the timezone string.
“TRUNCATE or TRUNC ” on page 548	Returns a datetime value, truncated to the unit specified by *format-string*.
“TRUNC_TIMESTAMP ” on page 547	Returns a timestamp that is the *expression* truncated to the unit specified by the *format-string*.
“VARCHAR_FORMAT ” on page 564	Returns a CHARACTER representation of a timestamp (*argument1*), formatted according to a template (*argument2*).
“WEEK ” on page 580	Returns the week of the year from a value, where the week starts with Sunday.
“WEEK_ISO ” on page 581	Returns the week of the year from a value, where the week starts with Monday.
“WEEKS_BETWEEN ” on page 581	Returns the number of full weeks between the specified arguments.
“YEAR ” on page 615	Returns the year part of a value.
“YEARS_BETWEEN ” on page 615	Returns the number of full years between the specified arguments.
“YMD_BETWEEN ” on page 616	Returns a numeric value that specifies the number of full years, full months, and full days between two datetime values.

Table 42. JSON scalar functions	
Function	**Description**
“BSON_TO_JSON ” on page 296	Converts a string that contains data that is formatted as BSON to a character string that contains data that is formatted as JSON.
“JSON_ARRAY ” on page 383	Generates a JSON array by explicitly listing the array elements by using an expression or by using a query.
“JSON_OBJECT ” on page 386	Generates a JSON object by using the specified key:value pairs. If no key:value pairs are provided, an empty object is returned.
“JSON_QUERY ” on page 389	Returns an SQL/JSON value from the specified JSON text by using an SQL/JSON path expression.
“JSON_TO_BSON ” on page 392	Converts a string that contains data that is formatted for JSON to a binary string that contains data that is formatted as BSON.
“JSON_VALUE ” on page 393	Returns an SQL scalar value from JSON text, by using an SQL/JSON path expression.

Table 43. Miscellaneous scalar functions	
Function	**Description**
“BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT ” on page 293	These bitwise functions operate on the "two's complement" representation of the integer value of the input arguments and return the result as a corresponding base 10 integer value in a data type based on the data type of the input arguments.
“COALESCE ” on page 308	Returns the first argument that is not null.
“CURSOR_ROWCOUNT ” on page 314	Returns the cumulative count of all rows fetched by the specified cursor since the cursor was opened.
“DECODE ” on page 334	Compares each specified *expression2* to *expression1*. If *expression1* is equal to *expression2*, or both *expression1* and *expression2* are null, the value of the following *result-expresssion* is returned. If no *expression2* matches *expression1*, the value of *else-expression* is returned; otherwise a null value is returned.
“DEREF ” on page 337	Returns an instance of the target type of the reference type argument.
“EVENT_MON_STATE ” on page 343	Returns the operational state of particular event monitor.
“GREATEST ” on page 358	Returns the maximum value in a set of values.
“HEX ” on page 362	Returns a hexadecimal representation of a value.
“IDENTITY_VAL_LOCAL ” on page 366	Returns the most recently assigned value for an identity column.
“ INTNAND, INTNOR, INTNXOR, and INTNNOT ” on page 381	These bitwise functions operate on the "two's complement" representation of the integer value of the input arguments and return the result as a corresponding base 10 integer value.
“LEAST ” on page 398	Returns the minimum value in a set of values.
“LENGTH ” on page 402	Returns the length of a value.
“MAX ” on page 418	Returns the maximum value in a set of values.
“MIN ” on page 421	Returns the minimum value in a set of values.
“NULLIF ” on page 437	Returns a null value if the arguments are equal; otherwise, it returns the value of the first argument.
“NVL ” on page 438	Returns the first argument that is not null.
“RAISE_ERROR ” on page 452	Raises an error in the SQLCA. The sqlstate that is to be returned is indicated by *argument1*. The second argument contains any text that is to be returned.
“RAWTOHEX ” on page 454	Returns a hexadecimal representation of a value as a character string.
“REC2XML ” on page 456	Returns a string formatted with XML tags, containing column names and column data.
“RID and RID_BIT ” on page 479	The RID_BIT scalar function returns the row identifier (RID) of a row in a character string format. The RID scalar function returns the RID of a row in large integer format. The RID function is not supported in partitioned database environments. The RID_BIT function is preferred over the RID function.
“TABLE_NAME ” on page 520	Returns an unqualified name of a table or view based on the object name specified in *argument1*, and the optional schema name specified in *argument2*. The returned value is used to resolve aliases.

Table 43. Miscellaneous scalar functions (continued)	
Function	**Description**
“TABLE_SCHEMA ” on page 521	Returns the schema name portion of a two-part table or view name (given by the object name in *argument1* and the optional schema name in *argument2*). The returned value is used to resolve aliases.
“TO_HEX ” on page 537	Converts a numeric expression into the hexadecimal representation.
“TYPE_ID ” on page 550	Returns the internal data type identifier of the dynamic data type of the argument. The result of this function is not portable across databases.
“TYPE_NAME ” on page 551	Returns the unqualified name of the dynamic data type of the argument.
“TYPE_SCHEMA ” on page 552	Returns the schema name of the dynamic data type of the argument.
“VALUE ” on page 556	Returns the first argument that is not null.

Table 44. Numeric scalar functions	
Function	**Description**
“ABS or ABSVAL ” on page 276	Returns the absolute value of a number.
“ACOS ” on page 276	Returns the arc cosine of a number, in radians.
“ASIN ” on page 289	Returns the arc sine of a number, in radians.
“ATAN ” on page 289	Returns the arc tangent of a number, in radians.
“ATANH ” on page 290	Returns the hyperbolic arc tangent of a number, in radians.
“ATAN2 ” on page 289	Returns the arc tangent of x and y coordinates as an angle expressed in radians.
“CEILING or CEIL ” on page 299	Returns the smallest integer value that is greater than or equal to a number.
“COMPARE_DECFLOAT ” on page 311	Returns a SMALLINT value that indicates whether the two arguments are equal or unordered, or whether one argument is greater than the other.
“COS ” on page 313	Returns the cosine of a number.
“COSH ” on page 313	Returns the hyperbolic cosine of a number.
“COT ” on page 314	Returns the cotangent of the argument, where the argument is an angle expressed in radians.
“DECFLOAT_FORMAT ” on page 329	Returns a DECFLOAT(34) from a character string.
“DEGREES ” on page 337	Returns the number of degrees of an angle.
“DIGITS ” on page 338	Returns a character-string representation of the absolute value of a number.
“EXP ” on page 343	Returns a value that is the base of the natural logarithm (e) raised to a power specified by the argument.
“FLOOR ” on page 349	Returns the largest integer value that is less than or equal to a number.
“LN ” on page 404	Returns the natural logarithm of a number.
“LOG10 ” on page 410	Returns the common logarithm (base 10) of a number.
“MOD ” on page 423	Returns the remainder of the first argument divided by the second argument.

Table 44. Numeric scalar functions (continued)	
Function	**Description**
“MOD (SYSFUN schema) ” on page 424	Returns the remainder of the first argument divided by the second argument.
“MULTIPLY_ALT ” on page 427	Returns the product of two arguments as a decimal value. This function is useful when the sum of the argument precisions is greater than 31.
“NORMALIZE_ DECFLOAT ” on page 436	Returns a decimal floating-point value that is the result of the argument set to its simplest form.
“POW ” on page 448	Returns the result of raising the first argument to the power of the second argument.
“POWER ” on page 448	Returns the result of raising the first argument to the power of the second argument.
“QUANTIZE ” on page 448	Returns a decimal floating-point number that is equal in value and sign to the first argument, and whose exponent is equal to the exponent of the second argument.
“RADIANS ” on page 451	Returns the number of radians for an argument that is expressed in degrees.
“RANDOM ” on page 454	Returns a floating point value between 0 and 1.
“RAND (SYSFUN schema) ” on page 453	Returns a random number.
“RAND (SYSIBM schema) ” on page 453	Returns a floating point value between 0 and 1.
“ROUND ” on page 485	Returns a numeric value that has been rounded to the specified number of decimal places.
“SIGN ” on page 501	Returns the sign of a number.
“SIN ” on page 501	Returns the sine of a number.
“SINH ” on page 502	Returns the hyperbolic sine of a number.
“SQRT ” on page 504	Returns the square root of a number.
“TAN ” on page 522	Returns the tangent of a number.
“TANH ” on page 523	Returns the hyperbolic tangent of a number.
“TO_NUMBER ” on page 539	Returns a DECFLOAT(34) from a character string.
“TOTALORDER ” on page 542	Returns a SMALLINT value of -1, 0, or 1 that indicates the comparison order of two arguments.
“TRUNCATE or TRUNC ” on page 548	Returns a number value that has been truncated at a specified number of decimal places.
“VARCHAR_FORMAT ” on page 564	Returns a CHARACTER representation of a timestamp (*argument1*), formatted according to a template (*argument2*).
“WIDTH_BUCKET ” on page 582	Creates equal-width histograms.

Table 45. Partitioning scalar functions	
Function	**Description**
“DATAPARTITIONNUM ” on page 314	Returns the sequence number (SYSDATAPARTITIONS.SEQNO) of the data partition in which the row resides. The argument is any column name within the table.
“DBPARTITIONNUM ” on page 326	Returns the database partition number of the row. The argument is any column name within the table.
“HASH ” on page 359	Returns the 128-bit, 160-bit, 256-bit or 512-bit hash of the input data.
“HASH4 ” on page 360	Returns the 32-bit hash of the input data.
“HASH8 ” on page 361	Returns the 64-bit hash of the input data.
“HASHEDVALUE ” on page 361	Returns the distribution map index (0 to 32767) of the row. The argument is a column name within a table.

Table 46. Regular expression scalar functions	
Function	**Description**
“REGEXP_COUNT ” on page 460	Returns a count of the number of times that a regular expression pattern is matched in a string.
“REGEXP_EXTRACT ” on page 462	Returns one occurrence of a substring of a string that matches the regular expression pattern.
“REGEXP_INSTR ” on page 462	Returns the starting or ending position of the matched substring, depending on the value of the return_option argument.
“REGEXP_LIKE ” on page 465	Returns a boolean value indicating if the regular expression pattern is found in a string. The function can be used only where a predicate is supported.
“REGEXP_MATCH_COUNT ” on page 467	Returns a count of the number of times that a regular expression pattern is matched in a string.
“REGEXP_REPLACE ” on page 468	Returns a modified version of the source string where occurrences of the regular expression pattern found in the source string are replaced with the specified replacement string.
“REGEXP_SUBSTR ” on page 470	Returns one occurrence of a substring of a string that matches the regular expression pattern.

Table 47. Security scalar functions	
Function	**Description**
“SECLABEL ” on page 496	Returns an unnamed security label.
“SECLABEL_BY_NAME ” on page 497	Returns a specific security label.
“SECLABEL_TO_CHAR ” on page 497	Accepts a security label and returns a string that contains all elements in the security label.
VERIFY_GROUP_FOR_USER	Returns a value that indicates whether any of the groups associated with **authorization-id-expression** are in the group names specified by the list of **group-name-expression** arguments.

Table 47. Security scalar functions (continued)	
Function	**Description**
“VERIFY_ROLE_FOR_USER ” on page 579	Returns a value that indicates whether any of the roles associated with **authorization-id-expression** are in, or contain any of, the role names specified by the list of **role-name-expression** arguments.
VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER	Returns a value that indicates whether **authorization-id-expression** has acquired a role under a trusted connection associated with some trusted context and that role is in, or contained in any of, the role names specified by the list of **role-name-expression** arguments.

Table 48. String scalar functions	
Function	**Description**
“ASCII ” on page 288	Returns the ASCII code value of the leftmost character of the argument as an integer.
“BTRIM ” on page 297	Removes characters from the beginning and end of a string expression.
“CHARACTER_LENGTH ” on page 306	Returns the length of an expression in the specified *string-unit*.
“CHR ” on page 307	Returns the character that has the ASCII code value specified by the argument.
“COLLATION_KEY” on page 309	Returns a VARBINARY string representing the collation key of the specified *string-expression* in the specified *collation-name*.
“COLLATION_KEY_BIT ” on page 310	Returns a VARCHAR FOR BIT DATA string representing the collation key of the specified *string-expression* in the specified *collation-name*.
“CONCAT ” on page 312	Returns a string that is the concatenation of two strings.
“DECRYPT_BIN and DECRYPT_CHAR ” on page 335	Returns a value that is the result of decrypting encrypted data using a password string.
“DIFFERENCE ” on page 338	Returns the difference between the sounds of the words in two argument strings, as determined by the SOUNDEX function. A value of 4 means the strings sound the same.
“ENCRYPT ” on page 341	Returns a value that is the result of encrypting a data string expression.
“GENERATE_UNIQUE ” on page 351	Returns a bit data character string that is unique compared to any other execution of the same function.
“GETHINT ” on page 352	Returns the password hint if one is found.
“INITCAP” on page 369	Returns a string with the first character of each word converted to uppercase and the rest to lowercase.
“INSERT ” on page 371	Returns a string, where *argument3* bytes have been deleted from *argument1* (beginning at *argument2*), and *argument4* has been inserted into *argument1* (beginning at *argument2*).
“INSTR ” on page 374	Returns the starting position of a string within another string.
“INSTRB ” on page 376	Returns the starting position, in bytes, of a string within another string.

Table 48. String scalar functions (continued)	
Function	**Description**
“LCASE ” on page 397	Returns a string in which all the SBCS characters have been converted to lowercase characters.
“LCASE (locale sensitive) ” on page 397	Returns a string in which all characters have been converted to lowercase characters using the rules from the Unicode standard associated with the specified locale.
“LCASE (SYSFUN schema) ” on page 397	Returns a string in which all the SBCS characters have been converted to lowercase characters.
“LOWER (locale sensitive) ” on page 412	Returns a string in which all characters have been converted to lowercase characters using the rules from the Unicode standard associated with the specified locale.
“LEFT ” on page 398	Returns the leftmost characters from a string.
“LOCATE ” on page 405	Returns the starting position of one string within another string.
“LOCATE_IN_STRING ” on page 408	Returns the starting position of the first occurrence of one string within another string.
“LOWER ” on page 411	Returns a string in which all the characters have been converted to lowercase characters.
“LPAD” on page 414	Returns a string that is padded on the left with the specified character, or with blanks.
“LTRIM ” on page 416	Removes blanks from the beginning of a string expression.
“LTRIM (SYSFUN schema) ” on page 418	Removes blanks from the beginning of a string expression.
“OCTET_LENGTH ” on page 439	Returns the length of an expression in octets (bytes).
“OVERLAY ” on page 440	Returns a string in which, beginning at *start* in the specified *source-string*, *length* of the specified code units have been deleted and *insert-string* has been inserted.
“POSITION ” on page 444	Returns the starting position of *argument2* within *argument1*.
“POSSTR ” on page 446	Returns the starting position of one string within another string.
“QUOTE_IDENT ” on page 450	Returns a string that can be used as an identifier in an SQL statement.
“QUOTE_LITERAL ” on page 451	Returns a string that can be used as a string constant in an SQL statement.
“REPEAT ” on page 473	Returns a character string composed of the first argument repeated the number of times specified by the second argument.
“REPEAT (SYSFUN schema) ” on page 474	Returns a character string composed of the first argument repeated the number of times specified by the second argument.
“REPLACE ” on page 475	Replaces all occurrences of *argument2* in *argument1* with *argument3*.
“REPLACE (SYSFUN schema) ” on page 478	Replaces all occurrences of *expres sion2* in *expression1* with *e xpression3*.
“RIGHT ” on page 481	Returns the rightmost characters from a string.
“RPAD” on page 491	Returns a string that is padded on the right with the specified character, string, or with blanks.

Table 48. String scalar functions (continued)	
Function	**Description**
“RTRIM ” on page 494	Removes blanks from the end of a string expression.
“RTRIM (SYSFUN schema) ” on page 496	Removes blanks from the end of a string expression.
“SOUNDEX ” on page 503	Returns a 4-character code representing the sound of the words in the argument. This result can be compared with the sound of other strings.
“SPACE ” on page 504	Returns a character string that consists of a specified number of blanks.
“STRIP ” on page 505	Removes blanks or another specified character from the end, the beginning, or both ends of a string expression.
“STRLEFT ” on page 506	Returns the leftmost string of *string-expression* of length *length*, expressed in the specified string unit.
“STRPOS ” on page 506	Returns the starting position of one string within another string.
“STRRIGHT ” on page 506	Returns the rightmost string of *string-expression* of length *length*, expressed in the specified string unit.
“SUBSTR ” on page 506	Returns a substring of a string.
“SUBSTRB ” on page 515	Returns a substring of a string.
“SUBSTRING ” on page 518	Returns a substring of a string.
“TO_SINGLE_BYTE ” on page 540	Returns a string in which multi-byte characters are converted to the equivalent single-byte character where an equivalent character exists.
“TRANSLATE ” on page 543	Returns a string in which one or more characters in a string are converted to other characters.
“TRIM ” on page 545	Removes blanks or another specified character from the end, the beginning, or both ends of a string expression.
“UCASE ” on page 552	The UCASE function is identical to the TRANSLATE function except that only the first argument (*char-string-exp*) is specified.
“UCASE (locale sensitive) ” on page 552	Returns a string in which all characters have been converted to uppercase characters using the rules from the Unicode standard associated with the specified locale.
“UPPER ” on page 554	Returns a string in which all the characters have been converted to uppercase characters.
“UPPER (locale sensitive) ” on page 554	Returns a string in which all characters have been converted to uppercase characters using the rules from the Unicode standard associated with the specified locale.

Table 49. Table functions	
Function	**Description**
“BASE_TABLE ” on page 618	Returns both the object name and schema name of the object found after any alias chains have been resolved.
“JSON_TABLE ” on page 619	Returns a result table from the evaluation of SQL/JSON path expressions. Each item in the result sequence of the row SQL/JSON path expression represents one or more rows in the result table.

Table 49. Table functions (continued)	
Function	**Description**
“UNNEST ” on page 624	Returns a result table that includes a row for each element of the specified array.
“XMLTABLE ” on page 626	Returns a table from the evaluation of XQuery expressions, possibly using specified input arguments as XQuery variables. Each sequence item in the result sequence of the row XQuery expression represents a row of the result table.

Table 50. XML functions	
Function	**Description**
“PARAMETER ” on page 443	Represents a position in an SQL statement where the value is provided dynamically by XQuery as part of the invocation of the db2-fn:sqlquery function.
“XMLAGG ” on page 271	Returns an XML sequence containing an item for each non-null value in a set of XML values.
“XMLATTRIBUTES ” on page 585	Constructs XML attributes from the arguments.
“XMLCOMMENT ” on page 586	Returns an XML value with a single XQuery comment node with the input argument as the content.
“XMLCONCAT ” on page 586	Returns a sequence containing the concatenation of a variable number of XML input arguments.
“XMLDOCUMENT ” on page 587	Returns an XML value with a single XQuery document node with zero or more children nodes.
“XMLELEMENT ” on page 588	Returns an XML value that is an XML element node.
“XMLFOREST ” on page 594	Returns an XML value that is a sequence of XML element nodes.
“XMLGROUP ” on page 273	Returns an XML value with a single XQuery document node containing one top-level element node.
“XMLNAMESPACES ” on page 596	Constructs namespace declarations from the arguments.
“XMLPARSE ” on page 597	Parses the argument as an XML document and returns an XML value.
“XMLPI ” on page 599	Returns an XML value with a single XQuery processing instruction node.
“XMLQUERY ” on page 600	Returns an XML value from the evaluation of an XQuery expression possibly using specified input arguments as XQuery variables.
“XMLROW ” on page 603	Returns an XML value with a single XQuery document node containing one top-level element node.
“XMLSERIALIZE ” on page 605	Returns a serialized XML value of the specified data type generated from the argument.
“XMLTABLE ” on page 626	Returns a table from the evaluation of XQuery expressions, possibly using specified input arguments as XQuery variables. Each sequence item in the result sequence of the row XQuery expression represents a row of the result table.
“XMLTEXT ” on page 606	Returns an XML value with a single XQuery text node having the input argument as the content.
“XMLVALIDATE ” on page 607	Returns a copy of the input XML value augmented with information obtained from XML schema validation, including default values.

Table 50. XML functions (continued)	
Function	**Description**
“XMLXSROBJECTID ” on page 611	Returns the XSR object identifier of the XML schema used to validate the XML document that is specified in the argument
“XSLTRANSFORM ” on page 612	Converts XML data into other formats, including the conversion of XML documents that conform to one XML schema into documents that conform to another schema.

Aggregate functions

An aggregate function (formerly known as a column function) accepts arguments and returns a single scalar value that is the result of an evaluation of a set of like values, such as those in a column within a set of one or more rows.

The argument of an aggregate function is a set of values derived from an expression. The expression can include columns, but cannot include a *scalar-fullselect*, another aggregate function, or an XMLQUERY or XMLEXISTS expression (SQLSTATE 42607). The scope of the set is a group or an intermediate result table.

If a GROUP BY clause is specified in a query, and the intermediate result of the FROM, WHERE, GROUP BY, and HAVING clauses is the empty set, the aggregate functions are not applied; the result of the query is the empty set; the SQLCODE is set to +100; and the SQLSTATE is set to '02000'.

If a GROUP BY clause is *not* specified in a query, and the intermediate result of the FROM, WHERE, and HAVING clauses is the empty set, the aggregate functions are applied to the empty set.

For example, the result of the following SELECT statement is the number of distinct values of JOBCODE for employees in department D01:

```
SELECT COUNT(DISTINCT JOBCODE)
  FROM CORPDATA.EMPLOYEE
  WHERE WORKDEPT = 'D01'
```

The keyword DISTINCT is not considered to be an argument of the function, but rather a specification of an operation that is performed before the function is applied. If DISTINCT is specified, duplicate values are eliminated. When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

If ALL is implicitly or explicitly specified, duplicate values are not eliminated.

For compatibility with other SQL implementations, UNIQUE can be specified as a synonym for DISTINCT in aggregate functions.

Expressions can be used in aggregate functions. For example:

```
SELECT MAX(BONUS + 1000)
  INTO :TOP_SALESREP_BONUS
  FROM EMPLOYEE
  WHERE COMM > 5000
```

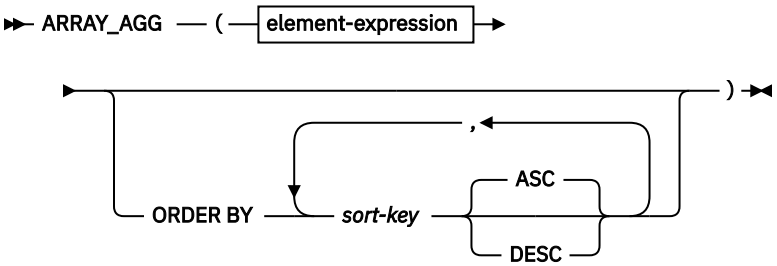
Aggregate functions can be qualified with a schema name (for example, SYSIBM.COUNT(*)).

ARRAY_AGG

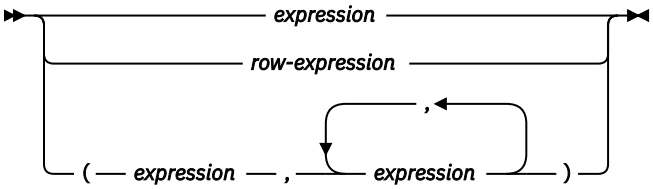
The ARRAY_AGG function aggregates a set of elements into an array.

Invocation of the ARRAY_AGG aggregate function is based on the result array type.

Ordinary array aggregation

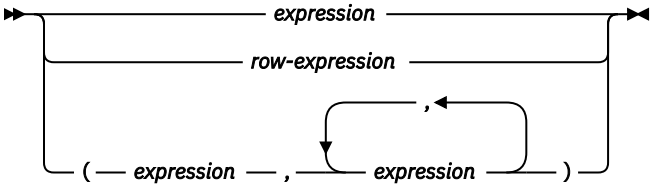


element-expression



Associative array aggregation

element-expression



The schema is SYSIBM.

Ordinary array aggregation

element-expression

Specifies the source for the elements of the array.

expression

An expression that specifies the element value for the array. The data type of the expression must be a data type that can be specified in a CREATE TYPE (array) statement (SQLSTATE 429C2).

row-expression

A row expression that specifies the value that has a row data type as the element of the array.

(*expression,expression...*)

A list of two or more expressions that specify the fields for a value that has a row data type as the element of the array. The data type of each expression must be a valid data type for a row field as described in CREATE TYPE (row) statement (SQLSTATE 429C5).

ORDER BY

Specifies the order of the rows from the same grouping set that are processed in the aggregation of an ordinary array. If the ORDER BY clause cannot distinguish the order of the column data, the rows in the same grouping set are arbitrarily ordered.

If ORDER BY is specified, it determines the order of the aggregated elements in the ordinary array. If ORDER BY is not specified and no other ARRAY_AGG, LISTAGG or XMLAGG is included in the same SELECT clause with ordering specified, the ordering of elements within the ordinary array is not deterministic. If ORDER BY is not specified and the same SELECT clause has multiple

occurrences of ARRAY_AGG, LISTAGG, or XMLAGG that specify ordering the same ordering of elements within the ordinary array is used for each result of ARRAY_AGG.

sort-key
The sort key can be a column name or a *sort-key-expression*. If the sort key is a constant, it does not refer to the position of the output column (as in the ORDER BY clause of a query), but it is simply a constant that implies no sort key.

ASC
Processes the *sort-key* in ascending order. This is the default option.

DESC
Processes the *sort-key* in descending order.

The result data type is an ordinary array. If the element values are specified using a single *expression* or *row-expression*, then the data type of the array element is the same as the type of the *expression* or *row-expression*. If the element values are specified with a list of expressions, then the array element is a row type with field types that correspond to the expressions.

If a SELECT clause includes an ARRAY_AGG function, then all invocations of ARRAY_AGG, LISTAGG, XMLAGG, and XMLGROUP functions in the same SELECT clause must specify the same order or not specify an order (SQLSTATE 428GZ).

Associative array aggregation

index-expression
Specifies the index of an associative array. When used in a context where there is a target user-defined array data type in the same statement or the result of the ARRAY_AGG is explicitly cast to a user-defined array data type, the data type of *index-expression* must be castable to the index data type of the target associative array data type. Otherwise, the data type of the *index-expression* must be a data type that can be specified for the index of an associative array in a CREATE TYPE (array) statement (SQLSTATE 429C2).

There cannot be any duplicate *index-expression* values in the grouping set that is processed to aggregate the associative array (SQLSTATE 22545).

element-expression
Specifies the source for the elements of the array.

expression
An expression that specifies the element value for the array. The data type of the expression must be a data type that can be specified in a CREATE TYPE (array) statement (SQLSTATE 429C2).

row-expression
A row expression that specifies the value that has a row data type as the element of the array.

(*expression,expression...*)
A list of two or more expressions that specify the fields for a value that has a row data type as the element of the array. The data type of each expression must be a valid data type for a row field as described in CREATE TYPE (row) statement (SQLSTATE 429C5).

The result data type is an associative array. If the ARRAY_AGG is used in a context where there is a target user-defined array data type in the same statement or the result of the ARRAY_AGG is explicitly cast to a user-defined array data type, the data type of the index matches the data type of the target associative array. If the element values are specified using a single *expression* or *row-expression*, then the data type of the array element is the same as the type of the expression or row-expression. If the element values are specified with a list of expressions, then the array element is a row type with field types that correspond to the expressions.

Notes

- The ARRAY_AGG function can only be specified within an SQL procedure, compiled SQL function, or compound SQL (compiled) statement the following specific contexts (SQLSTATE 42887):

  - The select-list of a SELECT INTO statement
  - The select-list of a fullselect in the definition of a cursor that is not scrollable
  - The select-list of a scalar subquery on the right side of a SET statement
- ARRAY_AGG cannot be used as part of an OLAP function (SQLSTATE 42887).
- The SELECT statement that uses ARRAY_AGG cannot contain an ORDER BY clause or a DISTINCT clause, and the SELECT clause or HAVING clause cannot contain a subquery or invoke an inlined SQL function that returns a subquery (SQLSTATE 42887).

Examples

- *Example 1:* Given the following DDL:

```
CREATE TYPE PHONELIST AS DECIMAL(10, 0)ARRAY[10]

CREATE TABLE EMPLOYEE (
  ID          INTEGER NOT NULL,
  PRIORITY    INTEGER NOT NULL,
  PHONENUMBER DECIMAL(10, 0),
  PRIMARY KEY(ID, PRIORITY))
```

  Create a procedure that uses a SELECT INTO statement to return the prioritized list of contact numbers under which an employee can be reached.

```
CREATE PROCEDURE GETPHONENUMBERS
  (IN  EMPID   INTEGER,
   OUT NUMBERS PHONELIST)
BEGIN
  SELECT ARRAY_AGG(PHONENUMBER ORDER BY PRIORITY)
    INTO NUMBERS
  FROM EMPLOYEE
  WHERE ID = EMPID;
END
```

  Create a procedure that uses a SET statement to return the list of contact numbers for an employee, in an arbitrary order.

```
CREATE PROCEDURE GETPHONENUMBERS
   (IN  EMPID   INTEGER,
    OUT NUMBERS PHONELIST)
 BEGIN
   SET NUMBERS =
     (SELECT ARRAY_AGG(PHONENUMBER)
     FROM EMPLOYEE
     WHERE ID = EMPID);
 END
```

- *Example 2:* Create a procedure that uses a SELECT INTO statement to aggregate priority 1 phone numbers into an associative array indexed by IDs from the EMPLOYEE table.

```
CREATE TYPE EMPPHONES AS DECIMAL(10,0) ARRAY[INTEGER]

CREATE PROCEDURE GETPHONES
(OUT EMPLOYEES EMPPHONES)
 BEGIN
  SELECT ARRAY_AGG(ID, PHONENUMBER)
  INTO EMPLOYEES
  FROM EMPLOYEE WHERE PRIORITY=1;
 END
```

- *Example 3:* Create a procedure that uses a SELECT INTO statement to aggregate the EMPLOYEE table into an array of row variable.

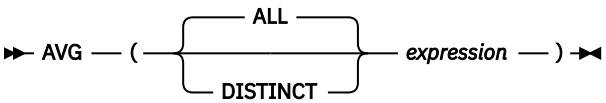
```
CREATE TYPE EMPROW AS ROW ANCHOR ROW EMPLOYEE

CREATE TYPE EMPARRAY AS EMPROW ARRAY[]

CREATE PROCEDURE GETEMPLOYEES
   (OUT EMPLOYEES EMPARRAY)
BEGIN
  SELECT ARRAY_AGG((ID, PRIORITY, PHONENUMBER) ORDER BY ID)
```

```
  INTO EMPLOYEES
  FROM EMPLOYEE;
END
```

AVG

The AVG function returns the average of a set of numbers.

Note: The result of the function can be affected by the enablement of the large_aggregation configuration parameter.



The schema is SYSIBM.

expression
An expression that returns a set of built-in numeric or Boolean values. The AVG function ignores any null values the input expression might contain.

ALL or DISTINCT
If ALL is specified, all values returned by the expression, including duplicate values, are used to calculate the average (this is the default). If DISTINCT is specified, duplicate values are ignored. Decimal floating-point values that are numerically equal are treated as duplicates even if they have different numbers of significant digits. For example, if the set of values returned by an expression includes the decimal floating-point numbers 123, 123.0, and 123.00, only one of these values is used to calculate the average.

Result

The data type of the result is the same as the data type of the input expression, with the following exceptions:

- If the data type of the input expression is SMALLINT, the data type of the result is INTEGER.
- If the data type of the input expression is BOOLEAN, the data type of the result is BIGINT. The result is 1 only if all values returned by the input expression are also 1; otherwise, the result is 0.
- If the data type of the input expression is single-precision floating point (REAL), the data type of the result is double-precision floating point (DOUBLE).
- If the data type of the input expression is DECFLOAT(*n*), the data type of the result is DECFLOAT(34).
- If the input expression is a DECIMAL value with precision *p* and scale *s*, the result is a DECIMAL with precision and scale as follows:

Table 51. Result precision and scale of the AVG aggregate function with DECIMAL input

DECIMAL arithmetic mode [1]	***p***	**Result precision**	**Result scale**
default	n/a	31	31-p+s
DEC15	<=15	15	15-p+s
DEC15	>15	31	MAX(0,28-p+s)
DEC31	n/a	31	MAX(0,28-p+s)

Note:

1. These modes are determined by the **dec_arithmetic** configuration parameter.

If the data type of the result is SMALLINT, INTEGER, or BIGINT, the fractional part of the average is truncated. It is not rounded up.

During evaluation, the order in which the input values are added together is undefined, but every intermediate result must be within the range of the data type of the result.

The result can be null. If the function is applied to an empty set, the result is a null value; otherwise, the result is the average value of the set.

Examples

- *Example 1*: Using the PROJECT table, set the host variable AVERAGE (decimal(5,2)) to the average staffing level (PRSTAFF) of projects in department (DEPTNO) 'D11'.

```
SELECT AVG(PRSTAFF)
  INTO :AVERAGE
  FROM PROJECT
  WHERE DEPTNO = 'D11'
```

  Results in AVERAGE being set to 4.25 (that is 17/4) when using the sample table.

- *Example 2*: Using the PROJECT table, set the host variable ANY_CALC (decimal(5,2)) to the average of each unique staffing level value (PRSTAFF) of projects in department (DEPTNO) 'D11'.

```
SELECT AVG(DISTINCT PRSTAFF)
  INTO :ANY_CALC
  FROM PROJECT
  WHERE DEPTNO = 'D11'
```

  Results in ANY_CALC being set to 4.66 (that is 14/3) when using the sample table.

CORRELATION

The CORRELATION function returns the coefficient of correlation of a set of number pairs.

The schema is SYSIBM.

expression1
An expression that returns a value of any built-in numeric data type.

expression2
An expression that returns a value of any built-in numeric data type.

If either argument is decimal floating-point, the result is DECFLOAT(34); otherwise, the result is a double-precision floating-point number. The result can be null. When not null, the result is between -1 and 1.

The function is applied to the set of (*expression1*, *expression2*) pairs derived from the argument values by the elimination of all pairs for which either *expression1* or *expression2* is null.

If the function is applied to an empty set, or if either STDDEV(*expression1*) or STDDEV(*expression2*) is equal to zero, the result is a null value. Otherwise, the result is the correlation coefficient for the value pairs in the set. The result is equivalent to the following expression:

```
COVARIANCE(expression1,expression2)/
(STDDEV(expression1)*
STDDEV(expression2))
```

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

CORR can be specified in place of CORRELATION.

Example

Using the EMPLOYEE table, set the host variable CORRLN (double-precision floating point) to the correlation between salary and bonus for those employees in department (WORKDEPT) 'A00'.

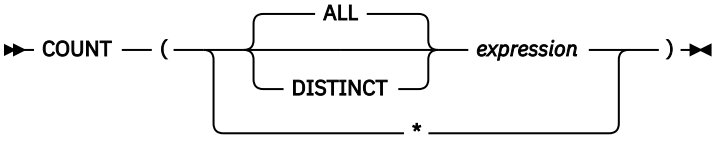
```
SELECT CORRELATION(SALARY, BONUS)
  INTO :CORRLN
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00'
```

CORRLN is set to approximately 9.99853953399538E-001 when using the sample table.

COUNT

The COUNT function returns the number of rows or values in a set of rows or values.

Note: The result of the function can be affected by the enablement of the large_aggregation configuration parameter.



The schema is SYSIBM.

expression

If ALL is implied or specified, an expression that returns a value of any built-in data type. If DISTINCT is specified, an expression that returns a value of any built-in data type except BLOB, CLOB, DBCLOB, or XML.

The result of the function is a large integer. The result cannot be null.

The argument of COUNT(*) is a set of rows. The result is the number of rows in the set. A row that includes only null values is included in the count.

The argument of COUNT(DISTINCT *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null and duplicate values. The result is the number of different non-null values in the set.

The argument of COUNT(*expression*) or COUNT(ALL *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values. The result is the number of non-null values in the set, including duplicates.

Examples

- *Example 1*: Using the EMPLOYEE table, set the host variable FEMALE (int) to the number of rows where the value of the SEX column is 'F'.

  ```
  SELECT COUNT(*)
    INTO :FEMALE
    FROM EMPLOYEE
    WHERE SEX = 'F'
  ```

  Results in FEMALE being set to 13 when using the sample table.

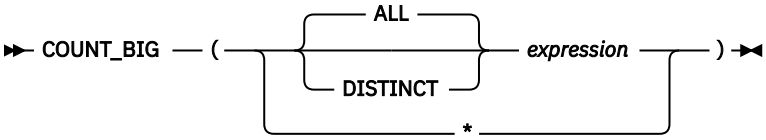
- *Example 2*: Using the EMPLOYEE table, set the host variable FEMALE_IN_DEPT (int) to the number of departments (WORKDEPT) that have at least one female as a member.

  ```
  SELECT COUNT(DISTINCT WORKDEPT)
    INTO :FEMALE_IN_DEPT
    FROM EMPLOYEE
    WHERE SEX = 'F'
  ```

  Results in FEMALE_IN_DEPT being set to 5 when using the sample table. (There is at least one female in departments A00, C01, D11, D21, and E11.)

COUNT_BIG

The COUNT_BIG function returns the number of rows or values in a set of rows or values. It is similar to COUNT except that the result can be greater than the maximum value of integer.



The schema is SYSIBM.

expression

If ALL is implied or specified, an expression that returns a value of any built-in data type. If DISTINCT is specified, an expression that returns a value of any built-in data type except BLOB, CLOB, DBCLOB, or XML.

The result of the function is a decimal with precision 31 and scale 0. The result cannot be null.

The argument of COUNT_BIG(*) is a set of rows. The result is the number of rows in the set. A row that includes only null values is included in the count.

The argument of COUNT_BIG(DISTINCT *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null and duplicate values. The result is the number of different non-null values in the set.

The argument of COUNT_BIG(*expression*) or COUNT_BIG(ALL *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values. The result is the number of non-null values in the set, including duplicates.

Examples

- *Example 1*: Refer to COUNT examples and substitute COUNT_BIG for occurrences of COUNT. The results are the same except for the data type of the result.
- *Example 2*: Some applications may require the use of COUNT but need to support values larger than the largest integer. This can be achieved by use of sourced user-defined functions and setting the SQL path. The following series of statements shows how to create a sourced function to support COUNT(*) based on COUNT_BIG and returning a decimal value with a precision of 15. The SQL path is set such that the sourced function based on COUNT_BIG is used in subsequent statements such as the query shown.

```
CREATE FUNCTION RICK.COUNT() RETURNS DECIMAL(15,0)
  SOURCE SYSIBM.COUNT_BIG();
SET CURRENT PATH RICK, SYSTEM PATH;
SELECT COUNT(*) FROM EMPLOYEE;
```

Note how the sourced function is defined with no parameters to support COUNT(*). This only works if you name the function COUNT and do not qualify the function with the schema name when it is used. To get the same effect as COUNT(*) with a name other than COUNT, invoke the function with no parameters. Thus, if RICK.COUNT had been defined as RICK.MYCOUNT instead, the query would have to be written as follows:

```
SELECT MYCOUNT() FROM EMPLOYEE;
```

If the count is taken on a specific column, the sourced function must specify the type of the column. The following statements created a sourced function that will take any CHAR column as a argument and use COUNT_BIG to perform the counting.

```
CREATE FUNCTION RICK.COUNT(CHAR()) RETURNS DOUBLE
  SOURCE SYSIBM.COUNT_BIG(CHAR());
SELECT COUNT(DISTINCT WORKDEPT) FROM EMPLOYEE;
```

COVARIANCE

The COVARIANCE function returns the (population) covariance of a set of number pairs.

The schema is SYSIBM.

expression1
An expression that returns a value of any built-in numeric data type.

expression2
An expression that returns a value of any built-in numeric data type.

If either argument is decimal floating-point, the result is DECFLOAT(34); otherwise, the result is a double-precision floating-point number. The result can be null.

The function is applied to the set of (*expression1*,*expression2*) pairs derived from the argument values by the elimination of all pairs for which either *expression1* or *expression2* is null.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the covariance of the value pairs in the set.

The calculation that is used to determine the biased covariance is logically equivalent to the following formula:

```
COVARIANCE = SUM(
   ( expression1 - AVG(expression1) ) *
   ( expression2 - AVG(expression2) ) ) / COUNT(expression1)
```

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

COVAR or COVAR_POP can be specified in place of COVARIANCE.

Example

Set the host variable COVARNCE to the covariance between salary and bonus for those employees in department 'A00' in the EMPLOYEE table. The data type of the host variable COVARNCE is double-precision floating point.

```
   SELECT COVARIANCE(SALARY, BONUS)
     INTO :COVARNCE
     FROM EMPLOYEE
     WHERE WORKDEPT = 'A00'
```

COVARNCE is set to approximately 1.68888888888889E+006 when using the sample table.

The following result set is shown for reference.

```
SELECT SALARY, BONUS FROM EMPLOYEE WHERE WORKDEPT = 'A00'
```

```
SALARY      BONUS
----------- -----------
  152750.00     1000.00
   66500.00      900.00
   49250.00      600.00
   46500.00     1000.00
   39250.00      600.00

5 record(s) selected.
```

COVARIANCE_SAMP

The COVARIANCE_SAMP function returns the sample covariance of a set of number pairs.

The schema is SYSIBM.

expression1
An expression that returns a value of any built-in numeric data type.

expression2
An expression that returns a value of any built-in numeric data type.

If either argument is decimal floating-point, the result is DECFLOAT(34); otherwise, the result is a double-precision floating-point number. The result can be null.

The function is applied to the set of (*expression1*, *expression2*) pairs that are derived from the argument values by the elimination of all pairs for which either *expression1* or *expression2* is null.

If the function is applied to an empty set or a set with only one row, the result is a null value. Otherwise, the result is the sample covariance of the value pairs in the set.

The calculation that is used to determine the sample covariance is logically equivalent to the following formula:

```
COVARIANCE_SAMP = SUM(
   ( expression1 - AVG(expression1) ) *
   ( expression2 - AVG(expression2) ) ) / ( COUNT(expression1) – 1 )
```

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

COVAR_SAMP can be specified in place of COVARIANCE_SAMP.

Example

Set the host variable COVARNCE to the sample covariance between the salary and bonus for those employees in department 'A00' of the EMPLOYEE table. The data type of the host variable COVARNCE is double-precision floating point.

```
   SELECT COVARIANCE_SAMP(SALARY, BONUS)
     INTO :COVARNCE
     FROM EMPLOYEE
     WHERE WORKDEPT = 'A00'
```

COVARNCE is set to approximately +5.42875000000000E+006 when the sample table is used.

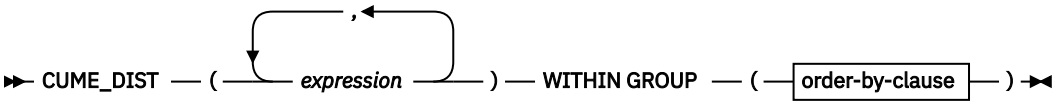
The following result set is shown for reference.

```
SELECT SALARY, BONUS FROM EMPLOYEE WHERE WORKDEPT = 'A00'
```

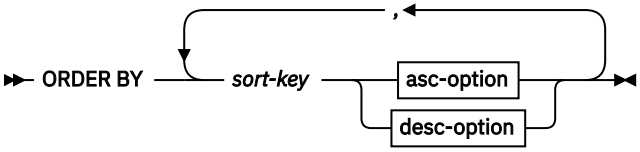
```
SALARY      BONUS
----------- -----------
  152750.00     1000.00
   66500.00      900.00
   49250.00      600.00
   46500.00     1000.00
   39250.00      600.00

5 record(s) selected.
```

CUME_DIST

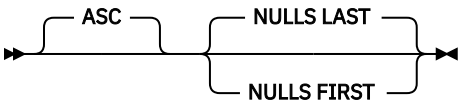
The CUME_DIST function returns the cumulative distribution of a row that is hypothetically inserted into a group of rows.



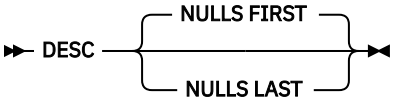
order-by-clause



asc-option



desc-option



The schema is SYSIBM.

expression
An expression that specifies a row that is hypothetically inserted into a group of rows. The expression must return a value that is a built-in data type. The expression must be a constant, a variable, or a cast of a constant or variable (SQLSTATE 428I9).

WITHIN GROUP
Indicates that the aggregation follows the specified ordering within the grouping set.

order-by-clause

ORDER BY
Specifies the order of the rows from the same grouping set that are processed in the aggregation.

sort-key
The sort key can be a column name or a sort-key-expression. If the sort key is a constant, it does not refer to the position of the output column (as in the ordinary ORDER BY clause), but it is a constant, which implies no sort key.

ASC
Uses the values of the *sort-key* in ascending order.

DESC
Uses the values of the *sort-key* in descending order.

NULLS FIRST
The ordering considers null values before all non-null values in the sort order.

NULLS LAST
The ordering considers null values after all non-null values in the sort order.

The number of expressions must be the same as the number of sort-key expressions (SQLSTATE 42822). The data type of each expression and the data type of the corresponding sort-key expression must be compatible (SQLSTATE 42822).

The data type of the result is DECFLOAT(34). The actual result is greater than 0.0 and less than or equal to 1.0.

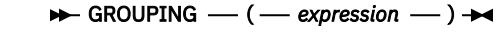
Example

Set the host variable CD to the cumulative distribution of a hypothetical new employee's salary of 47000 within the salaries of the employees in department 'A00'.

```
SELECT CUME_DIST(47000) WITHIN GROUP (ORDER BY SALARY)
   INTO :CD FROM EMPLOYEE WHERE WORKDEPT = 'A00'
```

GROUPING

Used in conjunction with grouping-sets and super-groups, the GROUPING function returns a value that indicates whether or not a row returned in a GROUP BY answer set is a row generated by a grouping set that excludes the column represented by *expression*.



The schema is SYSIBM.

expression

An expression that matches a *grouping-expression* from the GROUP BY clause of the same subselect.

The result of the function is a small integer. It is set to one of the following values:

1
The value of *expression* in the returned row is a null value, and the row was generated by the super-group. This generated row can be used to provide sub-total values for the GROUP BY expression.

0
The value is other than the previously listed value.

Example

The following query:

```
SELECT SALES_DATE, SALES_PERSON,
    SUM(SALES) AS UNITS_SOLD,
    GROUPING(SALES_DATE) AS DATE_GROUP,
    GROUPING(SALES_PERSON) AS SALES_GROUP
  FROM SALES
  GROUP BY CUBE (SALES_DATE, SALES_PERSON)
  ORDER BY SALES_DATE, SALES_PERSON
```

results in:

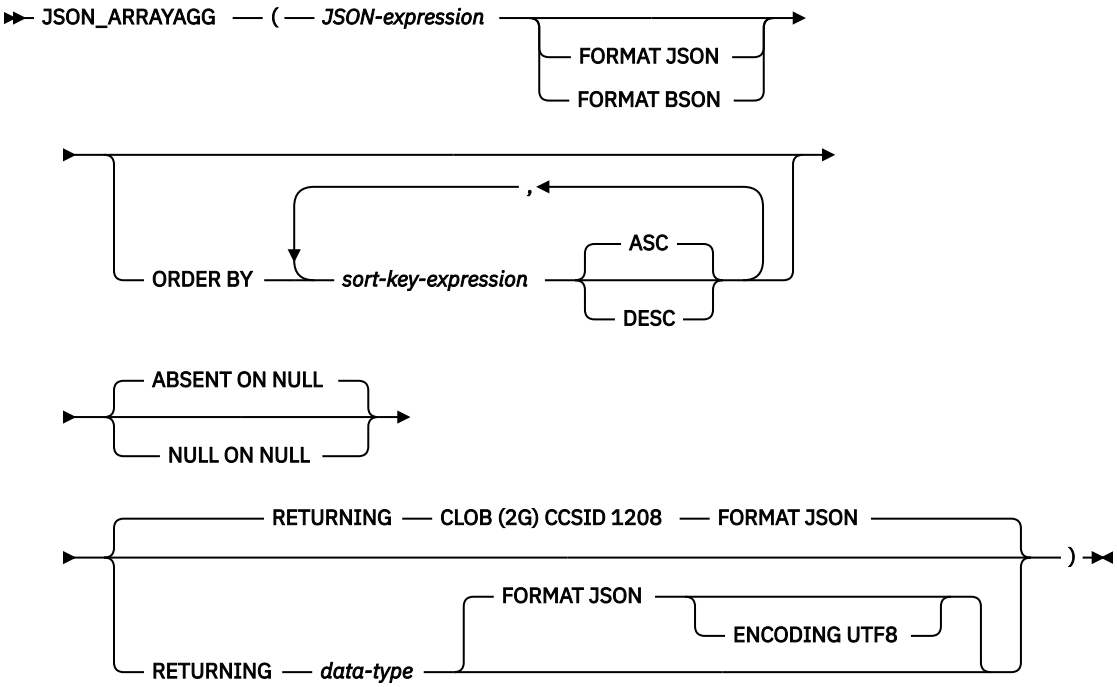
```
SALES_DATE SALES_PERSON    UNITS_SOLD  DATE_GROUP  SALES_GROUP
---------- --------------- ----------- ----------- -----------
12/31/1995 GOUNOT                    1           0           0
12/31/1995 LEE                       6           0           0
12/31/1995 LUCCHESSI                 1           0           0
12/31/1995 -                         8           0           1
03/29/1996 GOUNOT                   11           0           0
03/29/1996 LEE                      12           0           0
03/29/1996 LUCCHESSI                 4           0           0
03/29/1996 -                        27           0           1
03/30/1996 GOUNOT                   21           0           0
03/30/1996 LEE                      21           0           0
03/30/1996 LUCCHESSI                 4           0           0
03/30/1996 -                        46           0           1
03/31/1996 GOUNOT                    3           0           0
03/31/1996 LEE                      27           0           0
03/31/1996 LUCCHESSI                 1           0           0
03/31/1996 -                        31           0           1
04/01/1996 GOUNOT                   14           0           0
04/01/1996 LEE                      25           0           0
04/01/1996 LUCCHESSI                 4           0           0
04/01/1996 -                        43           0           1
-          GOUNOT                   50           1           0
-          LEE                      91           1           0
```

```
-           LUCCHESSI                14           1           0
-           -                       155           1           1
```

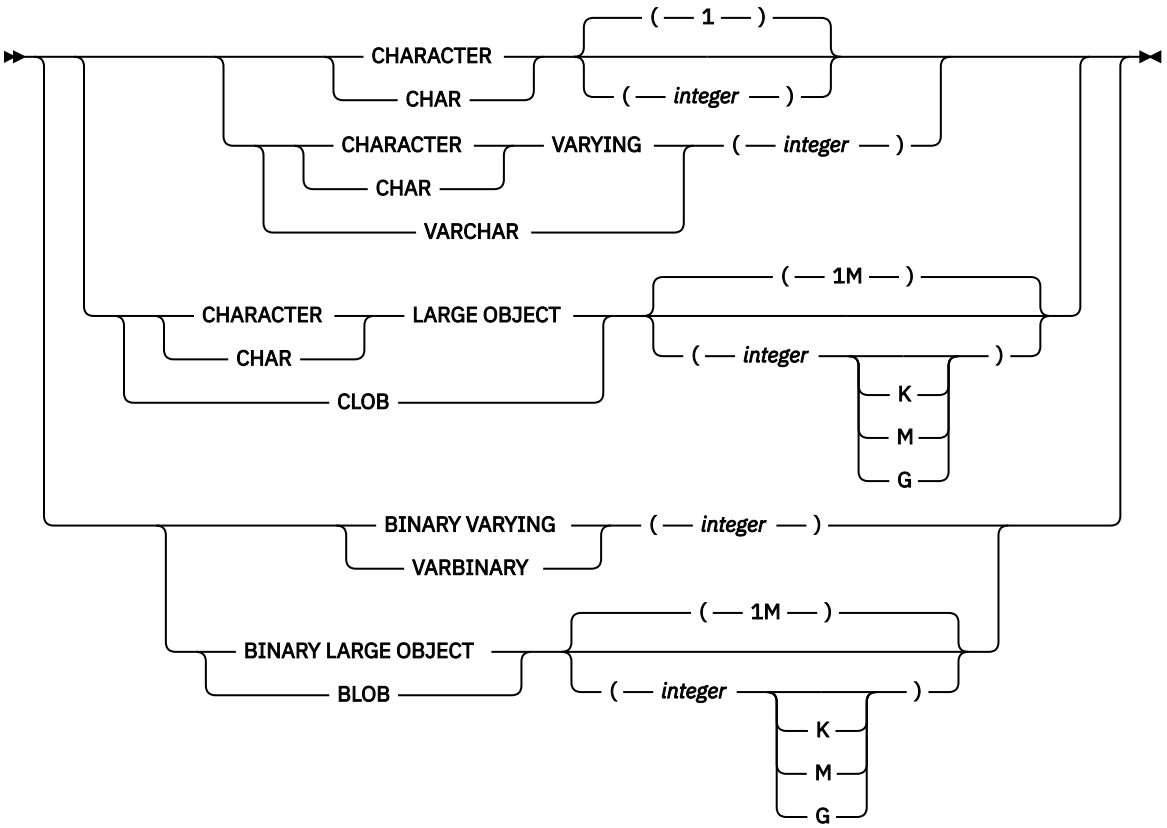
An application can recognize a SALES_DATE sub-total row by the fact that the value of DATE_GROUP is 0 and the value of SALES_GROUP is 1. A SALES_PERSON sub-total row can be recognized by the fact that the value of DATE_GROUP is 1 and the value of SALES_GROUP is 0. A grand total row can be recognized by the value 1 for both DATE_GROUP and SALES_GROUP.

JSON_ARRAYAGG

The JSON_ARRAYAGG function returns a JSON array that contains an array element for each value in a set of JSON or SQL values.



data-type



Note: The schema is SYSIBM. The function name cannot be specified as a qualified name.

JSON-expression

The expression to use to generate a value in the JSON array. The result type of this expression can be any built-in data type. It cannot be CHAR or VARCHAR bit data. It cannot be a user-defined type that is sourced on any of these data types.

FORMAT JSON or FORMAT BSON

Specifies whether the JSON expression is already formatted data.

FORMAT JSON

The JSON expression is formatted as JSON data. If the JSON expression is a character or graphic string data type, it is treated as JSON data. If the JSON expression is a binary string data type, it is interpreted as UTF-8 data.

FORMAT BSON

The JSON expression is formatted as the BSON representation of JSON data and must be a binary string data type.

If either the FORMAT JSON or FORMAT BSON properties are omitted from the statement, the format of the JSON expression is determined in the following ways:

- By the explicit or implicit FORMAT value of the JSON expression. This method applies when the JSON expression is one of the following built-in functions:
  - JSON_ARRAY
  - JSON_OBJECT
  - JSON_QUERY
  - JSON_ARRAYAGG
  - JSON_OBJECTAGG
- As FORMAT BSON when the JSON expression is a binary string type.

- As unformatted data in all other situations. If the generated value is not numeric, the result string is constructed with strings that are enclosed in quotation marks, and escape sequences for any special characters. A numeric value that is not a valid JSON number, such as INFINITY or NAN, results in an error.

ORDER BY
Specifies the order of the rows from the same grouping set that is processed in the aggregation. Rows in the same grouping are arbitrarily ordered when the ORDER BY clause is not specified, or if the ORDER BY clause cannot differentiate the order of the sort key value.

sort-key-expression
Specifies a sort key value that is either a column name or an expression. The data type of the column or expression must not be a DATALINK or XML value.

ASC
Processes the sort key expression in ascending order. This setting is the default.

DESC
Processes the sort key expression in descending order.

The ordering is based on the values of the sort keys, which might or might not be used in the JSON expression.

When you are running a statement that contains the JSON_ARRAYAGG function, weighted values are sometimes used to return the result. These values are used under the following circumstances:

- When a collating sequence other than *HEX is in effect.
- When the sort-key-expressions contain SBCS data, mixed data, or Unicode data.

The weighted values are derived by applying the collating sequence to the sort key expressions.

ABSENT ON NULL or NULL ON NULL
Specifies what to return when an array element that is produced by the JSON expression is the null value.

ABSENT ON NULL
A null array element is not included in the JSON array. This setting is the default.

NULL ON NULL
A null array element is included in the JSON array.

RETURNING *data-type*
Specifies the format of the result.

data-type
The data type of the result. The default data type is Character Large Object [CLOB (2G)]

FORMAT JSON
JSON data is returned as a JSON string.

ENCODING UTF8
The encoding to use when *data-type* is a binary string type. This clause is only allowed for binary string types. The result can be null. If the set of values is empty, the result is the null value.

The result can be null. If the set of values is empty, the result is the null value.

Notes

The JSON_ARRAYAGG aggregate function cannot be used as part of an OLAP specification.

Example

- The following example shows the command syntax for returning a JSON array containing all the department numbers.

```
SELECT JSON_ARRAYAGG(deptno) AS deptlist FROM dept;
```

The result is the following JSON array.

```
DEPT_NAME
-----------------------------
["sales","Procurment","finance","Eng","Design","Labour"]

1 record(s) selected.
```

- The following example shows the command syntax for returning a JSON array for each department that contains a list of employees that are assigned to that department.

```
SELECT deptno, JSON_ARRAYAGG(id) AS employe_id from employe group by deptno
```

The result is the following two rows.

```
DEPTN  EMPLOYE_ID
---------- ---------------------------------------
       102 [102001,102002,102003,102004]
       103 [103001,103002,103003,103004]
      2 record(s) selected.
```

- The following example shows the command syntax for returning a JSON array for each department number and its corresponding department name.

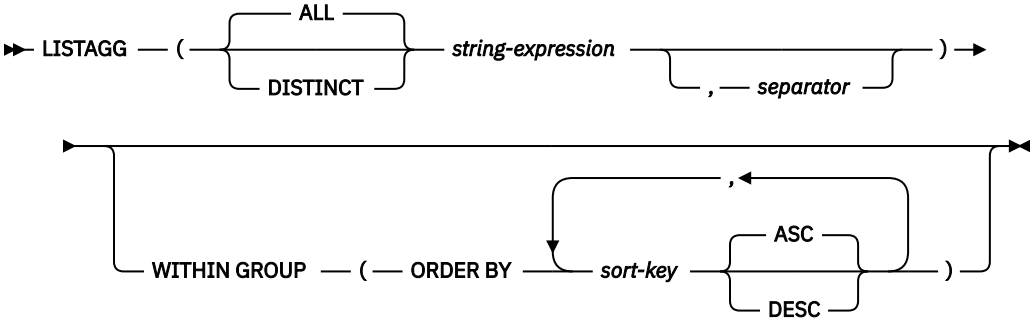
```
SELECT num as dept_num, JSON_ARRAYAGG (deptname) as dept_name FROM dept group by num
```

The result is the following JSON array.

```
DEPT_NUM    DEPT_NAME
----------- ---------------------------------------
        101 ["sales"]
        102 ["Procurment"]
        103 ["finance"]
        104 ["Eng"]
        105 ["Design"]
        106 ["Labour"]
6 record(s) selected.
```

LISTAGG

The LISTAGG function aggregates a set of string elements into one string by concatenating the strings. Optionally, a separator string can be provided which is inserted between contiguous input strings.



The schema is SYSIBM.

The LISTAGG function aggregates a set of string values for the group into one string by appending the *string-expression* values based on the order specified in the WITHIN GROUP clause.

The function is applied to the set of values that are derived from the first argument by the elimination of null values. If DISTINCT is specified, duplicate *string-expression* values are eliminated. If a *separator* argument is specified that is not the null value, the value is inserted between each pair of non-null *string-expression* values.

string-expression
An expression that specifies the string values to be aggregated. The expression must return a built-in character string, graphic string, binary string, numeric value, Boolean value, or datetime value:

- If the value is not a character, graphic, or, binary string, or if it is a CLOB, it is implicitly cast to VARCHAR before the function is evaluated.
- If the value is a DBCLOB, it is implicitly cast to VARGRAPHIC before the function is evaluated.
- The value cannot be a BLOB (SQLSTATE 42815).

separator
A constant expression that defines the separation string that is used between non-null *string-expression* values. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before the function is evaluated. CLOB and DBCLOB are supported through implicit casting. If the value is a CLOB, it is implicitly cast to VARCHAR before the function is evaluated. If the value is a DBCLOB, it is implicitly cast to VARGRAPHIC before the function is evaluated. The data type of *separator* cannot be a BLOB (SQLSTATE 42815).

The separator can be a constant, special register, variable, or an expression based on constants, special registers, or variables, provided that the expression does not include a non-deterministic function or a function that takes external action.

WITHIN GROUP
Indicates that the aggregation will follow the specified ordering within the grouping set.

If WITHIN GROUP is not specified and no other LISTAGG, ARRAY_AGG, or XMLAGG is included in the same SELECT clause with ordering specified, the ordering of strings within the result is not deterministic. If WITHIN GROUP is not specified, and the same SELECT clause has multiple occurrences of XMLAGG, ARRAY_AGG, or LISTAGG that specify ordering, the same ordering is used for the result of the LISTAGG function invocation.

ORDER BY
Specifies the order of the rows from the same grouping set that are processed in the aggregation. If the ORDER BY clause cannot distinguish the order of the column data, the rows in the same grouping set are arbitrarily ordered.

sort-key
The sort key can be a column name or a *sort-key-expression*. If the sort key is a constant, it does not refer to the position of the output column (as in the ORDER BY clause of a query); it is a constant, which implies no sort key.

ASC
Processes the *sort-key* in ascending order. This is the default option.

DESC
Processes the *sort-key* in descending order.

Result

The result data type of LISTAGG is based on the data type of *string-expression*:

Table 52. Determining the result data type and length

Data type of *string-expression*	**Result data type and length**
CHAR(*n*) or VARCHAR(*n*)	VARCHAR(MAX(4000, *n*))
GRAPHIC(*n*) or VARGRAPHIC(*n*)	VARGRAPHIC(MAX(2000, *n*))
BINARY(*n*) or VARBINARY(*n*)	VARBINARY(MAX(4000, *n*))

The string unit of the result data type is the same as the string units of the data type of *string-expression*.

The result data type can exceed VARCHAR(4000), VARBINARY(4000), or VARGRAPHIC(2000) if a derived size is used to determine the size of the result, to a maximum for the result data type. The following example successfully yields a return data type of VARCHAR(10000):

```
LISTAGG(CAST(NAME AS VARCHAR(10000)), ',')
```

If the actual length of the aggregated result string exceeds the maximum for the result data type, an error is returned (SQLSTATE 22001).

The result can be null. If the function is applied to an empty set or all of the *string-expression* values in the set are null values, the result is a null value.

Rules

- If DISTINCT is specified for LISTAGG, the *sort-key* of the ORDER BY specification must match *string-expression* (SQLSTATE 42822). If *string-expression* is implicitly cast, the *sort-key* must explicitly include a corresponding matching cast specification.
- If a SELECT clause includes an ARRAY_AGG function, then all invocations of ARRAY_AGG, LISTAGG, XMLAGG, and XMLGROUP functions in the same SELECT clause must meet one of the following criteria (SQLSTATE 428GZ):
  - Specify the same order
  - Not specify an order
  - Have the *string-expression* argument of a LISTAGG with DISTINCT match the *sort-key* expression of the ORDER BY clause in ARRAY_AGG
- LISTAGG cannot be used as part of an OLAP specification (SQLSTATE 42887).

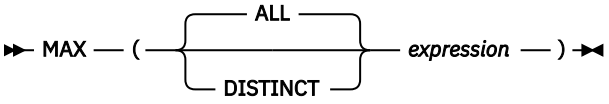
Example

Produce an alphabetical list of comma-separated names, grouped by department.

```
SELECT workdept,
      LISTAGG(lastname, ', ') WITHIN GROUP(ORDER BY lastname)
      AS employees
   FROM emp
   GROUP BY workdept
```

MAX

The MAX function returns the maximum value in a set of values.



The schema is SYSIBM.

expression
An expression that returns a value of any built-in data type other than BLOB, CLOB, DBCLOB, ROWID, or XML.

The data type, length and code page of the result are the same as the data type, length and code page of the argument values. The result is considered to be a derived value and can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the maximum value in the set.

The specification of DISTINCT has no effect on the result and therefore is not recommended. It is included for compatibility with other relational systems.

Notes

- ***Results involving DECFLOAT special values:*** If the data type of the argument is decimal floating-point and positive or negative infinity, sNaN, or NaN is found, the maximum value is determined using decimal floating-point ordering rules. If multiple representations of the same decimal floating-point value are found (for example, 2.00 and 2.0), it is unpredictable which representation will be returned.

Examples

- *Example 1*: Using the EMPLOYEE table, set the host variable MAX_SALARY (decimal(7,2)) to the maximum monthly salary (SALARY/12) value.

```
SELECT MAX(SALARY) / 12
  INTO :MAX_SALARY
  FROM EMPLOYEE
```

  Results in MAX_SALARY being set to 4395.83 when using the sample table.

- *Example 2*: Using the PROJECT table, set the host variable LAST_PROJ(char(24)) to the project name (PROJNAME) that comes last in the collating sequence.

```
SELECT MAX(PROJNAME)
  INTO :LAST_PROJ
  FROM PROJECT
```

  Results in LAST_PROJ being set to "WELD LINE PLANNING" when using the sample table.

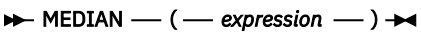
- *Example 3*: Similar to the previous example, set the host variable LAST_PROJ (char(40)) to the project name that comes last in the collating sequence when a project name is concatenated with the host variable PROJSUPP. PROJSUPP is "_Support"; it has a char(8) data type.

```
SELECT MAX(PROJNAME CONCAT PROJSUPP)
  INTO :LAST_PROJ
  FROM PROJECT
```

  Results in LAST_PROJ being set to "WELD LINE PLANNING_SUPPORT" when using the sample table.

MEDIAN

The MEDIAN function returns the median value in a set of values.



The schema is SYSIBM.

expression
An expression that specifies the set of values from which the median is determined. The expression must return a value that is a built-in numeric data type, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported through implicit casting. If the expression is not a numeric data type, it is cast to DECFLOAT(34) before the function is evaluated.

If the data type of *expression* is DECFLOAT(*n*), the data type of the result is DECFLOAT(34). Otherwise, the data type of the result is DOUBLE.

The function is applied to the set of values that are derived from the argument values by the elimination of null values.

The result can be null. If *expression* is null or if the function is applied to an empty set, the result is a null value.

The MEDIAN function is a synonym for the following expression:

```
PERCENTILE_CONT( 0.5 ) WITHIN GROUP( ORDER BY expression )
```

Example

Set the host variable MED to the value that corresponds to the median of the salaries of the employees in department 'E21'.

```
   SELECT MEDIAN(SALARY) INTO :MED FROM EMPLOYEE WHERE WORKDEPT = 'E21'
```

MED is set to a value of 41895.00.

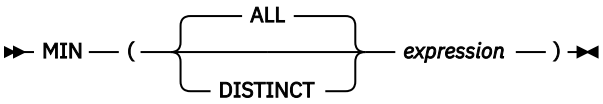
The following result set is shown for reference.

```
   SELECT SALARY FROM EMPLOYEE WHERE WORKDEPT = 'E21' ORDER BY SALARY
```

```
SALARY
-----------
   31840.00
   35370.00
   39950.00
   43840.00
   45370.00
   86150.00

  6 record(s) selected.
```

MIN

The MIN function returns the minimum value in a set of values.



expression
An expression that returns a value of any built-in data type other than a BLOB, CLOB, DBCLOB, or XML.

The data type, length, and code page of the result are the same as the data type, length, and code page of the argument values. The result is considered to be a derived value and can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If this function is applied to an empty set, the result of the function is a null value. Otherwise, the result is the minimum value in the set.

The specification of DISTINCT has no effect on the result and therefore is not recommended. It is included for compatibility with other relational systems.

Notes

- ***Results involving DECFLOAT special values:*** If the data type of the argument is decimal floating-point and positive or negative infinity, sNaN, or NaN is found, the minimum value is determined using decimal floating-point ordering rules. If multiple representations of the same decimal floating-point value are found (for example, 2.00 and 2.0), it is unpredictable which representation will be returned.

Examples

- *Example 1*: Using the EMPLOYEE table, set the host variable COMM_SPREAD (decimal(7,2)) to the difference between the maximum and minimum commission (COMM) for the members of department (WORKDEPT) 'D11'.

```
SELECT MAX(COMM) - MIN(COMM)
  INTO :COMM_SPREAD
  FROM EMPLOYEE
  WHERE WORKDEPT = 'D11'
```

  Results in COMM_SPREAD being set to 1118 (that is, 2580 - 1462) when using the sample table.

- *Example 2*: Using the PROJECT table, set the host variable (FIRST_FINISHED (char(10)) to the estimated ending date (PRENDATE) of the first project scheduled to be completed.

```
SELECT MIN(PRENDATE)
  INTO :FIRST_FINISHED
  FROM PROJECT
```

  Results in FIRST_FINISHED being set to '1982-09-15' when using the sample table.

PERCENTILE_CONT

The PERCENTILE_CONT function returns the value that corresponds to the specified percentile given a sort specification by using a continuous distribution model.

The schema is SYSIBM.

percentile
An expression that specifies the percentile. The expression must return a value that is a built-in numeric data type, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported through implicit casting. If the expression is not a numeric data type, it is cast to DECFLOAT(34) before the function is evaluated. The expression must be a constant, a variable, or a cast of a constant or variable (SQLSTATE 428I9). The value must be 0 - 1 (SQLSTATE 22003).

WITHIN GROUP
Indicates that the aggregation follows the specified ordering within the grouping set.

ORDER BY
Specifies the order of the rows from the same grouping set that are processed in the aggregation.

sort-key
The sort key must be a column name or an expression that contains a column reference. *sort-key* must be a built-in numeric data type (SQLSTATE 42822). If the single column result set specified by *sort-key* contains one or more NULL values, they are not considered and a warning is returned (SQLSTATE 01003).

ASC
Processes the *sort-key* in ascending order.

DESC
Processes the *sort-key* in descending order.

If the data type of *sort-key* is DECFLOAT(*n*), the data type of the result is DECFLOAT(34). Otherwise, the data type of the result is DOUBLE.

The result can be null. If *percentile* is null or the single column result set specified by *sort-key* is empty, the result is NULL.

When used in an OLAP specification, only the *window-partition-clause* can be specified.

Example

Set the host variable PC to the value that corresponds to the 75th percentile of the salaries of the employees in department 'E21' using a continuous distribution model.

```
SELECT PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY SALARY)
   INTO :PC FROM EMPLOYEE
   WHERE WORKDEPT = 'E21'
```

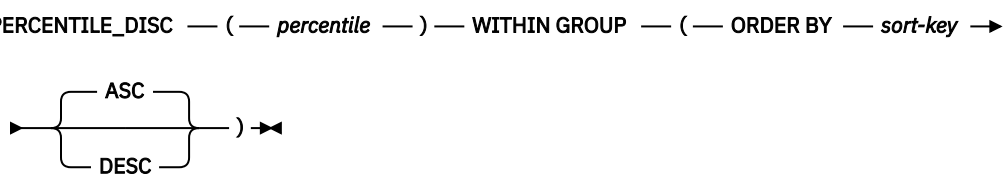
PC is set to a value of 44987.50.

The following result set is shown for reference:

```
SELECT SALARY FROM EMPLOYEE WHERE WORKDEPT = 'E21'
   ORDER BY SALARY

SALARY
-----------
   31840.00
   35370.00
   39950.00
   43840.00
   45370.00
   86150.00

  6 record(s) selected.
```

PERCENTILE_DISC

The PERCENTILE_DISC function returns the value that corresponds to the specified percentile given a sort specification by using a discrete distribution model.



The schema is SYSIBM.

percentile
An expression that specifies the percentile. The expression must return a value that is a built-in numeric data type, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If the expression is not a numeric data type, it is cast to DECFLOAT(34) before the function is evaluated. The expression must be a constant, a variable, or a cast of a constant or variable (SQLSTATE 428I9). The value must be 0 - 1 (SQLSTATE 22003).

WITHIN GROUP
Indicates that the aggregation follows the specified ordering within the grouping set.

ORDER BY
Specifies the order of the rows from the same grouping set that are processed in the aggregation.

sort-key
The sort key must be a column name or an expression that contains a column reference. *sort-key* must be a built-in numeric data type (SQLSTATE 42822). If the single column result set specified by *sort-key* contains one or more NULL values, they are not considered and a warning is returned (SQLSTATE 01003).

ASC
Processes the *sort-key* in ascending order.

DESC
Processes the *sort-key* in descending order.

The data type of the result is the same as the data type of *sort-key*.

The result can be null. If *percentile* is null or the single column result set specified by *sort-key* is empty, the result is NULL.

When used in an OLAP specification, only the *window-partition-clause* can be specified.

Example

Set the host variable PD to the value that corresponds to the 75th percentile of the salaries of the employees in department 'E21' using a discrete distribution model.

```
   SELECT PERCENTILE_DISC(0.75) WITHIN GROUP (ORDER BY SALARY)
      INTO :PD FROM EMPLOYEE WHERE WORKDEPT = 'E21'
```

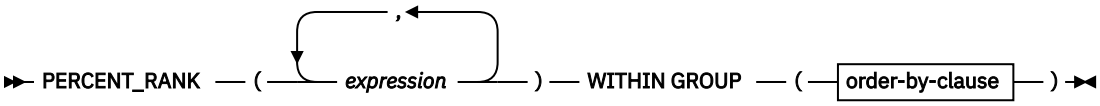
PD is set to a value of 45370.00.

The following result set is shown for reference.

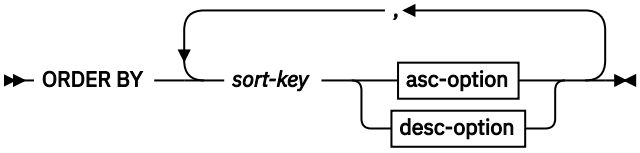
```
   SELECT SALARY FROM EMPLOYEE WHERE WORKDEPT = 'E21' ORDER BY SALARY

SALARY
-----------
   31840.00
   35370.00
   39950.00
   43840.00
   45370.00
   86150.00

  6 record(s) selected.
```

PERCENT_RANK

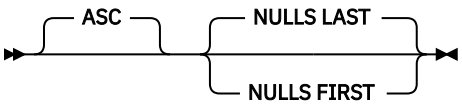
The PERCENT_RANK function returns the relative percentile rank of a row that is hypothetically inserted into a group of rows.



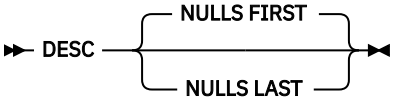
order-by-clause



asc-option



desc-option



The schema is SYSIBM.

expression
An expression that specifies a row that is hypothetically inserted into a group of rows. The expression must return a value that is a built-in data type. The expression must be a constant, a variable, or a cast of a constant or variable (SQLSTATE 428I9).

order-by-clause

ORDER BY
Specifies the order of the rows from the same grouping set that are processed in the aggregation.

sort-key
The sort key can be a column name or a *sort-key-expression*. If the sort key is a constant, it does not refer to the position of the output column. A constant implies no sort key, unlike a constant in the ordinary ORDER BY clause.

ASC
Uses the values of the *sort-key* in ascending order.

DESC
Uses the values of the *sort-key* in descending order.

NULLS FIRST
The ordering considers null values before all non-null values in the sort order.

NULLS LAST
The ordering considers null values after all non-null values in the sort order.

The number of expressions must be the same as the number of *sort-key* expressions (SQLSTATE 42822). The data type of each expression and the data type of the corresponding *sort-key* expression must be compatible (SQLSTATE 42822).

The data type of the result is DECFLOAT(34). The actual result is greater than 0.0 and less than or equal to 1.0.

Example

Set the host variable PR to the relative percentile rank of a hypothetical new employee's salary of 47000 within the salaries of the employees in department 'A00'.

```
   SELECT PERCENT_RANK(47000) WITHIN GROUP (ORDER BY SALARY)
      INTO :PR FROM EMPLOYEE WHERE WORKDEPT = 'A00'
```

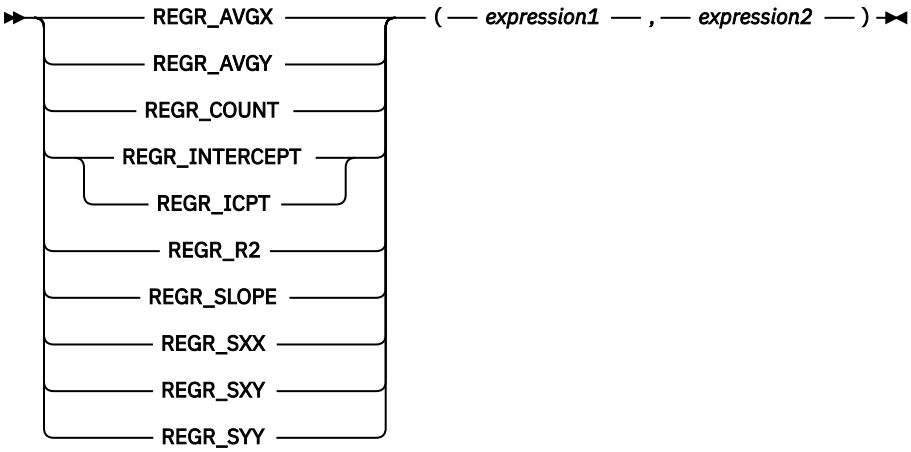
PR is set to a value of 0.4.

The following result set is shown for reference.

```
   SELECT SALARY FROM EMPLOYEE WHERE WORKDEPT = 'A00' ORDER BY SALARY

SALARY
-----------
   39250.00
   46500.00
   49250.00
   66500.00
  152750.00

  5 record(s) selected.
```

Regression functions (REGR_AVGX, REGR_AVGY, REGR_COUNT, ...)

The regression functions support the fitting of an ordinary-least-squares regression line of the form y = a * x + b to a set of number pairs.



The schema is SYSIBM.

expression1
An expression that returns a value of any built-in numeric data type. It is interpreted as a value of the dependent variable (that is, a "y value").

expression2
An expression that returns a value of any built-in numeric data type. It is interpreted as a value of the independent variable (that is, an "x value").

The REGR_COUNT function returns the number of non-null number pairs used to fit the regression line.

The REGR_INTERCEPT (or REGR_ICPT) function returns the y-intercept of the regression line ("b" in the equation mentioned previously).

The REGR_R2 function returns the coefficient of determination ("R-squared" or "goodness-of-fit") for the regression.

The REGR_SLOPE function returns the slope of the line ("a" in the equation mentioned previously).

The REGR_AVGX, REGR_AVGY, REGR_SXX, REGR_SXY, and REGR_SYY functions return quantities that can be used to compute various diagnostic statistics needed for the evaluation of the quality and statistical validity of the regression model.

The data type of the result of REGR_COUNT is integer. For the remaining functions, if either argument is DECFLOAT(*n*), the data type of the result is DECFLOAT(34); otherwise, the data type of the result is double-precision floating-point. If either argument is a special decimal floating-point value, the rules for general arithmetic operations for decimal floating-point apply. See "General arithmetic operation rules for decimal floating-point" in “General arithmetic operation rules for decimal floating-point” on page 142.

The result can be null. When not null, the result of REGR_R2 is between 0 and 1, and the result of both REGR_SXX and REGR_SYY is non-negative.

Each function is applied to the set of (*expression1*, *expression2*) pairs derived from the argument values by the elimination of all pairs for which either *expression1* or *expression2* is null.

If the set is not empty and VARIANCE(*expression2*) is positive, REGR_COUNT returns the number of non-null pairs in the set, and the remaining functions return results that are defined as follows:

```
REGR_SLOPE(expression1,expression2) =
COVARIANCE(expression1,expression2)/VARIANCE(expression2)
```

```
REGR_INTERCEPT(expression1, expression2) =
AVG(expression1) - REGR_SLOPE(expression1, expression2) * AVG(expression2)
```

```
REGR_R2(expression1, expression2) =
POWER(CORRELATION(expression1, expression2), 2) if VARIANCE(expression1)>0
REGR_R2(expression1, expression2) = 1 if VARIANCE(expression1)=0
```

```
REGR_AVGX(expression1, expression2) = AVG(expression2)
```

```
REGR_AVGY(expression1, expression2) = AVG(expression1)
```

```
REGR_SXX(expression1, expression2) =
REGR_COUNT(expression1, expression2) * VARIANCE(expression2)
```

```
REGR_SYY(expression1, expression2) =
REGR_COUNT(expression1, expression2) * VARIANCE(expression1)
```

```
REGR_SXY(expression1, expression2) =
REGR_COUNT(expression1, expression2) * COVARIANCE(expression1, expression2)
```

If the set is not empty and VARIANCE(*expression2*) is equal to zero, then the regression line either has infinite slope or is undefined. In this case, the functions REGR_SLOPE, REGR_INTERCEPT, and REGR_R2 each return a null value, and the remaining functions return values as defined previously. If the set is empty, REGR_COUNT returns zero and the remaining functions return a null value.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

The regression functions are all computed simultaneously during a single pass through the data. In general, it is more efficient to use the regression functions to compute the statistics needed for a regression analysis than to perform the equivalent computations using ordinary column functions such as AVERAGE, VARIANCE, COVARIANCE, and so forth.

The usual diagnostic statistics that accompany a linear-regression analysis can be computed in terms of the functions listed previously. For example:

Adjusted R2
1 - ((1 - REGR_R2) * ((REGR_COUNT - 1) / (REGR_COUNT - 2)))

Standard error
SQRT((REGR_SYY-(POWER(REGR_SXY,2)/REGR_SXX))/(REGR_COUNT-2))

Total sum of squares
REGR_SYY

Regression sum of squares
POWER(REGR_SXY,2) / REGR_SXX

Residual sum of squares
(Total sum of squares)-(Regression sum of squares)

t statistic for slope
REGR_SLOPE * SQRT(REGR_SXX) / (Standard error)

t statistic for y-intercept
REGR_INTERCEPT/((Standard error) * SQRT((1/REGR_COUNT)+(POWER(REGR_AVGX,2)/REGR_SXX))

Example

Using the EMPLOYEE table, compute an ordinary-least-squares regression line that expresses the bonus of an employee in department (WORKDEPT) `'A00'` as a linear function of the employee's salary. Set the host variables SLOPE, ICPT, RSQR (double-precision floating point) to the slope, intercept, and coefficient of determination of the regression line, respectively. Also set the host variables AVGSAL and AVGBONUS to the average salary and average bonus, respectively, of the employees in department `'A00'`, and set the host variable CNT (integer) to the number of employees in department `'A00'` for whom both salary

and bonus data are available. Store the remaining regression statistics in host variables SXX, SYY, and SXY.

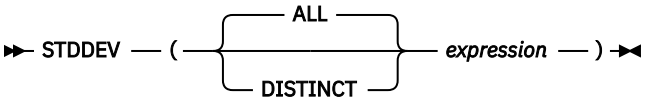
```
SELECT REGR_SLOPE(BONUS,SALARY), REGR_INTERCEPT(BONUS,SALARY),
REGR_R2(BONUS,SALARY), REGR_COUNT(BONUS,SALARY),
REGR_AVGX(BONUS,SALARY), REGR_AVGY(BONUS,SALARY),
REGR_SXX(BONUS,SALARY), REGR_SYY(BONUS,SALARY),
REGR_SXY(BONUS,SALARY)
INTO    :SLOPE, :ICPT,
:RSQR, :CNT,
:AVGSAL, :AVGBONUS,
:SXX, :SYY,
:SXY
FROM EMPLOYEE
WHERE WORKDEPT = 'A00'
```

When using the sample table, the host variables are set to the following approximate values:

```
SLOPE:  +1.71002671916749E-002
ICPT:  +1.00871888623260E+002
RSQR:  +9.99707928128685E-001
CNT:  3
AVGSAL:  +4.28333333333333E+004
AVGBONUS:  +8.33333333333333E+002
SXX:  +2.96291666666667E+008
SYY:  +8.66666666666667E+004
SXY:  +5.06666666666667E+006
```

STDDEV

The STDDEV function returns the biased standard deviation (division by *n*) of a set of numbers.



The schema is SYSIBM.

expression
An expression that returns a value of any built-in numeric data type.

If the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*); otherwise, the result is double-precision floating-point. The result can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are eliminated. When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the standard deviation of the values in the set.

The calculation that is used to determine the biased standard deviation is logically equivalent to the following formula:

```
STDDEV = SQRT(VARIANCE(expression))
```

where SQRT(VARIANCE(*expression*)) is the square root of the biased variance.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

STDDEV_POP can be specified in place of STDDEV.

Example

Set the host variable DEV to the standard deviation of the salaries of employees in department 'A00' in the EMPLOYEE table. The data type for the host variable DEV is double-precision floating point.

```
   SELECT STDDEV(SALARY)
     INTO :DEV
     FROM EMPLOYEE
     WHERE WORKDEPT = 'A00'
```

DEV is set to a number with an approximate value of 9938.00.

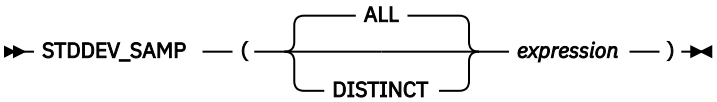
The following result set is shown for reference.

```
   SELECT SALARY FROM EMPLOYEE WHERE WORKDEPT = 'A00'
```

```
SALARY
-----------
  152750.00
   66500.00
   49250.00
   46500.00
   39250.00

  5 record(s) selected.
```

STDDEV_SAMP

The STDDEV_SAMP function returns the sample standard deviation (division by [*n*-1]) of a set of numbers.



The schema is SYSIBM.

expression
An expression that returns a value of any built-in numeric data type.

If the argument is DECFLOAT(*n*), the result is DECFLOAT(34); otherwise, the result is double-precision floating-point. The result can be null.

The function is applied to the set of values that are derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are eliminated. When the DISTINCT clause is interpreted for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number that is returned from the query is any one of the representations encountered (for example, either 123.00 or 123).

If the function is applied to an empty set or a set with only one row, the result is a null value. Otherwise, the result is the sample standard deviation of the values in the set.

The calculation that is used to determine the sample standard deviation is logically equivalent to the following formula:

```
   STDDEV_SAMP = SQRT(VARIANCE_SAMP(expression))
```

where SQRT(VARIANCE_SAMP(*expression*)) is the square root of the sample variance.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

Example

Set the host variable DEV to the sample standard deviation of the salaries for those employees in department 'A00' of the EMPLOYEE table. The data type for the host variable DEV is double-precision floating point.

```
SELECT STDDEV_SAMP(SALARY)
  INTO :DEV
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00'
```

DEV is set to a number with an approximate value of +4.68630318054647E+004.

The following result set is shown for reference.

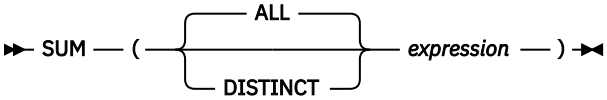
```
SELECT SALARY FROM EMPLOYEE WHERE WORKDEPT = 'A00'
```

```
SALARY
-----------
  152750.00
   66500.00
   49250.00
   46500.00
   39250.00

  5 record(s) selected.
```

SUM

The SUM function returns the sum of a set of numbers.

Note: The result of the function can be affected by the enablement of the large_aggregation configuration parameter.



The schema is SYSIBM.

expression
An expression that returns a value of any built-in numeric data type.

The data type of the result is the same as the data type of the input expression, with the following exceptions:

- If the data type of the input expression is SMALLINT, the data type of the result is INTEGER.
- If the data type of the input expression is single-precision floating point (REAL), the data type of the result is double-precision floating point (DOUBLE).
- If the data type of the input expression is DECFLOAT(*n*), the data type of the result is DECFLOAT(34).
- If the input expression is a DECIMAL value with precision *p* and scale *s*, the result is a DECIMAL with scale *s* and precision as follows:

Table 53. Result precision of the SUM aggregate function with DECIMAL input

DECIMAL arithmetic mode [1]	*p*	Result precision
default	n/a	31
DEC15	<=15	15
DEC15	>15	MIN(31,p+10)
DEC31	n/a	MIN(31,p+10)

Note:

1. These modes are determined by the **dec_arithmetic** configuration parameter.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are also eliminated. When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

The result can be null. If the function is applied to an empty set, the result is a null value. Otherwise, the result is the sum of the values in the set.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

Example

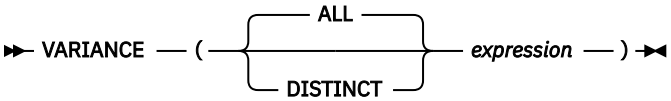
Using the EMPLOYEE table, set the host variable JOB_BONUS (decimal(9,2)) to the total bonus (BONUS) paid to clerks (JOB='CLERK').

```
SELECT SUM(BONUS)
  INTO :JOB_BONUS
  FROM EMPLOYEE
  WHERE JOB = 'CLERK'
```

Results in JOB_BONUS being set to 2800 when using the sample table.

VARIANCE

The VARIANCE function returns the biased variance (division by *n*) of a set of numbers.



The schema is SYSIBM.

expression
An expression that returns a value of any built-in numeric data type.

If the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*); otherwise, the result is double-precision floating-point. The result can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are eliminated. When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the variance of the values in the set.

The calculation that is used to determine the biased variance is logically equivalent to the following formula:

```
VARIANCE = SUM(expression**2)/COUNT(expression) - (SUM(expression)/
COUNT(expression))**2
```

The order in which the values are added is undefined, but every intermediate result must be within the range of the result data type.

VAR or VAR_POP can be specified in place of VARIANCE.

Example

Set the host variable VARNCE, whose data type is double-precision floating point, to the variance of the salaries for those employees in department 'A00' of the EMPLOYEE table.

```
SELECT VARIANCE(SALARY)
  INTO :VARNCE
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00'
```

Results in VARNCE being set to approximately 98763888.88 when using the sample table.

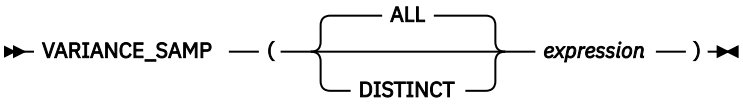
The following result set is shown for reference.

```
SELECT SALARY FROM EMPLOYEE WHERE WORKDEPT = 'A00'
```

```
SALARY
-----------
  152750.00
   66500.00
   49250.00
   46500.00
   39250.00

  5 record(s) selected.
```

VARIANCE_SAMP

The VARIANCE_SAMP function returns the sample variance (division by [*n*-1]) of a set of numbers.



The schema is SYSIBM.

expression
An expression that returns a value of any built-in numeric data type.

If the argument is DECFLOAT(*n*), the result is DECFLOAT(34); otherwise, the result is double-precision floating-point. The result can be null.

The function is applied to the set of values that are derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are eliminated. When the DISTINCT clause is interpreted for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number that is returned from the query is any one of the representations encountered (for example, 123.00 or 123).

If the function is applied to an empty set or a set with only one row, the result is a null value. Otherwise, the result is the sample variance of the values in the set.

The calculation that is used to determine the sample variance is logically equivalent to the following formula:

```
VARIANCE_SAMP = ( SUM(expression**2) - ((SUM(expression)**2) /
(COUNT(expression))) ) /
                  (COUNT(expression) - 1)
```

The order in which the values are added is undefined, but every intermediate result must be within the range of the result data type.

VAR_SAMP can be specified in place of VARIANCE_SAMP.

Example

Set the host variable VARNCE to the sample variance of the salaries for those employees in department 'A00' of the EMPLOYEE table. The data type for the host variable VARNCE is double-precision floating point.

```
SELECT VARIANCE_SAMP(SALARY)
  INTO :VARNCE
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00'
```

This statement results in VARNCE being set to approximately +2.19614375000000E+009 when the sample table is used.

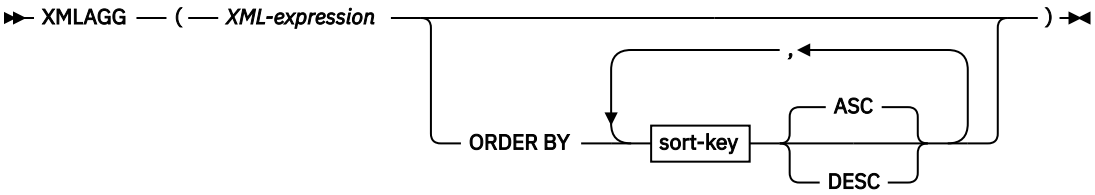
The following result set is shown for reference.

```
SELECT SALARY FROM EMPLOYEE WHERE WORKDEPT = 'A00'
```

```
SALARY
-----------
  152750.00
   66500.00
   49250.00
   46500.00
   39250.00

  5 record(s) selected.
```

XMLAGG

The XMLAGG function returns an XML sequence containing an item for each non-null value in a set of XML values.



The schema is SYSIBM. The function name cannot be specified as a qualified name.

XML-expression
Specifies an expression of data type XML. The data type of *XML-expression* cannot be a BINARY or VARBINARY type (SQLSTATE 42884).

ORDER BY
Specifies the order of the rows from the same grouping set that are processed in the aggregation. If the ORDER BY clause is omitted, or if the ORDER BY clause cannot distinguish the order of the column data, the rows in the same grouping set are arbitrarily ordered.

sort-key
The sort key can be a column name or a *sort-key-expression*. Note that if the sort key is a constant, it does not refer to the position of the output column (as in the ordinary ORDER BY clause), but it is simply a constant, which implies no sort key.

The data type of the result is XML.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If the *XML-expression* argument can be null, the result can be null. If the set of values is empty, the result is the null value. Otherwise, the result is an XML sequence containing an item for each value in the set.

If a SELECT clause includes an ARRAY_AGG function, then all invocations of ARRAY_AGG, LISTAGG, XMLAGG, and XMLGROUP functions in the same SELECT clause must specify the same order or not specify an order (SQLSTATE 428GZ).

Notes

- **Support in OLAP expressions**: XMLAGG cannot be used as a column function of an OLAP aggregation function (SQLSTATE 42601).

Example

Construct a department element for each department, containing a list of employees sorted by last name.

```
SELECT XMLSERIALIZE(
  CONTENT XMLELEMENT(
    NAME "Department", XMLATTRIBUTES(
      E.WORKDEPT AS "name"
    ),
    XMLAGG(
      XMLELEMENT(
        NAME "emp", E.LASTNAME
      )
      ORDER BY E.LASTNAME
    )
  )
  AS CLOB(110)
)
AS "dept_list"
FROM EMPLOYEE E
WHERE E.WORKDEPT IN ('C01','E21')
GROUP BY WORKDEPT
```

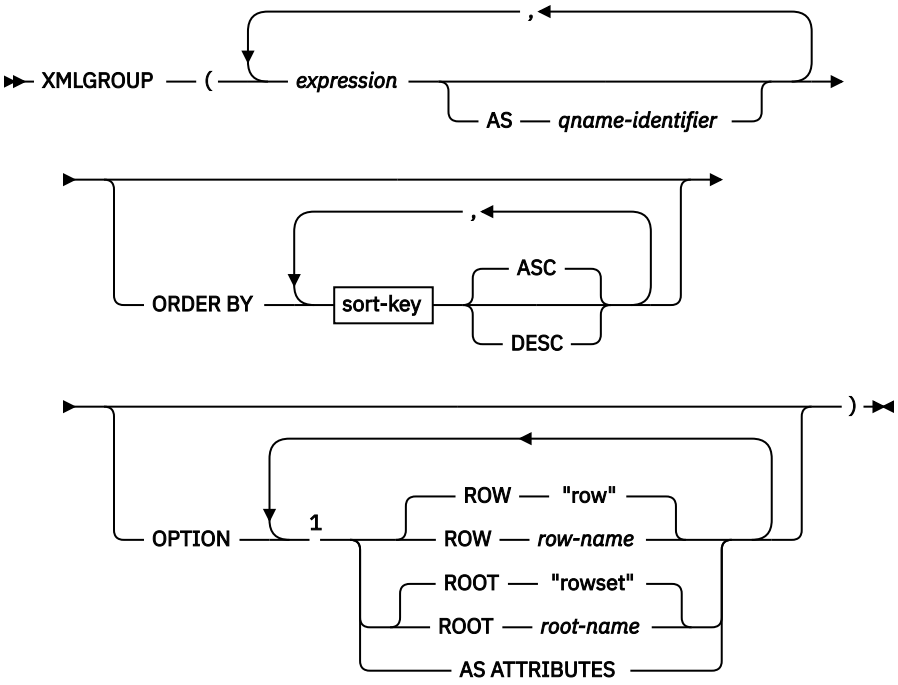
This query produces the following result:

```
dept_list
------------------------...
<Department name="C01">
  <emp>KWAN</emp>
  <emp>NICHOLLS</emp>
  <emp>QUINTANA</emp>
</Department>
<Department name="E21">
  <emp>GOUNOT</emp>
  <emp>LEE</emp>
  <emp>MEHTA</emp>
  <emp>SPENSER</emp>
</Department>
```

Note: XMLAGG does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

XMLGROUP

The XMLGROUP function returns an XML value with a single XQuery document node containing one top-level element node. This is an aggregate expression that will return a single-rooted XML document from a group of rows where each row is mapped to a row subelement.



Notes:

[1] The same clause must not be specified more than once.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

expression
The content of each generated XML element node (or the value of each generated attribute) is specified by an expression. The data type of *expression* cannot be a BINARY type, a VARBINARY type, or a structured type (SQLSTATE 42884). The expression can be any SQL expression. If the expression is not a simple column reference, a *qname-identifier* must be specified.

AS *qname-identifier*
Specifies the XML element name or attribute name as an SQL identifier. The *qname-identifier* must be of the form of an XML qualified name, or QName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope (SQLSTATE 42635). If *qname-identifier* is not specified, *expression* must be a column name (SQLSTATE 42703). The element name or attribute name is created from the column name using the fully escaped mapping from a column name to an QName.

OPTION
Specifies additional options for constructing the XML value. If no OPTION clause is specified, the default behavior applies.

ROW *row-name*
Specifies the name of the element to which each row is mapped. If this option is not specified, the default element name is "row".

ROOT *root-name*
Specifies the name of the root element node. If this option is not specified, the default root element name is "rowset"

AS ATTRIBUTES
Specifies that each expression is mapped to an attribute value with column name or *qname-identifier* serving as the attribute name.

ORDER BY
Specifies the order of the rows from the same grouping set that are processed in the aggregation. If the ORDER BY clause is omitted, or if the ORDER BY clause cannot distinguish the order of the column data, the rows in the same grouping set are arbitrarily ordered.

sort-key
The sort key can be a column name or a *sort-key-expression*. Note that if the sort key is a constant, it does not refer to the position of the output column (as in the ordinary ORDER BY clause), but it is simply a constant, which implies no sort key.

Rules

- If a SELECT clause includes an ARRAY_AGG function, then all invocations of ARRAY_AGG, LISTAGG, XMLAGG, and XMLGROUP functions in the same SELECT clause must specify the same order or not specify an order (SQLSTATE 428GZ).

Notes

The default behavior defines a simple mapping between a result set and an XML value. Some additional notes about function behavior apply:

- By default, each row is transformed into an XML element named "row" and each column is transformed into a nested element with the column name serving as the element name.
- The null handling behavior is NULL ON NULL. A null value in a column maps to the absence of the subelement. If all column values are null, no row element will be generated.
- The binary encoding scheme for BLOB and FOR BIT DATA data types is base64Binary encoding.
- By default, the elements corresponding to the rows in a group are children of a root element named "rowset".
- The order of the row subelements in the root element will be the same as the order in which the rows are returned in the query result set.
- A document node will be added implicitly to the root element to make the XML result a well-formed single-rooted XML document

Examples

The provided examples are based on the following table, T1, with integer columns C1 and C2 that contain numeric data stored in a relational format.

```
C1          C2
----------- -----------
          1           2
          -           2
          1           -
          -           -

  4 record(s) selected.
```

- *Example 1*: The following example shows an XMLGroup query and output fragment with default behavior, using a single top-level element to represent the table:

```
SELECT XMLGROUP(C1, C2)FROM T1
```

```
<rowset>
   <row>
      <C1>1</C1>
      <C2>2</C2>
   </row>
   <row>
      <C2>2</C2>
   </row>
   <row>
      <C1>1</C1>
   </row>
```

```
</rowset>

  1 record(s) selected.
```

- *Example 2*: The following example shows an XMLGroup query and output fragment with attribute centric mapping. Instead of appearing as nested elements as in the previous example, relational data is mapped to element attributes:

```
SELECT XMLGROUP(C1, C2 OPTION AS ATTRIBUTES) FROM T1
```

```
<rowset>
   <row C1="1" C2="2"/>
   <row C2="2"/>
   <row C1="1"/>
</rowset>

  1 record(s) selected.
```

- *Example 3*: The following example shows an XMLGroup query and output fragment with the default <rowset> root element replaced by <document> and the default <row> element replaced by <entry>. Columns C1 and C2 are returned as <column1> and <column2> elements, and the return set is ordered by column C1:

```
SELECT XMLGROUP(
   C1 AS "column1", C2 AS "column2"
   ORDER BY C1 OPTION ROW "entry" ROOT "document")
FROM T1
```

```
<document>
   <entry>
      <column1>1</column1>
      <column2>2</column2>
   </entry>
   <entry>
      <column1>1</column1>
   </entry>
   <entry>
      <column2>2</column2>
    </entry>
</document>
```

Scalar functions

A scalar function optionally accepts arguments and returns a single scalar value each time the function is called.

A scalar function can be used wherever an expression can be used. However, the restrictions that apply to the use of expressions and aggregate functions also apply when an expression or aggregate function is used within a scalar function. For example, the argument of a scalar function can be an aggregate function only if an aggregate function is allowed in the context in which the scalar function is used.

The restrictions on the use of aggregate functions do not apply to scalar functions, because a scalar function is applied to a single value rather than to a set of values.

The result of the following SELECT statement has as many rows as there are employees in department D01:

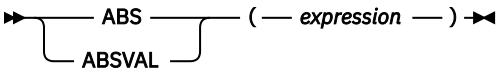
```
SELECT EMPNO, LASTNAME, YEAR(CURRENT DATE - BRTHDATE)
  FROM EMPLOYEE
  WHERE WORKDEPT = 'D01'
```

Scalar functions can be qualified with a schema name (for example, SYSIBM.CHAR(123)).

In a Unicode database, all scalar functions that accept a character or graphic string will accept any string types for which conversion is supported.

ABS or ABSVAL

Returns the absolute value of the argument.



The schema is SYSIBM.

The SYSFUN version of the ABS (or ABSVAL) function continues to be available.

expression
An expression that returns a value of any built-in numeric data type.

The result has the same data type and length attribute as the argument. The result can be null; if the argument is null, the result is the null value. If the argument is the maximum negative value for SMALLINT, INTEGER or BIGINT, the result is an overflow error.

Notes

Results involving DECFLOAT special values: For decimal floating-point values, the special values are treated as follows:

- ABS(NaN) and ABS(-NaN) return NaN.
- ABS(Infinity) and ABS(-Infinity) return Infinity.
- ABS(sNaN) and ABS(-sNaN) return sNaN.

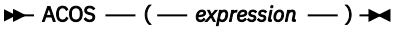
Example

```
ABS(-51234)
```

returns an INTEGER with a value of 51234.

ACOS

Returns the arccosine of the argument as an angle expressed in radians.



The schema is SYSIBM. (The SYSFUN version of the ACOS function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

Example

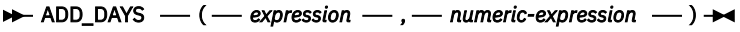
Assume that the host variable ACOSINE is a DECIMAL(10,9) host variable with a value of 0.070737202.

```
SELECT ACOS(:ACOSINE)
  FROM SYSIBM.SYSDUMMY1
```

This statement returns the approximate value 1.49.

ADD_DAYS

The ADD_DAYS function returns a datetime value that represents the first argument plus a specified number of days.



The schema is SYSIBM.

expression
An expression that specifies the starting date. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

numeric-expression
An expression that specifies the number of days to add to the starting date specified by *expression*. The expression must return a value that is a built-in numeric, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If the expression is not an INTEGER, it is cast to INTEGER before the function is evaluated. A negative numeric value can be used to subtract days.

The result of the function is a timestamp with the same precision as *expression* , if *expression* is a timestamp. Otherwise, the result of the function is a date. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

1. Assume that today is January 31, 2007. Set the host variable ADD_DAY with the current day plus 1 day.

   ```
   SET :ADD_DAY = ADD_DAYS(CURRENT_DATE, 1)
   ```

   The host variable ADD_DAY is set with the value representing 2007-02-01.

2. Assume that DATE is a host variable with the value July 27, 1965. Set the host variable ADD_DAY with the value of that day plus 3 days.

   ```
   SET :ADD_DAY = ADD_DAYS(:DATE,3)
   ```

   The host variable ADD_DAY is set with the value representing the day plus 3 days, 1965-07-30.

3. The ADD_DAYS function and datetime arithmetic can be used to achieve the same results. The following examples demonstrate this.

   ```
   SET :DATEHV = DATE('2008-2-28') + 4 DAYS
   SET :DATEHV = ADD_DAYS('2008-2-28', 4)
   ```

   In both cases, the host variable DATEHV is set with the value '2008-03-03'.

   Now consider the same examples but with the date '2008-2-29' as the argument.

   ```
   SET :DATEHV = DATE('2008-2-29') + 4 DAYS
   SET :DATEHV = ADD_DAYS('2008-2-29', 4)
   ```

   In both cases, the host variable DATEHV is set with the value '2008-03-04'.

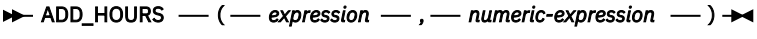
4. Assume that DATE is a host variable with the value July 27, 1965. Set the host variable ADD_DAY with the value of that day minus 3 days.

   ```
   SET :ADD_DAY = ADD_DAYS(:DATE,-3)
   ```

   The host variable ADD_DAY is set to 1965-07-24; the value representing July 27, 1965 minus 3 days.

ADD_HOURS

The ADD_HOURS function returns a timestamp value that represents the first argument plus a specified number of hours.



The schema is SYSIBM.

expression
An expression that specifies the starting timestamp. The expression must return a value that is a TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. The expression must not return a value that is a DATE (SQLSTATE 42815). If *expression* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

numeric-expression
An expression that specifies the number of hours to add to the starting timestamp specified by *expression*. The expression must return a value that is a built-in numeric, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If the expression is not an INTEGER, it is cast to INTEGER before the function is evaluated. A negative numeric value can be used to subtract hours.

The result of the function is a timestamp with the same precision as *expression*, if *expression* is a timestamp. Otherwise, the result is a TIMESTAMP(12). If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

1. Assume that the current timestamp is January 31, 2007, 01:02:03.123456. Set the host variable ADD_HOUR with the current timestamp plus 1 hour.

   ```
   SET :ADD_HOUR = ADD_HOURS(CURRENT TIMESTAMP, 1)
   ```

   The host variable ADD_HOUR is set with the value representing 2007-01-31-02.02.03.123456.

2. Assume that TIMESTAMP is a host variable with the value July 27, 1965 23:58:59. Set the host variable ADD_HOUR with the value of that timestamp plus 3 hours.

   ```
   SET :ADD_HOUR = ADD_HOURS(:TIMESTAMP,3)
   ```

   The host variable ADD_HOUR is set with the value representing the timestamp plus 3 hours, 1965-07-28-02.58.59.

3. The ADD_HOURS function and datetime arithmetic can be used to achieve the same results. The following examples demonstrate this.

   ```
   SET :TIMESTAMPHV = TIMESTAMP '2008-2-28-22.58.59' + 4 HOURS
   SET :TIMESTAMPHV = ADD_HOURS( TIMESTAMP '2008-2-28-22.58.59', 4)
   ```

   In both cases, the host variable TIMESTAMPHV is set with the value '2008-02-29-02.58.59'.

   Now consider the same examples but with 28 hours added.

   ```
   SET :TIMESTAMPHV = TIMESTAMP '2008-2-28-22.58.59' + 28 HOURS
   SET :TIMESTAMPHV = ADD_HOURS(TIMESTAMP '2008-2-28-22.58.59', 28)
   ```

   In both cases, the host variable TIMESTAMPHV is set with the value '2008-03-01-02.58.59'.

4. Assume that TIMESTAMP is a host variable with the value July 27, 1965 23:58:59. Set the host variable ADD_HOUR with the value of that timestamp minus 3 hours.

```
SET :ADD_HOUR = ADD_HOURS(:TIMESTAMP,-3)
```

The host variable ADD_HOUR is set to 1965-07-27-20.58.59; the value representing July 27, 1965 23:58:59 minus 3 hours.

ADD_MINUTES

The ADD_MINUTES function returns a timestamp value that represents the first argument plus a specified number of minutes.

The schema is SYSIBM.

expression
An expression that specifies the starting timestamp. The expression must return a value that is a TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. The expression must not return a value that is a DATE (SQLSTATE 42815). If *expression* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

numeric-expression
An expression that specifies the number of minutes to add to the starting timestamp specified by *expression*. The expression must return a value that is a built-in numeric, CHAR, or a VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If the expression is not a BIGINT, it is cast to BIGINT before the function is evaluated. A negative numeric value can be used to subtract minutes.

The result of the function is a timestamp with the same precision as *expression*, if *expression* is a timestamp. Otherwise, the result is a TIMESTAMP(12). If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

1. Assume that the current timestamp is January 31, 2007, 01:02:03.123456. Set the host variable ADD_MINUTE with the current timestamp plus 1 minute.

   ```
   SET :ADD_MINUTE = ADD_MINUTES(CURRENT TIMESTAMP, 1)
   ```

   The host variable ADD_MINUTE is set with the value representing 2007-01-31-01.03.03.123456.

2. Assume that TIMESTAMP is a host variable with the value July 27, 1965 23:58:59. Set the host variable ADD_MINUTE with the value of that timestamp plus 3 minutes.

   ```
   SET :ADD_MINUTE = ADD_MINUTES(:TIMESTAMP,3)
   ```

   The host variable ADD_MINUTE is set with the value representing the timestamp plus 3 minutes, 1965-07-28-00.01.59.

3. The ADD_MINUTES function and datetime arithmetic can be used to achieve the same results. The following examples demonstrate this.

   ```
   SET :TIMESTAMPHV = TIMESTAMP '2008-2-28-23.58.59' + 4 MINUTES
   SET :TIMESTAMPHV = ADD_MINUTES( TIMESTAMP '2008-2-28-23.58.59', 4)
   ```

   In both cases, the host variable TIMESTAMPHV is set with the value '2008-02-29-00.02.59'.

   Now consider the same examples but with 1442 minutes added.

   ```
   SET :TIMESTAMPHV = TIMESTAMP '2008-2-28-23.58.59' + 1442 MINUTES
   SET :TIMESTAMPHV = ADD_MINUTES(TIMESTAMP '2008-2-28-23.58.59', 1442)
   ```

In both cases, the host variable TIMESTAMPHV is set with the value '2008-03-01-00.00.59'.

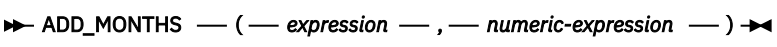
4. Assume that TIMESTAMP is a host variable with the value July 27, 1965 23:58:59. Set the host variable ADD_MINUTE with the value of that timestamp minus 3 minutes.

```
SET :ADD_MINUTE = ADD_MINUTES(:TIMESTAMP,-3)
```

The host variable ADD_MINUTE is set to 1965-07-27-23.55.59; the value representing July 27, 1965 23:58:59 minus 3 minutes.

ADD_MONTHS

The ADD_MONTHS function returns a datetime value that represents *expression* plus a specified number of months.



The schema is SYSIBM.

expression
An expression that specifies the starting date. The expression must return a value of one of the following built-in data types: a DATE or a TIMESTAMP.

numeric-expression
An expression that returns a value of any built-in numeric data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The *numeric-expression* specifies the number of months to add to the starting date specified by *expression*. A negative numeric value is allowed.

The result of the function has the same data type as *expression*, unless *expression* is a string, in which case the result data type is DATE. The result can be null; if any argument is null, the result is the null value.

If *expression* is the last day of the month or if the resulting month has fewer days than the day component of *expression*, the result is the last day of the resulting month. Otherwise, the result has the same day component as *expression*. Any hours, minutes, seconds or fractional seconds information included in *expression* is not changed by the function.

Examples

- *Example 1*: Assume today is January 31, 2007. Set the host variable ADD_MONTH with the last day of January plus 1 month.

```
SET :ADD_MONTH = ADD_MONTHS(LAST_DAY(CURRENT_DATE), 1);
```

The host variable ADD_MONTH is set with the value representing the end of February, 2007-02-28.

- *Example 2*: Assume DATE is a host variable with the value July 27, 1965. Set the host variable ADD_MONTH with the value of that day plus 3 months.

```
SET :ADD_MONTH = ADD_MONTHS(:DATE,3);
```

The host variable ADD_MONTH is set with the value representing the day plus 3 months, 1965-10-27.

- *Example 3*: The ADD_MONTHS function can be used to achieve similar results as datetime arithmetic. The following examples demonstrate the similarities and contrasts.

```
SET :DATEHV = DATE('2008-2-28') + 4 MONTHS;
SET :DATEHV = ADD_MONTHS('2008-2-28', 4);
```

In both cases, the host variable DATEHV is set with the value '2008-06-28'.

Now consider the same examples but with the date '2008-2-29' as the argument.

```
SET :DATEHV = DATE('2008-2-29') + 4 MONTHS;
```

The host variable DATEHV is set with the value '2008-06-29'.

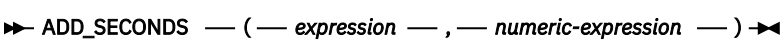
```
SET :DATEHV = ADD_MONTHS('2008-2-29', 4);
```

The host variable DATEHV is set with the value '2008-06-30'.

In this case, the ADD_MONTHS function returns the last day of the month, which is June 30, 2008, instead of June 29, 2008. The reason is that February 29 is the last day of the month. So, the ADD_MONTHS function returns the last day of June.

ADD_SECONDS

The ADD_SECONDS function returns a timestamp value that represents the first argument plus a specified number of seconds and fractional seconds.



The schema is SYSIBM.

expression
An expression that specifies the starting timestamp. The expression must return a value that is a TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. The *expression* must not return a value that is a DATE (SQLSTATE 42815). If expression is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function

numeric-expression
An expression that specifies the number of seconds and fractional seconds to add to the starting timestamp specified by *expression*. The expression must return a value that is a built-in numeric, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If the expression is not a DECIMAL(27,12), it is cast to DECIMAL(27,12) before the function is evaluated. A negative numeric value can be used to subtract seconds and fractional seconds.

The result of the function is a timestamp with the same precision as *expression*, if *expression* is a timestamp. Otherwise, the result is a TIMESTAMP(12). If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

1. Assume that the current timestamp is January 31, 2007, 01:02:03.123456. Set the host variable ADD_SECOND with the current timestamp plus 1 second.

   ```
   SET :ADD_SECOND = ADD_SECONDS(CURRENT TIMESTAMP, 1)
   ```

   The host variable ADD_SECOND is set with the value representing 2007-01-31-01.02.04.123456.
2. Assume that TIMESTAMP is a host variable with the value July 27, 1965 23:59:59.123456. Set the host variable ADD_SECOND with the value of that timestamp plus 3.123 seconds.

   ```
   SET :ADD_SECOND = ADD_SECONDS(:TIMESTAMP,3.123)
   ```

   The host variable ADD_SECOND is set with the value representing the timestamp plus 3.123 seconds, 1965-07-28-00.00.02.246456.

3. The ADD_SECONDS function and datetime arithmetic can be used to achieve the same results. The following examples demonstrate this.

```
SET :TIMESTAMPHV = TIMESTAMP '2008-2-28-23.58.59.123456' + 61.654321 SECONDS
```

```
SET :TIMESTAMPHV = ADD_SECONDS(
   TIMESTAMP '2008-2-28-23.58.59.123456', 61.654321)
```

   In both cases, the host variable TIMESTAMPHV is set with the value '2008-02-29-00.00.00.777777'.

   Now consider the same examples but with the timestamp '2008-2-29-23.59.59.123456' as the argument.

```
SET :TIMESTAMPHV = TIMESTAMP '2008-2-29-23.59.59.123456' + 61.654321 SECONDS
```

```
SET :TIMESTAMPHV = ADD_SECONDS(
   TIMESTAMP '2008-2-29-23.59.59.123456', 61.654321)
```

   In both cases, the host variable TIMESTAMPHV is set with the value '2008-03-01-00.01.00.777777'.

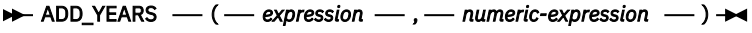
4. Assume that TIMESTAMP is a host variable with the value July 27, 1965 23:59:59.123456. Set the host variable ADD_SECOND with the value of that timestamp minus 3.123 seconds.

```
SET :ADD_SECOND = ADD_SECONDS(:TIMESTAMP,-3.123)
```

   The host variable ADD_SECOND is set to 1965-07-27-23.59.56.000456; the value representing July 27, 1965 23:59:59.123456 minus 3.123 seconds.

ADD_YEARS

The ADD_YEARS function returns a datetime value that represents the first argument plus a specified number of years.



The schema is SYSIBM.

expression
An expression that specifies the starting date. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

numeric-expression
An expression that specifies the number of years to add to the starting date specified by *expression*. The expression must return a value that is a built-in numeric, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If the expression is not an INTEGER, it is cast to INTEGER before the function is evaluated. A negative numeric value can be used to subtract years.

The result of the function is a timestamp with the same precision as *expression*, if *expression* is a timestamp. Otherwise, the result of the function is a date. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

If the result would be February 29 of a non-leap-year, the day is changed to 28, and a warning indicator in the SQLCA is set to indicate the adjustment.

Examples

1. Assume that today is January 31, 2007. Set the host variable ADD_YEAR with the current day plus 1 year.

```
SET :ADD_YEAR = ADD_YEARS(CURRENT_DATE, 1)
```

The host variable ADD_YEAR is set with the value representing 2008-01-31.

2. Assume that DATE is a host variable with the value July 27, 1965. Set the host variable ADD_YEAR with the value of that day plus 3 years.

```
SET :ADD_YEAR = ADD_YEARS(:DATE,3)
```

The host variable ADD_YEAR is set with the value representing the day plus 3 years, 1968-07-27.

3. The ADD_YEARS function and datetime arithmetic can be used to achieve the same results. The following examples demonstrate this.

```
SET :DATEHV = DATE('2008-2-29') + 4 YEARS
SET :DATEHV = ADD_YEARS('2008-2-29', 4)
```

In both cases, the host variable DATEHV is set with the value '2012-02-29'.

Now consider the same examples but with 3 years added.

```
SET :DATEHV = DATE('2008-2-29') + 3 YEARS
SET :DATEHV = ADD_YEARS('2008-2-29', 3)
```

In both cases, the host variable DATEHV is set with the value '2011-02-28'.

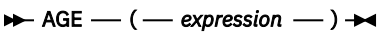
4. Assume that DATE is a host variable with the value July 27, 1965. Set the host variable ADD_YEAR with the value of that day minus 3 years.

```
SET :ADD_YEAR = ADD_YEARS(:DATE,-3)
```

The host variable ADD_YEAR is set to 1962-07-27; the value representing July 27, 1965 minus 3 years.

AGE

The AGE function returns a numeric value that represents the number of full years, full months, and full days between the current timestamp and the argument.



The schema is SYSIBM.

expression
An expression that specifies the datetime value for which the age is computed. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function

If there is less than a full day between the current timestamp and *expression*, the result is zero. If *expression* is earlier than the current timestamp, the result is positive. If *expression* is later than the current timestamp, the result is negative.

The result of the function is an INTEGER. If the argument can be null, the result can be null. If the argument is null, the result is the null value.

The AGE function is a synonym of the following expression:

```
INTEGER( ( CURRENT TIMESTAMP(12) - TIMESTAMP( expression, 12 ) ) / 1000000 )
```

The result is the integer representation of the extraction of the year, month, and day components of a timestamp duration.

Notes

- **Determinism**: AGE is a deterministic function. However, the invocation of the function depends on the value of the special register CURRENT TIMESTAMP. The AGE function can be used wherever special registers are supported (SQLSTATE 42621, 428EC, or 429BX).

Examples

1. Assume the CURRENT TIMESTAMP(12) is 2013-09-24-11.28.00.123456789012. Set the host variable AGE1 to the number of full years, full months, and full days between the current timestamp and 2012-02-28-12.00.00.

   ```
   SET :AGE1 = AGE(TIMESTAMP '2012-02-28-12.00.00')
   ```

   The host variable AGE1 is set to 10624.
2. Assume the CURRENT TIMESTAMP(12) is 2013-09-24-11.28.00.123456789012. Set the host variable AGE1 to the number of full years, full months, and full days between the current timestamp and 2013-09-23-12.00.00.

   ```
   SET :AGE1 = AGE(TIMESTAMP '2013-09-23-12.00.00')
   ```

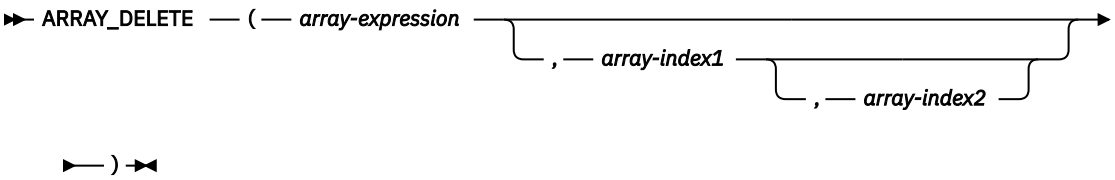
   The host variable AGE1 is set to 0.
3. Assume the CURRENT TIMESTAMP(12) is 2013-09-24-11.28.00.123456789012. Set the host variable AGE1 to the number of full years, full months, and full days between the current timestamp and 2020-01-01.

   ```
   SET :AGE1 = AGE(DATE '2020-01-01')
   ```

   The host variable AGE1 is set to -60306.

ARRAY_DELETE

The ARRAY_DELETE function deletes elements from an array.



The schema is SYSIBM.

array-expression
An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

array-index1
An expression that results in a value that can be assigned to the data type of the array index. If *array-expression* is an ordinary array, *array-index1* must be the null value (SQLTATE 42815).

array-index2
An expression that results in a value that can be assigned to the data type of the array index. If *array-expression* is an ordinary array, *array-index2* must be the null value. If *array-index2* is specified and is a non-null value, then *array-index1* must be a non-null value that is less than the value of *array-index2* (SQLSTATE 42815).

The result of the function has the same data type as *array-expression*. The result array is a copy of *array-expression*, with the following modifications:

- If the optional arguments are not specified or they are the null value, all of the elements of *array-expression* are deleted and the cardinality of the result array value is 0.

- If only *array-index1* is specified with a non-null value, the array element at index value *array-index1* is deleted.
- If *array-index2* is also specified with a non-null value, then the elements ranging from index value *array-index1* to *array-index2* (inclusive) are deleted.

The result can be null; if *array-expression* is null, the result is the null value.

Examples

1. Delete all the elements from the ordinary array variable RECENT_CALLS of array type PHONENUMBERS .

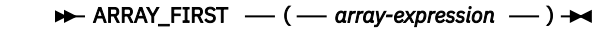
```
SETRECENT_CALLS = ARRAY_DELETE(RECENT_CALLS)
```

2. A supplier has discontinued some of their products. Delete the elements from the associative array variable FLOOR_TILES of array type PRODUCTS from index value 'PK5100' to index value 'PS2500'.

```
SETFLOOR_TILES = ARRAY_DELETE(FLOOR_TILES,'PK5100','PS2500')
```

ARRAY_FIRST

The ARRAY_FIRST function returns the minimum array index value of the array.



The schema is SYSIBM.

array-expression
An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

The data type of the result is the data type of the array index, which is INTEGER for an ordinary array. If *array-expression* is not null and the cardinality of the array is greater than zero, the value of the result is the minimum array index value, which is 1 for an ordinary array.

The result can be null; if *array-variable* is null or the cardinality of the array is zero, the result is the null value.

Examples

1. Return the first index value in the ordinary array variable SPECIALNUMBERS to the SQL variable E_CONSTIDX.

```
SET E_CONSTIDX = ARRAY_FIRST(SPECIALNUMBERS)
```

   The result is 1.

2. Given the associative array variable PHONELIST with index values and phone numbers: 'Home' is '4163053745', 'Work' is '4163053746', and 'Mom' is '416-4789683', assign the value of the minimum index in the array to the character string variable named X.

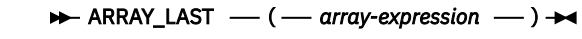
```
SET X = ARRAY_FIRST(PHONELIST)
```

   The value of 'Home' is assigned to X. Access the element value associated with index value 'Home' and assign it to the SQL variable NUMBER_TO_CALL:

```
SET NUMBER_TO_CALL = PHONELIST[X]
```

ARRAY_LAST

The ARRAY_LAST function returns the maximum array index value of the array.



The schema is SYSIBM.

array-expression
An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

The data type of the result is the data type of the array index, which is INTEGER for an ordinary array. If *array-expression* is not null and the cardinality of the array is greater than zero, the value of the result is the maximum array index value, which is the cardinality of the array for an ordinary array.

The result can be null; if array-variable is null or the cardinality of the array is zero, the result is the null value.

Examples

1. Return the last index value in the ordinary array variable SPECIALNUMBERS to the SQL variable PI_CONSTIDX.

   ```
   SET PI_CONSTIDX = ARRAY_LAST(SPECIALNUMBERS)
   ```

   The result is 10.
2. Given the associative array variable PHONELIST with index values and phone numbers: 'Home' is '4163053745', 'Work' is '4163053746', and 'Mom' is '4164789683', assign the value of the maximum index in the array to the character string variable named X.

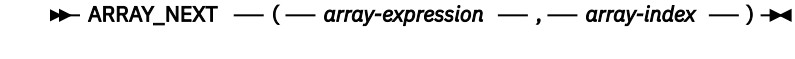
   ```
   SET X = ARRAY_LAST(PHONELIST)
   ```

   The value of 'Work' is assigned to X. Access the element value associated with index value 'Work' and assign it to the SQL variable NUMBER_TO_CALL:

   ```
   SET NUMBER_TO_CALL = PHONELIST[X]
   ```

ARRAY_NEXT

The ARRAY_NEXT function returns the next larger array index value for an array relative to the specified array index argument.



The schema is SYSIBM.

array-expression
An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

array-index
Specifies a value that is assignable to the data type of the index of the array. Valid values include any valid value for the data type.

The result is the next larger array index value defined in the array relative to the specified *array-index* value. If *array-index* is less than the minimum index array value in the array, the result is the first array index value defined in the array.

The data type of the result of the function is the data type of the array index. The result can be null; if either argument is null, the cardinality of the first argument is zero, or the value of *array-index* is greater than or equal to the value of the last index in the array, the result is the null value.

Examples

1. Return the next index value after the 9th index position in the ordinary array variable SPECIALNUMBERS to the SQL variable NEXT_CONSTIDX.

```
SET NEXT_CONSTIDX = ARRAY_NEXT(SPECIALNUMBERS,9)
```

   The result is 10.

2. Given the associative array variable PHONELIST with index values and phone numbers: 'Home' is '4163053745', 'Work' is '4163053746', and 'Mom' is '416-4789683', assign the value of the index in the array that is the next index after index value 'Dad', which does not exist for the array value, to the character string variable named X:

```
SET X = ARRAY_NEXT(PHONELIST, 'Dad')
```

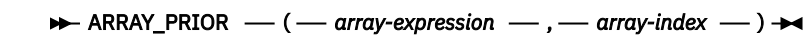
   The value of 'Home' is assigned to X, since the value 'Dad' is a value smaller than any index value for the array variable. Assign the value of the index in the array that is the next index after index value'Work':

```
SET X = ARRAY_NEXT(PHONELIST, 'Work')
```

   The null value is assigned to X.

ARRAY_PRIOR

The ARRAY_PRIOR function returns the next smaller array index value for an array relative to the specified array index argument.



The schema is SYSIBM.

array-expression
An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

array-index
Specifies a value that is assignable to the data type of the index of the array. Valid values include any valid value for the data type.

The result is the next smaller array index value defined in the array relative to the specified *array-index* value. If *array-index* is greater than the maximum index array value in the array, the result is the last array index value defined in the array.

The data type of the result of the function is the data type the array index. The result can be null; if either argument is null, the cardinality of the first argument is zero, or the value of *array-index* is less than or equal to the value of the first index in the array, the result is the null value.

Examples

1. Return the previous index value before the 2nd index position in the ordinary array variable SPECIALNUMBERS to the SQL variable PREV_CONSTIDX.

```
SET PREV_CONSTIDX = ARRAY_PRIOR(SPECIALNUMBERS,2)
```

   The result is 1.

2. Given the associative array variable PHONELIST with index values and phone numbers: 'Home' is '4163053745', 'Work' is '4163053746', and 'Mom' is '416-4789683', assign the value of the index in the array that is the previous index before index value 'Work' to the character string variable named X:

```
SET X = ARRAY_PRIOR(PHONELIST, 'Work')
```

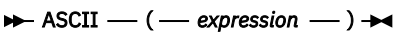
The value of 'Mom' is assigned to X. Assign the value of the index in the array that is the previous index before index value'Home':

```
SET X = ARRAY_PRIOR(PHONELIST, 'Home')
```

The null value is assigned to X.

ASCII

The ASCII function returns the ASCII code value of the leftmost character of the argument as an integer.



The schema is SYSFUN.

expression
An expression that returns a built-in character string or Boolean value. In a Unicode database, the expression can also return a graphic string, in which case it is first converted to a character string before the function is evaluated. The maximum length is:

- 4000 bytes for a VARCHAR
- 1,048,576 bytes for a CLOB

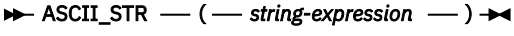
Result

The result of the function is always INTEGER.

The result can be null; if the argument is null, the result is the null value.

ASCII_STR

The ASCII_STR function returns an ASCII version of the string.



The schema is SYSIBM.

string-expression
An expression that returns a value of a built-in character or graphic string.

A character string must not be bit data (SQLSTATE 42815).

The argument can also be a numeric data type. The numeric argument is implicitly cast to a VARCHAR data type.

ASCII_STR returns an ASCII version of the string. Non-ASCII characters are converted to UTF-16 characters and appear in the result in the form of *xxxx* (or *xxxx**yyyy* for surrogate characters), where *xxxx* and *yyyy* represent a UTF-16 code unit. A backslash in a string-expression is converted to a double backslash.

The string unit of the result is OCTETS.

The length attribute of the result is MIN((5**n*),32672), where *n* is determined as follows:

- If a string-expression has the string units OCTETS or CODEUNITS16, *n* is the length attribute of the input string.
- If a string-expression has the string units CODEUNITS32, *n* is twice the length attribute of the input string.

Result

The result of the function is a varying-length character string.

If the actual length of the result string exceeds the maximum for the return type, an error occurs(SQLSTATE 54006).

If the argument can be null, the result can be null. If the argument is null, the result is the null value.

Notes

As a syntax alternative, you can specify ASCIISTR as a synonym for ASCII_STR.

The following example returns the ASCII string equivalent of the Unicode (UTF-8) string, '4869206D616D6520697320D090D0BDD0B4D180D0B5D0B9202020F0908080':

```
SET :HV1 = ASCII_STR(X'4869206D616D6520697320D090D0BDD0B4D180D0B5D0B9202020F0908080');
```

:HV1 is assigned the value 'Hi, my name is \0410\043D\0434\0440\0435\0439 \D800\DC00'.

In this example, the UTF-8 characters D090, D0BD, D0B4, D180, D0B5, and D0B9 are converted to \0410\043D\0434\0440\0435\0439, and the non-ASCII character F0908080 is converted to \D800\DC00.

```
SET :HV1 = ASCII_STR('Hi, my name is Андрей(Andrei)');
```

:HV1 is assigned the value "Hi, my name is \0410\043D\0434\0440\0435\0439 (Andrei)"

ASIN

Returns the arcsine on the argument as an angle expressed in radians.

The schema is SYSIBM. (The SYSFUN version of the ASIN function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

ATAN

Returns the arctangent of the argument as an angle expressed in radians.

The schema is SYSIBM. (The SYSFUN version of the ATAN function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

ATAN2

Returns the arctangent of x and y coordinates as an angle expressed in radians. The x and y coordinates are specified by the first and second arguments, respectively.

The schema is SYSIBM. (The SYSFUN version of the ATAN2 function continues to be available.)

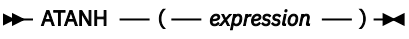
expression1
An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

expression2
An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

ATANH

Returns the hyperbolic arctangent of the argument, where the argument is an angle expressed in radians.



The schema is SYSIBM.

expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

BIGINT

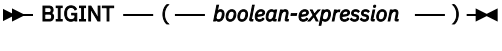
The BIGINT function returns a big integer (a binary integer with a precision of 63 bits) representation of a value of a different data type.

Numeric to BIGINT

String to BIGINT

Datetime to BIGINT

Boolean to BIGINT



The schema is SYSIBM.

Numeric to BIGINT

numeric-expression
An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a big integer column or variable. The fractional part of the argument is truncated. If the whole part of the argument is not within the range of big integers, an error is returned (SQLSTATE 22003).

String to BIGINT

string-expression
An expression that returns a value that is a character-string or Unicode graphic-string representation of a number with a length not greater than the maximum length of a character constant.

The result is the same number that would result from CAST(*string-expresssion* AS BIGINT). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018). If the whole part of the argument is not within the range of big integers, an error is returned (SQLSTATE 22003). The data type of *string-expresssion* must not be CLOB or DBCLOB (SQLSTATE 42884).

Datetime to BIGINT

datetime-expression
An expression that is of one of the following data types:

- DATE. The result is a BIGINT value representing the date as *yyyymmdd*.
- TIME. The result is a BIGINT value representing the time as *hhmmss*.
- TIMESTAMP. The result is a BIGINT value representing the timestamp as *yyyymmddhhmmss*. The fractional seconds portion of the timestamp value is not included in the result.

Boolean to BIGINT

boolean-expression
An expression that returns a Boolean value (TRUE or FALSE). The result is either 1 (for TRUE) or 0 (for FALSE).

Result

The result of the function is a big integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Notes

- **Increasing portability of applications:** If the first argument is numeric, or if the first argument is a string and the length argument is specified, use the "CAST specification" on page 152 instead of this function to increase the portability of your applications.

Examples

- *Example 1*: From ORDERS_HISTORY table, count the number of orders and return the result as a big integer value.

```
SELECT BIGINT (COUNT_BIG(*))
  FROM ORDERS_HISTORY
```

- *Example 2*: Using the EMPLOYEE table, select the EMPNO column in big integer form for further processing in the application.

```
SELECT BIGINT (EMPNO) FROM EMPLOYEE
```

- *Example 3*: Assume that the column RECEIVED (whose data type is TIMESTAMP) has an internal value equivalent to '1988-12-22-14.07.21.136421'.

```
BIGINT(RECEIVED)
```

results in the value 19 881 222 140 721.

- *Example 4*: Assume that the column STARTTIME (whose data type is TIME) has an internal value equivalent to '12:03:04'.

```
BIGINT(STARTTIME)
```

  results in the value 120 304.

- *Example 5*: The following statement returns the value 1 of data type BIGINT.

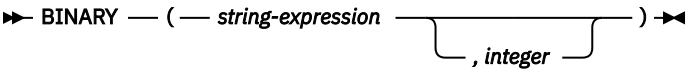
```
values BIGINT(TRUE)
```

- *Example 6*: The following statement returns the value 0 of data type BIGINT.

```
values BIGINT(3>3)
```

BINARY

The BINARY function returns a fixed-length binary string representation of a string of any data type.



The schema is SYSIBM.

string-expression
An expression that returns a value of a character string, graphic string, or binary string data type.

integer
An integer constant value, which specifies the length attribute of the resulting BINARY data type. The value must be 1 - 255. If *integer* is not specified, the length attribute of the result is the lower of the following values:

- The maximum length for the BINARY data type
- The length attribute for the data type of *string-expression* expressed in bytes:
  - The length attribute, if *string-expression* is a binary string, a character string that is FOR BIT DATA, or a character string with string units of OCTETS
  - The length attribute multiplied by 2, if *string-expression* is a graphic string with string units of CODEUNITS16 or double bytes
  - The length attribute multiplied by 4, if *string-expression* is a character or graphic string with string units of CODEUNITS32

If *string-expression* is an empty string and the *integer* argument is not specified, an error is returned (SQLSTATE 42815).

The result of the function is a BINARY. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The actual length is the same as the length attribute of the result. If the length of *string-expression* that is converted to a binary string is less than the length attribute of the result, the result is padded with hexadecimal zeros up to the length of the result. If the length of *string-expression* that is converted to a binary string is greater than the length attribute of the result, truncation occurs.

A warning (SQLSTATE 01004) is returned in the following situations:

- The first argument is a character or graphic string (other than a CLOB or DBCLOB) and non-blank characters are truncated.
- The first argument is a binary string (other than BLOB) and non-hexadecimal zeros are truncated.

Examples

1. The following function returns a fixed-length binary string with a length attribute 1 and a value BX'00'.

```
SELECT BINARY('',1)
  FROM SYSIBM.SYSDUMMY1
```

2. The following function returns a fixed-length binary string with a length attribute 5 and a value BX'4B42480000'.

```
SELECT BINARY('KBH',5)
  FROM SYSIBM.SYSDUMMY1
```

3. The following function returns a fixed-length binary string with a length attribute 3 and a value BX'4B4248'.

```
SELECT BINARY('KBH')
  FROM SYSIBM.SYSDUMMY1
```

4. The following function returns a fixed-length binary string with a length attribute 3 and a value BX'4B4248'.

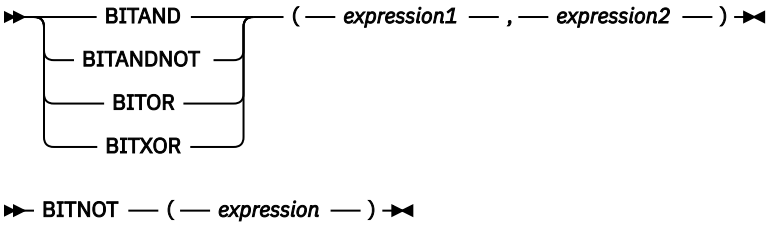
```
SELECT BINARY('KBH ',3)
  FROM SYSIBM.SYSDUMMY1
```

5. The following function returns a fixed-length binary string with a length attribute 3, a value BX'4B4248', and a warning (SQLSTATE 01004).

```
SELECT BINARY('KBH 93',3)
  FROM SYSIBM.SYSDUMMY1
```

BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT

These bitwise functions operate on the "two's complement" representation of the integer value of the input arguments and return the result as a corresponding base 10 integer value in a data type based on the data type of the input arguments.



The schema is SYSIBM.

Table 54. The bit manipulation functions

Function	Description	A bit in the two's complement representation of the result is:
BITAND	Performs a bitwise AND operation.	1 only if the corresponding bits in both arguments are 1.
BITANDNOT	Clears any bit in the first argument that is in the second argument.	Zero if the corresponding bit in the second argument is 1; otherwise, the result is copied from the corresponding bit in the first argument.
BITOR	Performs a bitwise OR operation.	1 unless the corresponding bits in both arguments are zero.

Table 54. The bit manipulation functions (continued)		
Function	**Description**	**A bit in the two's complement representation of the result is:**
BITXOR	Performs a bitwise exclusive OR operation.	1 unless the corresponding bits in both arguments are the same.
BITNOT	Performs a bitwise NOT operation.	Opposite of the corresponding bit in the argument.

expression* or *expression1* or *expression2
The arguments must be integer values represented by the data types SMALLINT, INTEGER, BIGINT, or DECFLOAT. Arguments of type DECIMAL, REAL, or DOUBLE are cast to DECFLOAT. The value is truncated to a whole number.

The bit manipulation functions can operate on up to 16 bits for SMALLINT, 32 bits for INTEGER, 64 bits for BIGINT, and 113 bits for DECFLOAT. The range of supported DECFLOAT values includes integers from -2^{112} to 2^{112} -1, and special values such as NaN or INFINITY are not supported (SQLSTATE 42815). If the two arguments have different data types, the argument supporting fewer bits is cast to a value with the data type of the argument supporting more bits. This cast impacts the bits that are set for negative values. For example, -1 as a SMALLINT value has 16 bits set to 1, which when cast to an INTEGER value has 32 bits set to 1.

The result of the functions with two arguments has the data type of the argument that is highest in the data type precedence list for promotion. If either argument is DECFLOAT, the data type of the result is DECFLOAT(34). If either argument can be null, the result can be null; if either argument is null, the result is the null value.

The result of the BITNOT function has the same data type as the input argument, except that DECIMAL, REAL, DOUBLE, or DECFLOAT(16) returns DECFLOAT(34). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Due to differences in internal representation between data types and on different hardware platforms, using functions (such as HEX) or host language constructs to view or compare internal representations of BIT function results and arguments is data type-dependent and not portable. The data type- and platform-independent way to view or compare BIT function results and arguments is to use the actual integer values.

Use of the BITXOR function is recommended to toggle bits in a value. Use the BITANDNOT function to clear bits. BITANDNOT(val, pattern) operates more efficiently than BITAND(val, BITNOT(pattern)).

Examples

The following examples are based on an ITEM table with a PROPERTIES column of type INTEGER.

- *Example 1*: Return all items for which the third property bit is set.

```
SELECT ITEMID FROM ITEM
  WHERE BITAND(PROPERTIES, 4) = 4
```

- *Example 2*: Return all items for which the fourth or the sixth property bit is set.

```
SELECT ITEMID FROM ITEM
  WHERE BITAND(PROPERTIES, 40) <> 0
```

- *Example 3*: Clear the twelfth property of the item whose ID is 3412.

```
UPDATE ITEM
    SET PROPERTIES = BITANDNOT(PROPERTIES, 2048)
  WHERE ITEMID = 3412
```

- *Example 4*: Set the fifth property of the item whose ID is 3412.

```
UPDATE ITEM
    SET PROPERTIES = BITOR(PROPERTIES, 16)
  WHERE ITEMID = 3412
```

- *Example 5*: Toggle the eleventh property of the item whose ID is 3412.

```
UPDATE ITEM
    SET PROPERTIES = BITXOR(PROPERTIES, 1024)
  WHERE ITEMID = 3412
```

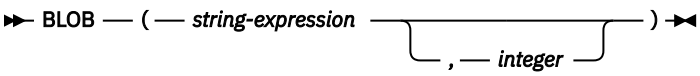
- *Example 6*: Switch all the bits in a 16-bit value that has only the second bit on.

```
VALUES BITNOT(CAST(2 AS SMALLINT))
```

returns -3 (with a data type of SMALLINT).

BLOB

The BLOB function returns a BLOB representation of a string of any type.



The schema is SYSIBM.

string-expression
An expression that returns a value of a character string, graphic string, or binary string data type.

integer
An integer value specifying the length attribute of the resulting BLOB data type. If *integer* is not specified, the length attribute of the result is the same as the length of the input, except where the input is graphic. In this case, the length attribute of the result is twice the length of the input.

The result of the function is a BLOB. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

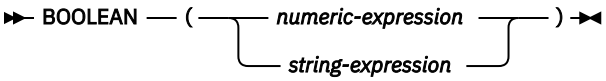
Example

Given a table with a BLOB column named TOPOGRAPHIC_MAP and a VARCHAR column named MAP_NAME, locate any maps that contain the string 'Pellow Island' and return a single binary string with the map name concatenated in front of the actual map.

```
SELECT BLOB(MAP_NAME  CONCAT ':  ') CONCAT TOPOGRAPHIC_MAP
  FROM ONTARIO_SERIES_4
  WHERE TOPOGRAPHIC_MAP LIKE BLOB('%Pellow Island%')
```

BOOLEAN

The BOOLEAN function returns the actual Boolean value that corresponds to a non-Boolean representation of a Boolean value.



The schema is SYSIBM.

numeric-expression
An expression that returns a binary integer or floating decimal value. The result is TRUE if the returned value is non-zero and FALSE if it is zero.

string-expression
An expression that returns a character-string or Unicode graphic-string representation of a Boolean value. Leading and trailing blanks are eliminated from the string before it is evaluated.

The returned string must be a valid representation of a Boolean value as described in "Boolean values" on page 41 (SQLSTATE 22018).

The result of the function is the same BOOLEAN value that would result from the expression:

```
CAST(string-expression AS BOOLEAN)
```

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

- *Example 1*: The following statement returns a value of data type BOOLEAN with the value TRUE.

```
values BOOLEAN(1)
```

- *Example 2*: The following statement returns a value of data type BOOLEAN with the value FALSE.

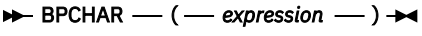
```
values BOOLEAN('NO')
```

- *Example 3*: The following statement returns a value of data type BOOLEAN with the value TRUE.

```
values BOOLEAN('Yes')
```

BPCHAR

The BPCHAR function returns a varying-length character string representation of a value of a different data type.

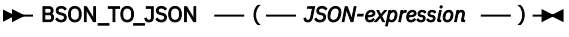


The schema is SYSIBM.

The BPCHAR scalar function is a synonym for the VARCHAR scalar function.

BSON_TO_JSON

The BSON_TO_JSON function converts a string that contains data that is formatted as BSON to a character string that contains data that is formatted as JSON.



Although the schema for this function is SYSIBM, the function cannot be specified as a qualified name.

JSON-expression
Specifies an expression that returns a binary string value. It must contain formatted BSON data (SQLSTATE 22032).

If *JSON-expression* can be null, the result can be null; if *JSON-expression* is null, the result is the null value.

Notes

- If parameter markers are not explicitly cast to a supported data type, an error is returned (SQLSTATE 42815)

Example

1. Retrieve a JSON document in string format, from a table where the data is stored in binary representation.

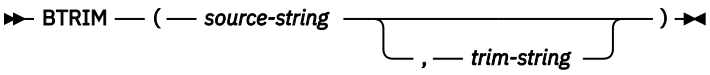
```
SELECT JSON_FIELD FROM TESTJSON;

JSON_FIELD
--------------------
x'16000000024E616D65000700000047656F7267650000'
```

   To extract the contents of a JSON field, use the BSON_TO_JSON function.

```
SELECT BSON_TO_JSON(JSON_FIELD) FROM TESTJSON;

1
--------------------
{ "Name" : "George" }
```

BTRIM

The BTRIM function removes the characters that are specified in a trim string from the beginning and end of a source string.



The schema is SYSIBM.

This function compares the binary representation of each character (consisting of one or more bytes) in the trim-string to the binary representation of each character (consisting of one or more bytes) at the beginning and end of the source string. The database collation does not affect the search. If the string-expression is defined as FOR BIT DATA, the search compares each byte in the trim-expression to the byte at the beginning and end of the string-expression.

source-string
An expression that specifies the string from which characters are to be removed. This expression must return a built-in character string, graphic string, numeric value, Boolean value, or datetime value. If the source string is:

- A numeric, Boolean, or datetime value, it is implicitly cast to VARCHAR before the function is evaluated
- A CLOB value, the length of the value is limited to the maximum size of a VARCHAR (SQLSTATE 22001)
- A DBCLOB value, the actual length of the value is limited to the maximum size of a VARGRAPHIC (SQLSTATE 22001)

trim-string
An expression that specifies the characters that are to be removed from the beginning and end of the source string. The expression must return a built-in character string, graphic string, numeric value, or datetime value. If the trim string is:

- Not a character string or graphic string, it is implicitly cast to VARCHAR before the function is evaluated
- A CLOB, the actual length of the value is limited to the maximum size of a VARCHAR (SQLSTATE 22001)
- A DBCLOB, the actual length of the value is limited to the maximum size of a VARGRAPHIC (SQLSTATE 22001)

The type of the source string determines the default trim string:

Type of source string	Default trim string
A graphic string in a DBCS or EUC database	double-byte blank
A graphic string in a Unicode database	UCS-2 blank
A FOR BIT DATA string	X'20'
All other cases	single-byte blank

Restrictions:

- If the source string is not defined as FOR BIT DATA, then the trim string cannot be defined as FOR BIT DATA (SQLSTATE 42815).
- If one parameter (source string or trim string) is character FOR BIT DATA, then the other parameter cannot be a graphic (SQLSTATE 42846).
- A combination of character string and graphic string arguments can be used only in a Unicode database (SQLSTATE 42815).

Result

The data type of the source string determines the data type of the result:

Data type of source string	Data type of result
VARCHAR or CHAR	VARCHAR
CLOB	CLOB
VARGRAPHIC or GRAPHIC	VARGRAPHIC
DBCLOB	DBCLOB

The length attribute of the data type of the result is the same as the length attribute of the data type of the source string. The length of the result is the length of the source string minus the number of string units that were removed. If all of the characters are removed, the result is an empty string with a length of zero.

If any argument can be null, the result can be null. If any argument is null, the result is the null value.

Example

The host variable BALANCE1 is of type CHAR(9) and has the value `'000345.50'`. The following statement returns the value `'345.5'`:

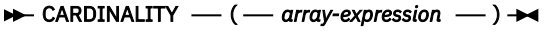
```
SELECT BTRIM(:BALANCE1, '0')
  FROM SYSIBM.SYSDUMMY1
```

The host variable BALANCE2 is of type CHAR(9) and has the value `'  345.50'`. The following statement returns the value `'345.50'`.

```
SELECT BTRIM(:BALANCE2)
  FROM SYSIBM.SYSDUMMY1
```

CARDINALITY

The CARDINALITY function returns a value of type BIGINT representing the number of elements of an array.



The schema is SYSIBM.

array-expression
An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

Result

For an:

- Ordinary array, the returned value is the highest array index for which the array has an assigned element. This includes elements that have been assigned the null value.
- Associative array, the returned value is the actual number of unique array index values defined in the array expression.

The function returns 0 if the array is empty. The result can be null; if the argument is null, the result is the null value.

Examples

1. Return the number of calls that have been stored in the recent calls list so far:

   ```
   SET HOWMANYCALLS = CARDINALITY(RECENT_CALLS)
   ```

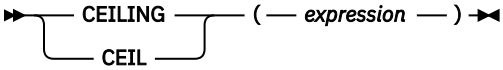
   The SQL variable HOWMANYCALLS contains the value 3.
2. Assume that the associative array variable CAPITALS of array type CAPITALSARRAY contains all of the capitals for the 10 provinces and 3 territories in Canada as well as the capital of the country, Ottawa. Return the cardinality of the array variable:

   ```
   SET NUMCAPITALS = CARDINALITY(CAPITALS)
   ```

   The SQL variable NUMCAPITALS contains the value 14.

CEILING or CEIL

Returns the smallest integer value greater than or equal to the argument.



The schema is SYSIBM. (The SYSFUN version of the CEILING function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type.

The result of the function has the same data type and length attribute as the argument except that the scale is 0 if the argument is DECIMAL. For example, an argument with a data type of DECIMAL(5,5) returns DECIMAL(5,0).

The result can be null if the argument can be null or if the argument is not a decimal floating-point number and the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

Notes

- **Results involving DECFLOAT special values:** For decimal floating-point values, the special values are treated as follows:
  - CEILING(NaN) returns NaN.
  - CEILING(-NaN) returns -NaN.
  - CEILING(Infinity) returns Infinity.
  - CEILING(-Infinity) returns -Infinity.

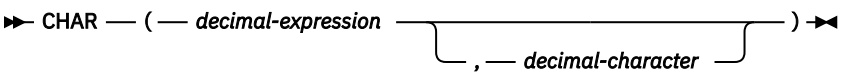
– CEILING(sNaN) returns NaN and a warning.
– CEILING(-sNaN) returns -NaN and a warning.

CHAR

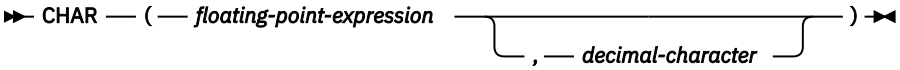
The CHAR function returns a fixed-length character string representation of a value of a different data type.

Integer to CHAR

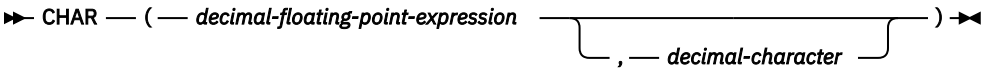
Decimal to CHAR



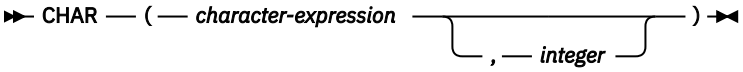
Floating-point to CHAR



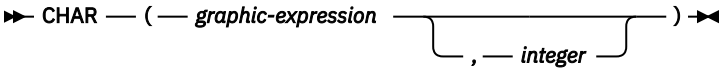
Decimal floating-point to CHAR



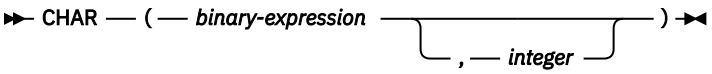
Character string to CHAR



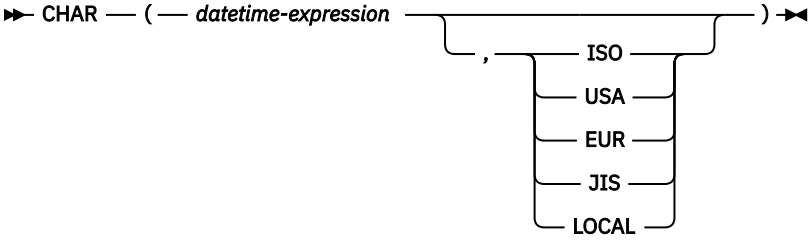
Graphic string to CHAR



Binary string to CHAR



Datetime to CHAR



Boolean to CHAR

The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature. The SYSFUN.CHAR(*floating-point-expression*) signature continues to be available. In this case, the decimal character is locale sensitive, and therefore returns either a period or a comma, depending on the locale of the database server.

Integer to CHAR

integer-expression
An expression that returns a value that is of an integer data type (SMALLINT, INTEGER, or BIGINT).

The result is a fixed-length character string representation of *integer-expression* in the form of an SQL integer constant. The result consists of *n* characters, which represent the significant digits in the argument, and is preceded by a minus sign if the argument is negative. The result is left-aligned. If the data type of the first argument is:

- SMALLINT, the length of the result is 6
- INTEGER, the length of the result is 11
- BIGINT, the length of the result is 20

If the number of bytes in the result is less than the defined length of the result, the result is padded on the right with single-byte blanks.

The code page of the result is the code page of the section.

Decimal to CHAR

decimal-expression
An expression that returns a value that is a decimal data type. If a different precision and scale are required, the DECIMAL scalar function can be used first to make the change.

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a fixed-length character string representation of *decimal-expression* in the form of an SQL decimal constant. The length of the result is 2+*p*, where *p* is the precision of *decimal-expression*. Leading zeros are not included. Trailing zeros are included. If *decimal-expression* is negative, the first character of the result is a minus sign; otherwise, the first character is a digit or the decimal character. If the scale of *decimal-expression* is zero, the decimal character is not returned. If the number of bytes in the result is less than the defined length of the result, the result is padded on the right with single-byte blanks.

The code page of the result is the code page of the section.

Floating-point to CHAR

floating-point-expression
An expression that returns a value that is a floating-point data type (DOUBLE or REAL).

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a fixed-length character string representation of *floating-point-expression* in the form of an SQL floating-point constant. The length of the result is 24. The result is the smallest number of characters that can represent the value of *floating-point-expression* such that the mantissa consists of a single digit other than zero followed by a period and a sequence of digits. If *floating-point-expression* is negative, the first character of the result is a minus sign; otherwise, the first character is a digit. If

floating-point-expression is zero, the result is 0E0. If the number of bytes in the result is less than 24, the result is padded on the right with single-byte blanks.

The code page of the result is the code page of the section.

Decimal floating-point to CHAR

decimal-floating-point-expression
An expression that returns a value that is a decimal floating-point data type (DECFLOAT).

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a fixed-length character string representation of *decimal-floating-point-expression* in the form of an SQL decimal floating-point constant. The length attribute of the result is 42. The result is the smallest number of characters that can represent the value of *decimal-floating-point-expression*. If *decimal-floating-point-expression* is negative, the first character of the result is a minus sign; otherwise, the first character is a digit. If *decimal-floating-point-expression* is zero, the result is 0.

If the value of *decimal-floating-point-expression* is the special value Infinity, sNaN, or NaN, the strings 'INFINITY', 'SNAN', and 'NAN', respectively, are returned. If the special value is negative, the first character of the result is a minus sign. The decimal floating-point special value sNaN does not result in warning when converted to a string. If the number of characters in the result is less than 42, the result is padded on the right with single-byte blanks.

The code page of the result is the code page of the section.

Character string to CHAR

character-expression
An expression that returns a value that is a built-in character string data type.

integer
An integer constant that specifies the length attribute for the resulting fixed-length character string. The value must be between 0 and the maximum length for the CHAR data type in the string units of the result.

If the second argument is not specified:

- If the *character-expression* is the empty string constant, the length attribute of the result is 0.
- Otherwise, the length attribute of the result is the lower of the following values:
  - The maximum length for the CHAR data type in the string units of the result
  - The length attribute of the first argument.

The result is a fixed-length character string. If *character-expression* is FOR BIT DATA, the result is FOR BIT DATA.

The length of the result is the same as the length attribute of the result. If the length of *character-expression* is:

- Less than the length attribute of the result, the result is padded with blanks up to the length attribute of the result
- Greater than the length attribute of the result:
  - If the string unit of the result is CODEUNITS32, truncation is performed. If only blank characters are truncated and *character-expression* is CHAR or VARCHAR, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004).
  - If *integer* is specified, truncation is performed. If only blank characters are truncated and *character-expression* is CHAR or VARCHAR, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004). When part of a multi-byte character is truncated, that partial character is replaced with the blank character. Do not rely on this behavior because it might change in a future release.

- If *integer* is not specified and *character-expression* is VARCHAR, truncation behavior is:
  - If only blank characters must be truncated, truncation is performed with no warning returned.
  - If non-blank characters must be truncated, an error is returned (SQLSTATE 22001).
- If *integer* is not specified and *character-expression* is CLOB, an error is returned (SQLSTATE 22001).

Graphic string to CHAR

graphic-expression
An expression that returns a value that is a built-in graphic string data type.

integer
An integer constant that specifies the length attribute for the resulting fixed-length character string. The value must be between 0 and the maximum length for the CHAR data type in the string units of the result.

If the second argument is not specified:

- If the *graphic-expression* is the empty string constant, the length attribute of the result is 0.
- If the string units of *graphic-expression* is CODEUNITS32, the length attribute of the result is the lower of the following values:
  - The maximum length for the CHAR data type in the string units of the result.
  - The length attribute of the first argument.
- Otherwise, the length attribute of the result is the lower of the following values:
  - The maximum length for the CHAR data type in the string units of the result.
  - 3 * length attribute of the first argument.

The result is a fixed-length character string that is converted from *graphic-expression*. The length of the result is the same as the length attribute of the result.

If the length of *graphic-expression* that is converted to a character string is:

- Less than the length attribute of the result, the result is padded with blanks up to the length attribute of the result.
- Greater than the length attribute of the result:
  - If the string unit of the result is CODEUNITS32, truncation is performed. If only blank characters are truncated and *graphic-expression* is GRAPHIC or VARGRAPHIC, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004).
  - If *integer* is specified and *graphic-expression* is a GRAPHIC or VARGRAPHIC, truncation is performed with no warning returned.
  - If *integer* is specified and graphic-expression is a DBCLOB, truncation is performed with a warning returned (SQLSTATE 01004).
  - If *integer* is not specified, an error is returned (SQLSTATE 22001).

Binary string to CHAR

binary-expression
An expression that returns a value that is a built-in binary string data type.

integer
An integer constant that specifies the length attribute for the resulting fixed-length character string.

The result is a fixed-length FOR BIT DATA character string, padded with blanks if necessary.

Datetime to CHAR

datetime-expression
An expression that is of one of the following data types:

DATE
The result is the character string representation of the date in the format specified by the second argument. The length of the result is 10. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

TIME
The result is the character string representation of the time in the format specified by the second argument. The length of the result is 8. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

TIMESTAMP
The result is the character string representation of the timestamp. If the data type of *datetime-expression* is TIMESTAMP(0), the length of the result is 19. If the data type of *datetime-expression* is TIMESTAMP(*n*), where *n* is between 1 and 12, the length of the result is 20+*n*. Otherwise, the length of the result is 26. The second argument is not applicable and must not be specified (SQLSTATE 42815).

The code page of the result is the code page of the section.

Boolean to CHAR

boolean-expression
An expression that returns a Boolean value (TRUE or FALSE). The result is either 'TRUE ' (note the blank after the E) or 'FALSE'.

Result

The CHAR function returns a fixed-length character string representation of:

- An integer number, if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number, if the first argument is a decimal number
- A double-precision floating-point number, if the first argument is a DOUBLE or REAL
- A decimal floating-point number, if the first argument is a DECFLOAT
- A character string, if the first argument is any type of character string
- A graphic string (Unicode databases only), if the first argument is any type of graphic string
- A datetime value, if the first argument is a DATE, TIME, or TIMESTAMP
- A Boolean value (TRUE or FALSE)

In a non-Unicode database, the string units of the result is OCTETS. Otherwise, the string units of the result are determined by the data type of the first argument.

- OCTETS, if the first argument is character string or a graphic string with string units of OCTETS, CODEUNITS16, or double bytes.
- CODEUNITS32, if the first argument is character string or a graphic string with string units of CODEUNITS32.
- Determined by the default string unit of the environment, if the first argument is not a character string or a graphic string.

In a Unicode database, when the output string is truncated part-way through a multiple-byte character:

- If the input was a character string, the partial character is replaced with one or more blanks
- If the input was a graphic string, the partial character is replaced by the empty string

Do not rely on either of these behaviors because they might change in a future release.

If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

Notes

- **Increasing portability of applications:** If the first argument is numeric, or if the first argument is a string and the length argument is specified, use the "CAST specification" on page 152 instead of this function to increase the portability of your applications.
- **Decimal to character and leading zeros**: In versions previous to version 9.7, the result for decimal input to this function includes leading zeros and a trailing decimal character. The database configuration parameter *dec_to_char_fmt* can be set to "V95" to have this function return the version 9.5 result for decimal input. The default value of the *dec_to_char_fmt* database configuration parameter for new databases is "NEW", which has this function return results which match the SQL standard casting rules and is consistent with results from the VARCHAR function.

Examples

- *Example 1*: Assume that the PRSTDATE column has an internal value equivalent to 1988-12-25. The following function returns the value '12/25/1988'.

```
CHAR(PRSTDATE, USA)
```

- *Example 2*: Assume that the STARTING column has an internal value equivalent to 17:12:30, and that the host variable HOUR_DUR (decimal(6,0)) is a time duration with a value of 050000 (that is, 5 hours). The following function returns the value '5:12 PM'.

```
CHAR(STARTING, USA)
```

  The following function returns the value '10:12 PM'.

```
CHAR(STARTING + :HOUR_DUR, USA)
```

- *Example 3*: Assume that the RECEIVED column (TIMESTAMP) has an internal value equivalent to the combination of the PRSTDATE and STARTING columns. The following function returns the value '1988-12-25-17.12.30.000000'.

```
CHAR(RECEIVED)
```

- *Example 4*: The LASTNAME column is defined as VARCHAR(15). The following function returns the values in this column as fixed-length character strings that are 10 bytes long. LASTNAME values that are more than 10 bytes long (excluding trailing blanks) are truncated and a warning is returned.

```
SELECT CHAR(LASTNAME,10) FROM EMPLOYEE
```

- *Example 5*: The EDLEVEL column is defined as SMALLINT. The following function returns the values in this column as fixed-length character strings. An EDLEVEL value of 18 is returned as the CHAR(6) value '18' followed by four blanks.

```
SELECT CHAR(EDLEVEL) FROM EMPLOYEE
```

- *Example 6*: The SALARY column is defined as DECIMAL with a precision of 9 and a scale of 2. The current value (18357.50) is to be displayed with a comma as the decimal character (18357,50). The following function returns the value '18357,50' followed by three blanks.

```
CHAR(SALARY, ',')
```

- *Example 7*: Values in the SALARY column are to be subtracted from 20000.25 and displayed with the default decimal character. The following function returns the value '-0001642.75' followed by three blanks.

```
CHAR(20000.25 - SALARY)
```

- *Example 8*: Assume that the host variable SEASONS_TICKETS is defined as INTEGER and has a value of 10000. The following function returns the value '10000.00'.

```
CHAR(DECIMAL(:SEASONS_TICKETS,7,2))
```

- *Example 9*: Assume that the host variable DOUBLE_NUM is defined as DOUBLE and has a value of -987.654321E-35. The following function returns the value '-9.87654321E-33' followed by nine trailing blanks because the result data type is CHAR(24).

```
CHAR(:DOUBLE_NUM)
```

- *Example 10*: The following statement returns a string of data type CHAR with the value 'TRUE '.

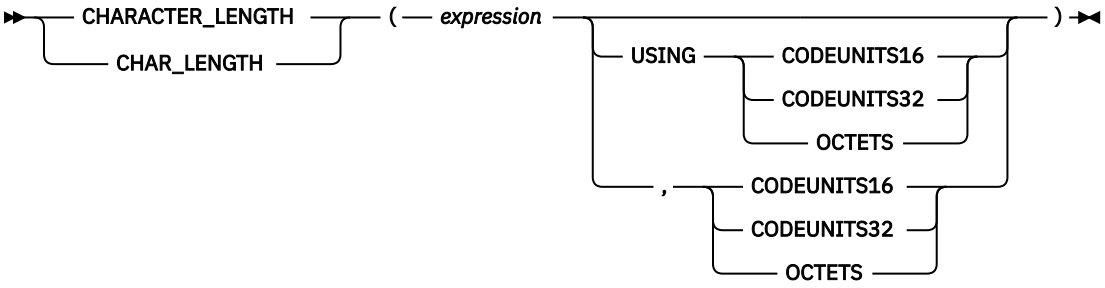
```
values CHAR(3=3)
```

- *Example 11*: The following statement returns a string of data type CHAR with the value 'FALSE'.

```
values CHAR(3>3)
```

CHARACTER_LENGTH

The CHARACTER_LENGTH function returns the length, in the specified string unit, of an expression.



The schema is SYSIBM.

expression
An expression that returns a built-in character, binary, or graphic string, or a Boolean value.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of the result:

- CODEUNITS16 specifies that the result is to be expressed in 16-bit UTF-16 code units.
- CODEUNITS32 specifies that the result is to be expressed in 32-bit UTF-32 code units.
- OCTETS specifies that the result is to be expressed in bytes.

If a string unit is specified as CODEUNITS16 or CODEUNITS32, and *expression* is a binary string or a FOR BIT DATA string, an error is returned (SQLSTATE 428GC).

If a string unit argument is not specified and *expression* is a character string that is not FOR BIT DATA or is a graphic string, the default is CODEUNITS32. Otherwise, the default is OCTETS.

For more information, see "String units in built-in functions" in "Character strings".

Result

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The length of character and graphic strings includes trailing blanks. The length of varying-length strings is the actual length and not the maximum length.

Examples

- Assume that NAME is a VARCHAR(128) column, encoded in Unicode UTF-8, that contains the value 'Jürgen'. The following two queries return the value 6:

```
SELECT CHARACTER_LENGTH(NAME, CODEUNITS32)
FROM T1 WHERE NAME = 'Jürgen'

SELECT CHARACTER_LENGTH(NAME, CODEUNITS16)
FROM T1 WHERE NAME = 'Jürgen'
```

  The following two queries return the value 7:

```
SELECT CHARACTER_LENGTH(NAME, OCTETS)
FROM T1 WHERE NAME = 'Jürgen'

SELECT LENGTH(NAME)
FROM T1 WHERE NAME = 'Jürgen'
```

- The following examples work with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown in different Unicode encoding forms in the following example:

	'&'	'N'	'~'	'A'	'B'
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'
UTF-32BE	X'0001D11E'	X'0000004E'	X'00000303'	X'00000041'	X'00000042'

  Assume that the variable UTF8_VAR contains the UTF-8 representation of the string.

```
SELECT CHARACTER_LENGTH(UTF8_VAR, CODEUNITS16),
  CHARACTER_LENGTH(UTF8_VAR, CODEUNITS32),
  CHARACTER_LENGTH(UTF8_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

  returns the values 6, 5, and 9, respectively.

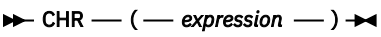
  Assume that the variable UTF16_VAR contains the UTF-16BE representation of the string.

```
SELECT CHARACTER_LENGTH(UTF16_VAR, CODEUNITS16),
   CHARACTER_LENGTH(UTF16_VAR, CODEUNITS32),
   CHARACTER_LENGTH(UTF16_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

  returns the values 6, 5, and 12, respectively.

CHR

Returns the character that has the ASCII code value specified by the argument.



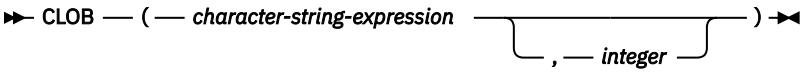
The schema is SYSFUN.

expression
An expression that returns a value of INTEGER or SMALLINT data type.

The result of the function is CHAR(1). The result can be null; if the argument is null, the result is the null value. If the argument value is between 1 and 255, the result is the character that has the ASCII code value corresponding to the argument. If the argument value is 0, the result is the blank character (X'20'). Otherwise the result is the same as CHR(255).

CLOB

The CLOB function returns a CLOB representation of a character string type.



The schema is SYSIBM.

In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string data type before the function is executed.

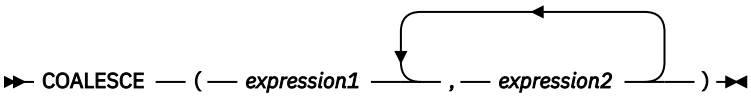
character-string-expression
An expression that returns a value that is a character string. The expression cannot be a character string defined as FOR BIT DATA (SQLSTATE 42846).

integer
An integer value specifying the length attribute of the resulting CLOB data type. If the *character-string-expression* string unit is OCTETS, the value must be between 0 and 2 147 483 647. If the *character-string-expression* string unit is CODEUNITS32, the value must be between 0 and 536 870 911. If a value for *integer* is not specified, the length of the result is the same as the length of the first argument.

The result of the function is a CLOB in the string units of *character-string-expression*. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

COALESCE

The COALESCE function returns the first non-null expression in a list of expressions.



The schema is SYSIBM.

expression1
An expression that returns a value of any built-in or user-defined data type.

expression2
An expression that returns a value of any built-in or user-defined data type and that is compatible with the data type of *expression1*. Which data types are compatible with each other is described in "Rules for result data types" on page 71.

The arguments are evaluated in the order in which they are specified, and the result of the function is the first argument that is not null. If all the arguments are null, the result is null.

Notes

- The COALESCE function cannot be used as a source function when creating a user-defined function. Because this function accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined data types.

Examples

- *Example 1*: When selecting all the values from all the rows in the DEPARTMENT table, if the department manager (MGRNO) is missing (that is, null), then return a value of 'ABSENT'.

```
SELECT DEPTNO, DEPTNAME, COALESCE(MGRNO, 'ABSENT'), ADMRDEPT
  FROM DEPARTMENT
```

- *Example 2*: When selecting the employee number (EMPNO) and salary (SALARY) from all the rows in the EMPLOYEE table, if the salary is missing (that is, null), then return a value of zero.

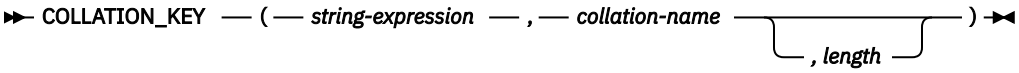
```
   SELECT EMPNO, COALESCE(SALARY, 0)
     FROM EMPLOYEE
```

- *Example 3*: In the following COALESCE statement, if the value of c1 is:
  - 5, the statement returns a value of 5
  - NULL, the statement returns a value of 10
  - 'AB', the statement returns an error, because the data types of the two expressions are incompatible

```
COALESCE(c1,10)
```

COLLATION_KEY

The COLLATION_KEY function returns a VARBINARY string that represents the collation key of the expression argument, in the specified collation.



The schema is SYSIBM.

The results of COLLATION_KEY for two strings can be binary compared to determine their order within the specified *collation-name*. For the comparison to be meaningful, the results that are used must be from the same *collation-name*.

string-expression
An expression for which the collation key is determined. The expression must return a value that is a built-in character string, graphic string, numeric, or datetime data type. Numeric and datetime data types are supported through implicit casting. The expression must not be a FOR BIT DATA subtype (SQLSTATE 429BM). If the expression is a CLOB, numeric, or datetime data type, the expression is cast to VARCHAR before the function is evaluated. If the expression is a DBCLOB, it is cast to VARGRAPHIC before the function is evaluated. If *string-expression* is not in UTF-16, this function converts the code page of *string-expression* to UTF-16. If the result of the code page conversion contains at least one substitution character, this function returns a collation key of the UTF-16 string with the substitution character or characters. In such cases, the warning flag SQLWARN8 in the SQLCA is set to 'W'.

collation-name
An expression that specifies the collation to use when the collation key is determined. The expression must return a value that is a CHAR or VARCHAR. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC. The expression must be a constant (SQLSTATE 428I9). The value of *collation-name* is not case-sensitive and must be one of the Unicode Collation Algorithm-based collations or language-aware collations for Unicode data (SQLSTATE 42704).

length
An expression that specifies the length attribute of the result in bytes. The expression must return a value that is a built-in numeric data type, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported through implicit casting. If the expression is not an INTEGER, it is cast to INTEGER before the function is evaluated. The value must be 1 - 32 672 (SQLSTATE 42815). The expression must be a constant (SQLSTATE 428I9).

If a value for *length* is not specified, the length of the result is determined as described in the following table:

Table 55. Determining the result length	
Data type of *string-expression*	**Result data type length**
CHAR(*n*) or VARCHAR(*n*)	Minimum of 12*n* bytes and 32 672 bytes
GRAPHIC(*n*) or VARGRAPHIC(*n*)	Minimum of 12*n* bytes and 32 672 bytes

Regardless of whether *length* is specified, if the length of the collation key is longer than the length of the result data type, an error is returned (SQLSTATE 42815). The actual result length of the collation key is approximately six times the length of *string-expression* after it is converted to UTF-16.

If *string-expression* is an empty string, the result is a valid collation key that can have a nonzero length.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

1. The following query orders employees by their surnames by using the language-aware collation for German in code page 923:

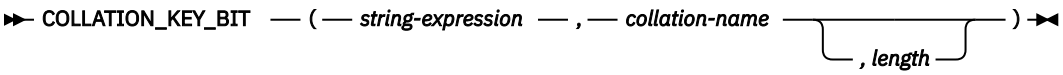
```
SELECT FIRSTNME, LASTNAME
   FROM EMPLOYEE
      ORDER BY COLLATION_KEY (LASTNAME, 'SYSTEM_923_DE')
```

2. The following query uses a culturally correct comparison to find the departments of employees in the province of Québec:

```
SELECT E.WORKDEPT
   FROM EMPLOYEE AS E INNER JOIN SALES AS S
   ON COLLATION_KEY(E.LASTNAME, 'CLDR181_LFR') =
      COLLATION_KEY(S.SALES_PERSON, 'CLDR181_LFR')
      WHERE S.REGION = 'Quebec'
```

COLLATION_KEY_BIT

The COLLATION_KEY_BIT function returns a VARCHAR FOR BIT DATA string that represents the collation key of the *string-expression* in the specified *collation-name*.



The schema is SYSIBM.

The results of COLLATION_KEY_BIT for two strings can be binary compared to determine their order within the specified *collation-name*. For the comparison to be meaningful, the results used must be from the same *collation-name*.

string-expression
An expression that returns a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC string for which the collation key should be determined. If *string-expression* is a CHAR or VARCHAR, the expression must not be FOR BIT DATA (SQLSTATE 429BM).

If *string-expression* is not in UTF-16, this function performs code page conversion of *string-expression* to UTF-16. If the result of the code page conversion contains at least one substitution character, this function returns a collation key of the UTF-16 string with the substitution character or characters and the warning flag SQLWARN8 in the SQLCA is set to 'W'.

collation-name
A character constant that specifies the collation to use when determining the collation key.

The value of *collation-name* is not case sensitive and must be one of the "Unicode Collation Algorithm-based collations" in *Globalization Guide* or "language-aware collations for Unicode data" in *Globalization Guide* (SQLSTATE 42616).

length
An expression that specifies the length attribute of the result in bytes. If specified, *length* must be an integer between 1 and 32 672 (SQLSTATE 42815).

If a value for *length* is not specified, the length of the result is determined as follows:

Table 56. Determining the result length

Data type of *string-expression*	**Result data type length**
CHAR(*n*) or VARCHAR(*n*)	Minimum of 12*n* bytes and 32 672 bytes
GRAPHIC(*n*) or VARGRAPHIC(*n*)	Minimum of 12*n* bytes and 32 672 bytes

Regardless of whether *length* is specified or not, if the length of the collation key is longer than the length of the result data type, an error is returned (SQLSTATE 42815). The actual result length of the collation key is approximately six times the length of *string-expression* after it is converted to UTF-16.

If *string-expression* is an empty string, the result is a valid collation key that can have a nonzero length.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: The following query orders employees by their family names by using the language-aware collation for German in code page 923:

```
SELECT FIRSTNME, LASTNAME
  FROM EMPLOYEE
  ORDER BY COLLATION_KEY_BIT (LASTNAME, 'SYSTEM_923_DE')
```

- *Example 2*: The following query uses a culturally correct comparison to find the departments of employees in the province of Québec:

```
SELECT E.WORKDEPT
 FROM EMPLOYEE AS E INNER JOIN SALES AS S
 ON COLLATION_KEY_BIT(E.LASTNAME, 'CLDR181_LFR') =
   COLLATION_KEY_BIT(S.SALES_PERSON, 'CLDR181_LFR')
 WHERE S.REGION = 'Quebec'
```

COMPARE_DECFLOAT

The COMPARE_DECFLOAT function returns a SMALLINT value that indicates whether the two arguments are equal or unordered, or whether one argument is greater than the other.

The schema is SYSIBM.

expression1
An expression that returns a value of any built-in numeric data type. If the argument is not DECFLOAT(34), it is logically converted to DECFLOAT(34) for processing.

expression2
An expression that returns a value of any built-in numeric data type. If the argument is not DECFLOAT(34), it is logically converted to DECFLOAT(34) for processing.

The value of *expression1* is compared with the value of *expression2*, and the result is returned according to the following rules:

- If both arguments are finite, the comparison is algebraic and follows the procedure for decimal floating-point subtraction. If the difference is exactly zero with either sign, the arguments are equal. If a nonzero difference is positive, the first argument is greater than the second argument. If a nonzero difference is negative, the first argument is less than the second.
- Positive zero and negative zero compare as equal.

- Positive infinity compares equal to positive infinity.
- Positive infinity compares greater than any finite number.
- Negative infinity compares equal to negative infinity.
- Negative infinity compares less than any finite number.
- Numeric comparison is exact. The result is determined for finite operands as if range and precision were unlimited. No overflow or underflow condition can occur.
- If either argument is NaN or sNaN (positive or negative), the result is unordered.

The result value is as follows:

- 0 if the arguments are exactly equal
- 1 if *expression1* is less than *expression2*
- 2 if *expression1* is greater than *expression2*
- 3 if the arguments are unordered

The result of the function is a SMALLINT value. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

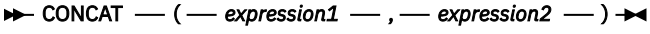
Examples

The following examples show the values that are returned by the COMPARE_DECFLOAT function, given a variety of input decimal floating-point values:

```
COMPARE_DECFLOAT(DECFLOAT(2.17), DECFLOAT(2.17)) = 0
COMPARE_DECFLOAT(DECFLOAT(2.17), DECFLOAT(2.170)) = 2
COMPARE_DECFLOAT(DECFLOAT(2.170), DECFLOAT(2.17)) = 1
COMPARE_DECFLOAT(DECFLOAT(2.17), DECFLOAT(0.0)) = 2
COMPARE_DECFLOAT(INFINITY,INFINITY) = 0
COMPARE_DECFLOAT(INFINITY,-INFINITY) = 2
COMPARE_DECFLOAT(DECFLOAT(-2),INFINITY) = 1
COMPARE_DECFLOAT(NAN,NAN) = 3
COMPARE_DECFLOAT(DECFLOAT(-0.1),SNAN) = 3
```

CONCAT

The CONCAT function combines two arguments to form a string expression.



The schema is SYSIBM.

expression1* or *expression2
An expression that returns a value of one of the following data types:

- Character, binary, or graphic string
- Numeric value (this is implicitly cast to VARCHAR)
- Datetime value (this is implicitly cast to VARCHAR)
- Boolean value (this is implicitly cast to VARCHAR)

The following restrictions apply:

- A binary string can be concatenated only with another binary string or with a character string that is defined as FOR BIT DATA.
- A character string and a graphic string can be concatenated only in a Unicode database. The character string is converted to a graphic string before concatenation. The character string cannot be defined as FOR BIT DATA, because such a character string cannot be cast to a graphic data string.
- If an argument is defined with CODEUNITS32, the other argument cannot be defined as FOR BIT DATA.

Result

The result of the function is a string that consists of the first argument followed by the second argument. The data type and the length of the result is determined by the data types and lengths of the arguments, after any applicable casting is done. For more information, refer to the "Data Type and Length of Concatenated Operands" table in “Expressions” on page 132.

If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Notes

- No check is made for improperly formed mixed data when doing concatenation.
- The CONCAT function is identical to the CONCAT operator. For more information, see “Expressions” on page 132.

Example

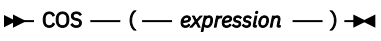
Concatenate the column FIRSTNME with the column LASTNAME.

```
SELECT CONCAT(FIRSTNME, LASTNAME)
FROM EMPLOYEE
WHERE EMPNO = '000010'
```

Returns the value "CHRISTINEHAAS".

COS

Returns the cosine of the argument, where the argument is an angle expressed in radians.



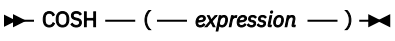
The schema is SYSIBM. (The SYSFUN version of the COS function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

COSH

Returns the hyperbolic cosine of the argument, where the argument is an angle expressed in radians.



The schema is SYSIBM.

expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

COT

Returns the cotangent of the argument, where the argument is an angle expressed in radians.

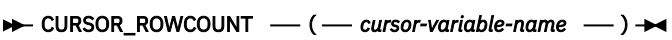
The schema is SYSIBM. (The SYSFUN version of the COT function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

CURSOR_ROWCOUNT

The CURSOR_ROWCOUNT function returns the cumulative count of all rows fetched by the specified cursor since the cursor was opened.



The schema is SYSIBM.

cursor-variable-name
The name of a SQL variable or SQL parameter of a cursor type. The underlying cursor of the *cursor-variable-name* must be open (SQLSTATE 24501).

The result is 0 if no FETCH action on the underlying cursor of the cursor-variable-name was performed before the evaluation of the function.

This function can only be used within a compound SQL (compiled) statement.

The data type of the result is BIGINT. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

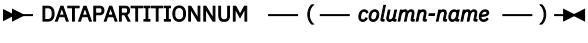
Example

The following example shows how to use the function to retrieve the count of rows associated with the cursor *curEmp* and assign it to a variable named *rows_fetched*:

```
SET rows_fetched = CURSOR_ROWCOUNT(curEmp);
```

DATAPARTITIONNUM

The DATAPARTITIONNUM function returns the sequence number (SYSDATAPARTITIONS.SEQNO) of the data partition in which the row resides.



The schema is SYSIBM.

column-name
The qualified or unqualified name of any column in the table. Because row-level information is returned, the result is the same regardless of which column is specified. The column can have any data type.

If the column is a column of a view, the expression for the column in the view must reference a column of the underlying base table, and the view must be deletable. A nested or common table expression follows the same rules as a view.

Result

The data type of the result is INTEGER and is never null.

Data partitions are sorted by range, and sequence numbers start at 0. For example, the DATAPARTITIONNUM function returns 0 for a row that resides in the data partition with the lowest range.

Notes

- This function cannot be used as a source function when creating a user-defined function. Because the function accepts any data type as an argument, it is not necessary to create additional signatures to support user-defined distinct types.
- The DATAPARTITIONNUM function cannot be used within check constraints or in the definition of generated columns (SQLSTATE 42881). The DATAPARTITIONNUM function cannot be used in a materialized query table (MQT) definition (SQLSTATE 428EC).
- The DATAPARTITIONNUM function cannot be used as part of an expression-based key in a CREATE INDEX statement.

Examples

- *Example 1:* Retrieve the sequence number of the data partition in which the row for EMPLOYEE.EMPNO resides.

```
SELECT DATAPARTITIONNUM (EMPNO)
FROM EMPLOYEE
```

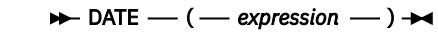
- *Example 2:* To convert a sequence number that is returned by DATAPARTITIONNUM (for example, 0) to a data partition name that can be used in other SQL statements (such as ALTER TABLE...DETACH PARTITION), you can query the SYSCAT.DATAPARTITIONS catalog view. Include the SEQNO obtained from DATAPARTITIONNUM in the WHERE clause, as shown in the following example.

```
SELECT DATAPARTITIONNAME
FROM SYSCAT.DATAPARTITIONS
WHERE TABNAME = 'EMPLOYEE' AND SEQNO = 0
```

  results in the value 'PART0'.

DATE

The DATE function returns a date from a value.



The schema is SYSIBM.

The special behavior of DATE with the Db2 compatibility features for Oracle applications is described in DATE data type based on TIMESTAMP(0).

expression
An expression that returns a value of one of the following built-in data types: DATE, TIMESTAMP, numeric, or character string that is not a CLOB.

A value with a numeric data type must be a positive number with an integral value less than or equal to 3 652 059.

A character string must be a valid string representation of a date or timestamp or a string of length 7. If the value is a string of length 7, it must represent a valid date in the form *yyyynnn* , where *yyyy* are digits denoting a year, and *nnn* are digits between 001 and 366, denoting a day of that year.

In a Unicode database, if an expression returns a value of a graphic string data type, the value is first converted to a character string before the function is executed.

The result of the function is a DATE. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a DATE, TIMESTAMP, or valid string representation of a date or timestamp:
  - The result is the date part of the value.
- If the argument is a number:
  - The result is the date that is *n*-1 days after January 1, 0001, where *n* is the integral part of the number.
- If the argument is a string with a length of 7:
  - The result is the date represented by the string.

Examples

Assume that the column RECEIVED (whose data type is TIMESTAMP) has an internal value equivalent to '1988-12-25-17.12.30.000000'.

- *Example 1*: This example results in an internal representation of '1988-12-25'.

```
DATE(RECEIVED)
```

- *Example 2*: This example results in an internal representation of '1988-12-25'.

```
DATE('1988-12-25')
```

- *Example 3*: This example results in an internal representation of '1988-12-25'.

```
DATE('25.12.1988')
```

- *Example 4*: This example results in an internal representation of '0001-02-04'.

```
DATE(35)
```

DATETIME

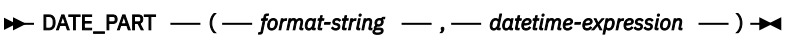
The DATETIME function returns a timestamp from a value or a pair of values.

The schema is SYSIBM.

The DATETIME scalar function is a synonym for the TIMESTAMP scalar function.

DATE_PART

The DATE_PART function returns a portion of a datetime based on its arguments. It extracts the subfield that is specified from the date, time, timestamp, and duration values.



The schema is SYSIBM.

format-string

An expression that represents which unit of datetime is to be extracted from the *date-expression*. The expression that returns a built-in character string data type with an actual length that is not greater than 255 bytes. The format element in *format-string* specifies how the datetime-expression should be truncated. Leading and trailing blanks are removed from the string, and the resulting substring must

be a valid format element for the type of *datetime-expression* (SQLSTATE 22007). Table 57 on page 317 lists the allowable values for *format-string*.

datetime-expression
An expression that specifies the datetime value from which the unit specified by *format-string* is extracted. The expression must return a value that is a DATE, TIME, TIMESTAMP, date duration, time duration, or timestamp duration.

Table 57. Allowable values for format-string

format-string	**Description**
EPOCH	
MILLENNIUM/ MILLENNIUMS	
CENTURY/CENTURIES	
DECADE/DECADES	
YEAR/YEARS	
QUARTER	
MONTH/MONTHS	
WEEK	
DAY/DAYS	
DOW	
DOY	
HOUR/HOURS	
MINUTE/MINUTES	
SECOND/SECONDS	
MILLISECOND/ MILLISECONDS	
MICROSECOND/ MICROSECONDS	

The format-string values are case insensitive.

The data type of the result of this function is BIGINT. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

- *Example 1*: Extracting the day part from a date value.

```
values DATE_PART('DAY', DATE('2007-02-18'));
    Result: 18
```

- *Example 2*: Extracting the year part from a date duration.

```
values DATE_PART('YEAR', cast(20130710 as decimal(8,0)));
    Result: 2013
```

- *Example 3*: Extracting the hour part from a time duration.

```
values DATE_PART('HOUR', cast(075559 as decimal(6,0)));
    Result: 7
```

DATE_TRUNC

THE DATE_TRUNC function truncates a date, time, or timestamp value to the specified time unit.

The schema is SYSIBM.

format-string

An expression that returns a character string that does not exceed 255 bytes and that, after all leading and trailing blanks are removed, is one of the character strings listed in Table 58 on page 318. The resulting substring must be a valid format element for the type of the specified datetime expression (SQLSTATE 22007). For example, a DATE value cannot be truncated to its first hour, minute, or second, and a TIME value cannot be truncated to the first day of its year.

datetime-expression

An expression that returns a DATE, TIME, or TIMESTAMP value, or a DECIMAL value representing a date, time, or timestamp duration. String representations of these data types must be explicitly cast to a DATE, TIME, TIMESTAMP, or DECIMAL value. For more information about durations, see "Datetime operations and durations" on page 145.

Table 58. Allowed values for a format string

If the format string is...	the datetime expression is truncated to the...	For example...	is truncated to...
'MILLENNIUM' or 'MILLENNIUMS'	First day of its millennium	1999-02-14	1000-01-01
'CENTURY' or 'CENTURIES'	First day of its century	1999-02-14	1900-01-01
'DECADE' or 'DECADES'	First day of its decade	1999-02-14	1990-01-01
'YEAR' or 'YEARS'	First day of its year	1999-02-14	1999-01-01
'QUARTER'	First day of its quarter	2017-05-14 20:38:40.24	2017-04-01 00:00:00
'MONTH' or 'MONTHS'	First day of its month	2017-05-14 20:38:40.24	2017-05-01 00:00:00
'WEEK'	First day of its week	2017-05-14 20:38:40.24	2017-05-08 00:00:00
'DAY' or 'DAYS'	Beginning of its day	2017-05-14 20:38:40.24	2017-05-14 00:00:00
'HOUR' or 'HOURS'	Beginning of its hour	2017-05-14 20:38:40.24	2017-05-14 20:00:00
'MINUTE' or 'MINUTES'	Beginning of its minute	20:38:40.24576985	20:38:00

Table 58. Allowed values for a format string (continued)			
If the format string is...	**the datetime expression is truncated to the...**	**For example...**	**is truncated to...**
'SECOND' or 'SECONDS'	Beginning of its second	20:38:40.24576985	20:38:40
'MILLISECOND' or 'MILLISECONDS'	Beginning of its millisecond	20:38:40.24576985	20:38:40.245
'MICROSECOND' or 'MICROSECOND'	Beginning of its microsecond	20:38:40.24576985	20:38:40.245769

The format-string values are case insensitive.

Result

If the specified datetime expression is:

- A DATE or TIMESTAMP value, the result is a TIMESTAMP value
- A TIME value, the result is a TIME value
- A DECIMAL value representing a date, time, or timestamp duration, the result is a DECIMAL value of the same type

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

- *Example 1*: Truncate a DATE value to the beginning of the month.

```
values date_trunc('MONTH', DATE('2007-02-18'))
      Result: 2007-02-01 00:00:00
```

- *Example 2*: Truncate a TIMESTAMP value to the beginning of the hour.

```
values date_trunc('HOUR', TIMESTAMP('2017-02-14 20:38:40.24'));
     Result: 2017-02-14 20:00:00
```

- *Example 3*: Truncate a TIME value to the beginning of the minute.

```
 values date_trunc('MINUTE', TIME('20:38:40'));
     Result: 20:38:00
```

- *Example 4*: A date duration has the data type DECIMAL(8,0). For example, the value 00200203 represents a duration of 20 years, 2 months, and 3 days. Truncate a date duration to the beginning of the month.

```
db2 "values date_trunc('MONTH', cast ('00200203' AS DECIMAL(8,0)))"

1
----------
   200200.

  1 record(s) selected.
```

- *Example 5*: A time duration has the data type DECIMAL(6,0). For example, the value 102930 represents a duration of 10 hours, 29 minutes, and 30 seconds. Truncate a time duration to the beginning of the minute.

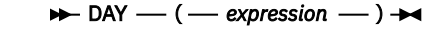
```
db2 "values date_trunc('MINUTE',  cast ('102930' AS DECIMAL(6,0)))"

1
--------
 102900.

  1 record(s) selected.
```

- *Example 6*: A timestamp duration has the data type DECIMAL(14+s,*s*), where *s* is the timestamp precision. For example, the DECIMAL(20,6) value 00070005032040.000301 represents a duration of 7 years, 0 months, 5 days, 3 hours, 20 minutes, and 40.000301 seconds. Truncate a timestamp duration to the beginning of the hour.

```
db2 "values date_trunc('HOUR', cast ('00070005032040.000301' AS DECIMAL(15,1)))"

1
-----------------
    70005030000.0

  1 record(s) selected.
```

DAY

The DAY function returns the day part of a value.



The schema is SYSIBM.

expression
An expression that returns a value of one of the following built-in data types: DATE, TIMESTAMP, numeric, or character string that is not a CLOB.

If the value is a number, it must be a date duration or timestamp duration (SQLSTATE 42815).

If the value is a character string, it must be a valid string representation of a date or timestamp. In a Unicode database, if the value is a graphic string (except DBCLOB), it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a DATE, TIMESTAMP, or valid string representation of a date or timestamp:
  - The result is the day part of the value, which is an integer between 1 and 31.
- If the argument is a date duration or timestamp duration:
  - The result is the day part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Examples

- *Example 1*: Using the PROJECT table, set the host variable END_DAY (smallint) to the day that the WELD LINE PLANNING project (PROJNAME) is scheduled to stop (PRENDATE).

```
SELECT DAY(PRENDATE)
  INTO :END_DAY
  FROM PROJECT
  WHERE PROJNAME = 'WELD LINE PLANNING'
```

Results in END_DAY being set to 15 when using the sample table.

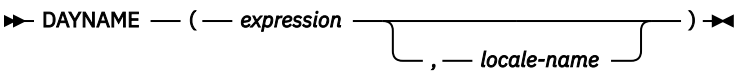
- *Example 2*: Assume that the column DATE1 (whose data type is DATE) has an internal value equivalent to 2000-03-15 and the column DATE2 (whose data type is DATE) has an internal value equivalent to 1999-12-31.

```
DAY(DATE1 - DATE2)
```

Results in the value 15.

DAYNAME

The DAYNAME function returns a character string containing the name of the day (for example, Friday) for the day portion of the input value.



The schema is SYSIBM. The SYSFUN version of the DAYNAME function continues to be available

The character string returned is based on *locale-name* or the value of the special register CURRENT LOCALE LC_TIME.

expression
An expression that returns a value of one of the following built-in data types: a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

locale-name
A character constant that specifies the locale used to determine the language of the result. The value of *locale-name* is not case-sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming, see "Locale names for SQL and XQuery". If *locale-name* is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result is a varying-length character string. The length attribute is 100. If the resulting string exceeds the length attribute of the result, the result will be truncated. If the *expression* argument can be null, the result can be null; if the *expression* argument is null, the result is the null value. The code page of the result is the code page of the section. The string units of the result is determined by the string units of the environment

Notes

- **Julian and Gregorian calendar**: The transition from the Julian calendar to the Gregorian calendar on 15 October 1582 is taken into account by this function. However, the SYSFUN version of the DAYNAME function assumes the Gregorian calendar for all calculations.
- **Determinism**: DAYNAME is a deterministic function. However, when *locale-name* is not explicitly specified, the invocation of the function depends on the value of the special register CURRENT LOCALE LC_TIME. This invocation that depends on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621, 428EC, or 429BX).

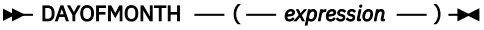
Example

Assume that the variable TMSTAMP is defined as TIMESTAMP and has the following value: 2007-03-09-14.07.38.123456. The following examples show several invocations of the function and the resulting string values. The result type in each case is VARCHAR(100).

```
Function invocation                          Result
--------------------------                   ----------
DAYNAME (TMSTAMP, 'CLDR181_en_US')           Friday
DAYNAME (TSMTAMP, 'CLDR181_de_DE')           Freitag
DAYNAME (TMSTAMP, 'CLDR181_fr_FR')           vendredi
```

DAYOFMONTH

The DAYOFMONTH function returns an integer between 1 and 31 that represents the day of the month.



The schema is SYSIBM.

expression
An expression that specifies the datetime value from which the day of the month is determined. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

The result of the function is an INTEGER. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

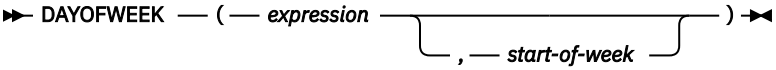
Set the host variable END_DAY to the day that the WELD LINE PLANNING project is scheduled to stop, by querying the PRENDATE column of the PROJECT table.

```
SELECT DAYOFMONTH(PRENDATE) INTO :END_DAY
   FROM PROJECT WHERE PROJNAME = 'WELD LINE PLANNING'
```

The host variable END_DAY is set to 15.

DAYOFWEEK

The DAYOFWEEK function returns the day of the week in the first argument as an integer value. The integer value is in the range 1-7, where 1 represents the first day of the week, as specified in the second argument.



The schema is SYSIBM. The SYSFUN version of the DAYOFWEEK function continues to be available.

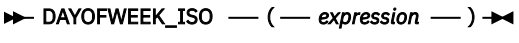
expression
An expression that specifies the datetime value from which the day of the week is determined. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. If *expression* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string representation of a date or timestamp.

start-of-week
An expression that specifies the starting day of the week. The expression must return a value that is a built-in numeric, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. The value must be in the range 1 - 7 (SQLSTATE 42815), where 1 represents Sunday. If not specified, the default value is 1.

The result of the function is INTEGER. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

DAYOFWEEK_ISO

Returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Monday.



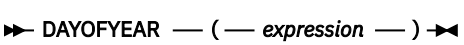
The schema is SYSFUN.

expression
An expression that returns a value of one of the following built-in data types: a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

DAYOFYEAR

Returns the day of the year in the argument as an integer value in the range 1-366.



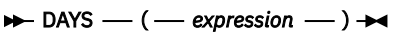
The schema is SYSFUN.

expression
An expression that returns a value of one of the following built-in data types: a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

DAYS

The DAYS function returns an integer representation of a date.



The schema is SYSIBM.

expression
An expression that returns a value of one of the following built-in data types: a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result is 1 more than the number of days from January 1, 0001 to *D*, where *D* is the date that would occur if the DATE function were applied to the argument.

Examples

- *Example 1*: Using the PROJECT table, set the host variable EDUCATION_DAYS (int) to the number of elapsed days (PRENDATE - PRSTDATE) estimated for the project (PROJNO) 'IF2000'.

```
SELECT DAYS(PRENDATE) - DAYS(PRSTDATE)
  INTO :EDUCATION_DAYS
```

```
    FROM PROJECT
    WHERE PROJNO = 'IF2000'
```

Results in EDUCATION_DAYS being set to 396.

- *Example 2*: Using the PROJECT table, set the host variable TOTAL_DAYS (int) to the sum of elapsed days (PRENDATE - PRSTDATE) estimated for all projects in department (DEPTNO) 'E21'.

```
SELECT SUM(DAYS(PRENDATE) - DAYS(PRSTDATE))
  INTO :TOTAL_DAYS
  FROM PROJECT
  WHERE DEPTNO = 'E21'
```

Results in TOTAL_DAYS being set to 1584 when using the sample table.

DAYS_BETWEEN

The DAYS_BETWEEN function returns the number of full days between the specified arguments.

The schema is SYSIBM.

expression1
An expression that specifies the first datetime value to compute the number of full days between two datetime values. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression1* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

expression2
An expression that specifies the second datetime value to compute the number of full days between two datetime values. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or a VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression2* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

If there is less than a full day between *expression1* and *expression2*, the result is zero. If *expression1* is later than *expression2*, the result is positive. If *expression1* is earlier than *expression2*, the result is negative. In NPS compatibility mode, this function always returns a positive number. If *expression1* or *expression2* contains time information, this information is also used to determine the number of full days. If *expression1* or *expression2* does not contain time information, a time of midnight (00.00.00) is used for the argument that is missing time information.

The result of the function is an INTEGER. If either argument can be null, the result can be null. If either argument is null, the result is the null value.

Examples

1. Set the host variable NUM_DAYS with the number of full days between 2012-03-03 and 2012-02-28.

```
SET :NUM_DAYS = DAYS_BETWEEN(DATE '2012-03-03',
   DATE '2012-02-28')
```

The host variable NUM_DAYS is set to 4 because an additional day is incurred for February 29, 2012.

2. Set the host variable NUM_DAYS with the number of full days between 2013-09-11-23.59.59 and 2013-09-01-00.00.00.

```
SET :NUM_DAYS = DAYS_BETWEEN(TIMESTAMP '2013-09-11-23.59.59',
   TIMESTAMP '2013-09-01-00.00.00')
```

The host variable NUM_DAYS is set to 10 because there is 1 second less than a full 11 days between the arguments. It is positive because the first argument is later than the second argument.

3. Set the host variable NUM_DAYS with the number of full days between 2013-09-01-00.00.00 and 2013-09-11-23.59.59.

```
SET :NUM_DAYS = DAYS_BETWEEN(TIMESTAMP '2013-09-01-00.00.00',
   TIMESTAMP '2013-09-11-23.59.59')
```

The host variable NUM_DAYS is set to -10 because there is 1 second less than a full 11 days between the arguments. It is negative because the first argument is earlier than the second argument.

DAYS_TO_END_OF_MONTH

The DAYS_TO_END_OF_MONTH function returns the number of days to the end of the month.

The schema is SYSIBM.

expression
An expression that specifies the datetime value for which the number of days to the end of the month is computed. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

The result of the function is INTEGER. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

1. Set the host variable NUMDAYS as the number of days to the end of the month for the date February 20, 2012.

```
SET :NUMDAYS = DAYS_TO_END_OF_MONTH(DATE '2012-02-20')
```

The host variable NUMDAYS is set with the value 9.

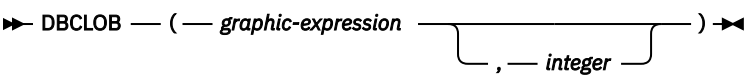
2. The DAYS_TO_END_OF_MONTH function and datetime arithmetic with DAYS and LAST_DAY functions can be used to achieve the same results. The following examples demonstrate this.

```
SET :NUMDAYS = DAYS(LAST_DAY(date '2013-02-20')) - DAYS(date '2013-02-20')
SET :NUMDAYS = DAYS_TO_END_OF_MONTH(DATE '2013-02-20')
```

In both cases, the host variable NUMDAYS is set with the value 8.

DBCLOB

The DBCLOB function returns a DBCLOB representation of a graphic string type.



The schema is SYSIBM.

graphic-expression
An expression that returns a value that is a graphic string.

integer
An integer value specifying the length attribute of the resulting DBCLOB data type. The value must be between 0 and 1 073 741 823 if the *graphic-expression* string unit is double bytes or CODEUNITS16,

or between 0 and 536 870 911 if *graphic-expression* string unit is CODEUNITS32. If *integer* is not specified, the length of the result is the same as the length of the first argument.

In a Unicode database, if a supplied argument is a character string, it is first converted to a graphic string before the function is executed. When the output string is truncated, such that the last character is a high surrogate, that surrogate is either:

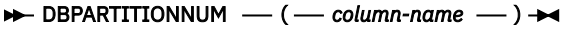
- Left as is, if the supplied argument is a character string
- Converted to the blank character (X'0020'), if the supplied argument is a graphic string

Do not rely on these behaviors, because they might change in a future release.

The result of the function is a DBCLOB in the string units of *graphic-expression*. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

DBPARTITIONNUM

The DBPARTITIONNUM function returns the database partition number for a row. For example, if used in a SELECT clause, it returns the database partition number for each row in the result set.



The schema is SYSIBM.

column-name
The qualified or unqualified name of any column in the table. Because row-level information is returned, the result is the same regardless of which column is specified. The column can have any data type.

If the column is a column of a view, the expression for the column in the view must reference a column of the underlying base table, and the view must be deletable. A nested or common table expression follows the same rules as a view.

The specific row (and table) for which the database partition number is returned by the DBPARTITIONNUM function is determined from the context of the SQL statement that uses the function.

The database partition number returned on transition variables and tables is derived from the current transition values of the distribution key columns. For example, in a before insert trigger, the function returns the projected database partition number, given the current values of the new transition variables. However, the values of the distribution key columns might be modified by a subsequent before insert trigger. Thus, the final database partition number of the row when it is inserted into the database might differ from the projected value.

Result

The data type of the result is INTEGER and is never null. If there is no `db2nodes.cfg` file, the result is 0.

Notes

- The DBPARTITIONNUM function cannot be used on replicated tables, within check constraints, or in the definition of generated columns (SQLSTATE 42881).
- The DBPARTITIONNUM function cannot be used as a source function when creating a user-defined function. Because it accepts any data type as an argument, it is not necessary to create additional signatures to support user-defined distinct types.
- The DBPARTITIONNUM function cannot be used as part of an expression-based key in a CREATE INDEX statement.
- ***Syntax alternatives:*** For compatibility with previous versions of Db2 products, the function name NODENUMBER is a synonym for DBPARTITIONNUM.

Examples

- *Example 1*: Count the number of instances in which the row for a given employee in the EMPLOYEE table is on a different database partition than the description of the employee's department in the DEPARTMENT table.

```
SELECT COUNT(*) FROM DEPARTMENT D, EMPLOYEE E
  WHERE D.DEPTNO=E.WORKDEPT
  AND DBPARTITIONNUM(E.LASTNAME) <> DBPARTITIONNUM(D.DEPTNO)
```

- *Example 2*: Join the EMPLOYEE and DEPARTMENT tables so that the rows of the two tables are on the same database partition.

```
SELECT * FROM DEPARTMENT D, EMPLOYEE E
  WHERE DBPARTITIONNUM(E.LASTNAME) = DBPARTITIONNUM(D.DEPTNO)
```

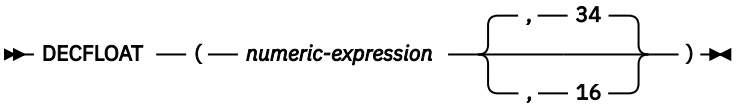
- *Example 3*: Using a before trigger on the EMPLOYEE table, log the employee number and the projected database partition number of any new row in the EMPLOYEE table in a table named EMPINSERTLOG1.

```
CREATE TRIGGER EMPINSLOGTRIG1
BEFORE INSERT ON EMPLOYEE
REFERENCING NEW AW NEWTABLE
FOR EACH ROW
INSERT INTO EMPINSERTLOG1
VALUES(NEWTABLE.EMPNO, DBPARTITIONNUM
(NEWTABLE.EMPNO))
```

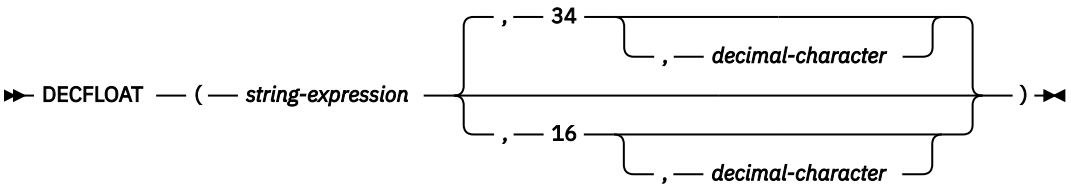
DECFLOAT

The DECFLOAT function returns a decimal floating-point representation of a value of a different data type.

Numeric to decimal floating-point



Character to decimal floating-point



Boolean to decimal floating-point

The schema is SYSIBM.

numeric-expression
An expression that returns a value of any built-in numeric data type.

string-expression
An expression that returns a value that is a character-string or Unicode graphic-string representation of a number with a length not greater than the maximum length of a character constant. The data type of string-expression must not be CLOB or DBCLOB (SQLSTATE 42884). Leading and trailing blanks are

removed from the string. The resulting substring is folded to uppercase and must conform to the rules for forming an integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018) and not be greater than 42 bytes (SQLSTATE 42820).

boolean-expression
An expression that returns a Boolean value (TRUE or FALSE). The result is either 1 (for TRUE) or 0 (for FALSE).

34 or 16
Specifies the number of digits of precision for the result. The default is 34.

decimal-character
Specifies the single-byte character constant used to delimit the decimal digits in *character-expression* from the whole part of the number. The character cannot be a digit, plus (+), minus (-), or blank, and it can appear at most once in *character-expression*.

Result

The result is the same number that would result from CAST(*string-expression* AS DECFLOAT(*n*)) or CAST(*numeric-expression* AS DECFLOAT(*n*)). Leading and trailing blanks are removed from the string.

The result of the function is a decimal floating-point number with the implicitly or explicitly specified number of digits of precision. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

If necessary, the source is rounded to the precision of the target. The CURRENT DECFLOAT ROUNDING MODE special register determines the rounding mode.

Notes

- **Increasing portability of applications:** If the first argument is numeric, or if the first argument is a string and the length argument is specified, use the "CAST specification" on page 152 instead of this function to increase the portability of your applications.
- All numeric values are interpreted as integer, decimal, or floating-point constants and then cast to decimal floating-point. The use of a floating-point constant can result in round-off errors and is therefore strongly discouraged. Use the string to decimal floating-point version of the DECFLOAT function instead.

Examples

- *Example 1*: Use the DECFLOAT function in order to force a DECFLOAT data type to be returned in a select-list for the EDLEVEL column (data type = SMALLINT) in the EMPLOYEE table. The EMPNO column should also appear in the select list.

```
SELECT EMPNO, DECFLOAT(EDLEVEL,16)
FROM EMPLOYEE
```

- *Example 2*: The following statement returns the value 1 of data type DECFLOAT.

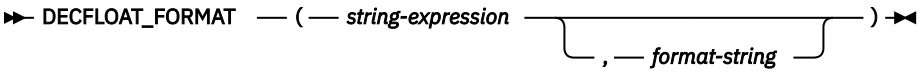
```
   values DECFLOAT(TRUE)
```

- *Example 3*: The following statement returns the value 0 of data type DECFLOAT.

```
   values DECFLOAT(3>3)
```

DECFLOAT_FORMAT

The DECFLOAT_FORMAT function returns a DECFLOAT(34) value that is based on the interpretation of the input string using the specified format.



The schema is SYSIBM.

string-expression
An expression that returns a value that is a CHAR and VARCHAR data type. In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function.Leading and trailing blanks are removed from the string. If *format-string* is not specified, the resulting substring must conform to the rules for forming an SQL integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018) and not be greater than 42 bytes (SQLSTATE 42820); otherwise, the resulting substring must contain the components of a number that correspond to the format specified by *format-string* (SQLSTATE 22018).

format-string
An expression that returns a value that is a built-in character string data type (except CLOB). In a Unicode database, if a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before evaluating the function. The actual length must not be greater than 255 bytes (SQLSTATE 22018). The value is a template for how *string-expression* is to be interpreted for conversion to a DECFLOAT value. Format elements specified as a prefix can be used only at the beginning of the template. Format elements specified as a suffix can be used only at the end of the template. The format elements are case sensitive. The template must not contain more than one of the MI, S, or PR format elements (SQLSTATE 22018).

Table 59. Format elements for the DECFLOAT_FORMAT function

Format element	**Description**
0 or 9	Each 0 or 9 represents a digit.
MI	Suffix: If *string-expression* is to represent a negative number, a trailing minus sign (-) is expected. If *string-expression* is to represent a positive number, a trailing blank can be specified.
S	Prefix: If *string-expression* is to represent a negative number, a leading minus sign (-) is expected. If *string-expression* is to represent a positive number, a leading plus sign (+) or leading blank can be specified.
PR	Suffix: If *string-expression* is to represent a negative number, a leading less than character (<) and a trailing greater than character (>) are expected. If *string-expression* is to represent a positive number, a leading space and a trailing space can be specified.
$	Prefix: A leading dollar sign ($) must be specified.
,	Specifies the expected location of a comma. This comma is used as a group separator.
.	Specifies the expected location of the period. This period is used as a decimal point.

If *format-string* is not specified, *string-expression* must conform to the rules for forming an SQL integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018) and have a length not greater than 42 bytes (SQLSTATE 42820).

The result is a DECFLOAT(34). If the first or second argument can be null, the result can be null; if the first or second argument is null, the result is the null value.

Notes

- Syntax alternatives: TO_NUMBER is a synonym for DECFLOAT_FORMAT.

Examples

- *Example 1*: The following example returns 123.45

```
DECFLOAT_FORMAT( '123.45' )
```

- *Example 2*: The following example returns -123456.78

```
DECFLOAT_FORMAT( '-123456.78' )
```

- *Example 3*: The following example returns 123456.78

```
DECFLOAT_FORMAT( '+123456.78' )
```

- *Example 4*: The following example returns 12300

```
DECFLOAT_FORMAT( '1.23E4' )
```

- *Example 5*: The following example returns 123.40

```
DECFLOAT_FORMAT( '123.4', '9999.99' )
```

- *Example 6*: The following example returns 1234

```
DECFLOAT_FORMAT( '001,234', '000,000' )
```

- *Example 7*: The following example returns 1234

```
DECFLOAT_FORMAT( '1234 ', '9999MI' )
```

- *Example 8*: The following example returns -1234

```
DECFLOAT_FORMAT( '1234-', '9999MI' )
```

- *Example 9*: The following example returns 1234

```
DECFLOAT_FORMAT( '+1234', 'S9999' )
```

- *Example 10*: The following example returns -1234

```
DECFLOAT_FORMAT( '-1234', 'S9999' )
```

- *Example 11*: The following example returns 1234

```
DECFLOAT_FORMAT( ' 1234 ', '9999PR' )
```

- *Example 12*: The following example returns -1234

```
DECFLOAT_FORMAT( '<1234>', '9999PR' )
```

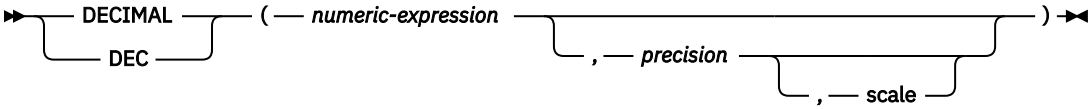
- *Example 13*: The following example returns 123456.78

```
DECFLOAT_FORMAT( '$123,456.78', '$999,999.99' )
```

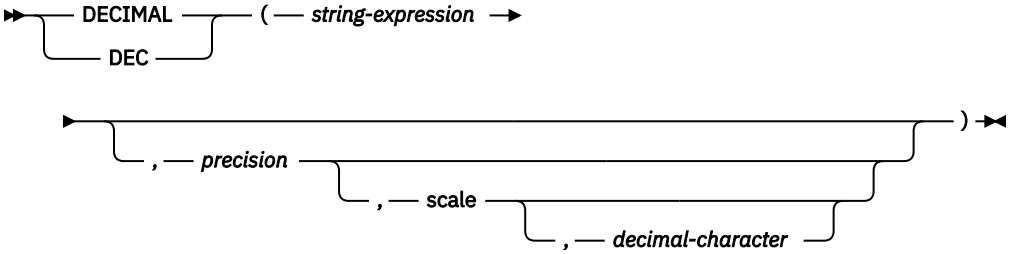
DECIMAL or DEC

The DECIMAL function returns a decimal representation of a number, a string representation of a number, or a datetime value.

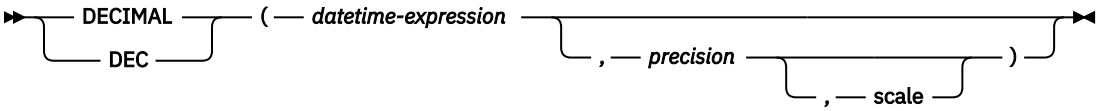
Numeric to DECIMAL



String to DECIMAL



Datetime to DECIMAL



The schema is SYSIBM.

Numeric to DECIMAL

numeric-expression
An expression that returns a value of any built-in numeric data type.

precision
An integer constant with a value in the range 1 - 31. The default precision depends on the data type of the input expression:

- 31 for decimal floating point (DECFLOAT)
- 15 for floating point (REAL or DOUBLE) or decimal (DECIMAL)
- 19 for big integer (BIGINT)
- 11 for large integer (INTEGER)
- 5 for small integer (SMALLINT)

scale
An integer constant in the range of 0 to the precision value. The default scale is zero.

The result is the same number that would occur if the first argument were assigned to a decimal column or variable with a precision of *precision* and a scale of *scale*. Digits are truncated from the end of the decimal number if the number of digits to the right of the decimal separator character is greater than the scale *scale*. An error is returned if the number of significant decimal digits required to represent the whole part of the number is greater than *precision* - *scale* (SQLSTATE 22003).

String to DECIMAL

string-expression
An *expression* that returns a value that is a character string or a Unicode graphic-string representation of a number with a length not greater than the maximum length of a character

constant. The data type of *string-expression* must not be CLOB or DBCLOB (SQLSTATE 42884). Leading and trailing blanks are eliminated from the string and the resulting string must conform to the rules for forming an integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018).

The *string-expression* is converted to the section code page if required to match the code page of the constant *decimal-character*.

precision
An integer constant with a value in the range 1 to 31 that specifies the precision of the result. If not specified, the default is 15.

scale
An integer constant with a value in the range 0 to *precision* that specifies the scale of the result. If not specified, the default is 0.

decimal-character
Specifies the single-byte character constant used to delimit the decimal digits in *string-expression* from the whole part of the number. The character cannot be a digit, plus (+), minus (-), or blank, and it can appear at most once in *string-expression* (SQLSTATE 42815).

The result is the same number that would result from CAST(*string-expression* AS DECIMAL(*precision*, *scale*)). Digits are truncated from the end of the decimal number if the number of digits to the right of the decimal separator character is greater than the scale *scale*. An error is returned if the number of significant digits to the left of the decimal character (the whole part of the number) in *string-expression* is greater than *precision* - *scale* (SQLSTATE 22003). The default decimal character is not valid in the substring if a different value for the *decimal-character* argument is specified (SQLSTATE 22018).

In NPS mode, casting an empty string to DECIMAL returns 0.

Datetime to DECIMAL

datetime-expression
An expression that returns a value of type DATE, TIME or TIMESTAMP.

precision
An integer constant with a value in the range 1 to 31 that specifies the precision of the result. If not specified, the default for the precision and scale depends on the data type of *datetime-expression* as follows:

- Precision is 8 and scale is 0 for a DATE. The result is a DECIMAL(8,0) value representing the date as *yyyymmdd*.
- Precision is 6 and scale is 0 for a TIME. The result is a DECIMAL(6,0) value representing the time as *hhmmss*.
- Precision is 14+*tp* and scale is *tp* for a TIMESTAMP(*tp*). The result is a DECIMAL(14+*tp*,*tp*) value representing the timestamp as *yyyymmddhhmmss.nnnnnnnnnnn*.

scale
An integer constant with a value in the range 0 to *precision* that specifies the scale of the result. If not specified and a *precision* is specified, the default is 0.

The result is the same number that would result from CAST(*datetime* - *expression* AS DECIMAL(*precision*, *scale*)). Digits are truncated from the end of the decimal number if the number of digits to the right of the decimal separator character is greater than the scale *scale*. An error is returned if the number of significant digits to the left of the decimal character (the whole part of the number) in *string-expression* is greater than *precision* - *scale* (SQLSTATE 22003).

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Note: The CAST specification should be used to increase the portability of applications. For more information, see "CAST specification".

Examples

- *Example 1*: Use the DECIMAL function in order to force a DECIMAL data type (with a precision of 5 and a scale of 2) to be returned in a select-list for the EDLEVEL column (data type = SMALLINT) in the EMPLOYEE table. The EMPNO column should also appear in the select list.

```
SELECT EMPNO, DECIMAL(EDLEVEL,5,2)
  FROM EMPLOYEE
```

- *Example 2*: Assume the host variable PERIOD is of type INTEGER. Then, in order to use its value as a date duration it must be "cast" as decimal(8,0).

```
SELECT PRSTDATE + DECIMAL(:PERIOD,8)
  FROM PROJECT
```

- *Example 3*: Assume that updates to the SALARY column are input through a window as a character string using comma as a decimal character (for example, the user inputs 21400,50). Once validated by the application, it is assigned to the host variable *newsalary* which is defined as CHAR(10).

```
UPDATE STAFF
  SET SALARY = DECIMAL(:newsalary, 9, 2, ',')
  WHERE ID = :empid;
```

  The value of *newsalary* becomes 21400.50.

- *Example 4*: Add the default decimal character (.) to a value.

```
DECIMAL('21400,50', 9, 2, '.')
```

  This fails because a period (.) is specified as the decimal character, but a comma (,) appears in the first argument as a delimiter.

- *Example 5*: Assume that the column STARTING (whose data type is TIME) has an internal value equivalent to '12:10:00'.

```
DECIMAL(STARTING)
```

  results in the value `121 000`.

- *Example 6*: Assume that the column RECEIVED (whose data type is TIMESTAMP) has an internal value equivalent to '1988-12-22-14.07.21.136421'.

```
DECIMAL(RECEIVED)
```

  results in the value `19 881 222 140 721.136421`.

- *Example 7*: This example shows the decimal result and resulting precision and scale for various datetime input values. Assume the existence of the following columns with associated values:

Column name	Data type	Value
ENDDT	DATE	2000-03-21
ENDTM	TIME	12:02:21
ENDTS	TIMESTAMP	2000-03-21-12.02.21.123456
ENDTS0	TIMESTAMP(0)	2000-03-21-12.02.21
ENDTS9	TIMESTAMP(9)	2000-03-21-12.02.21.123456789

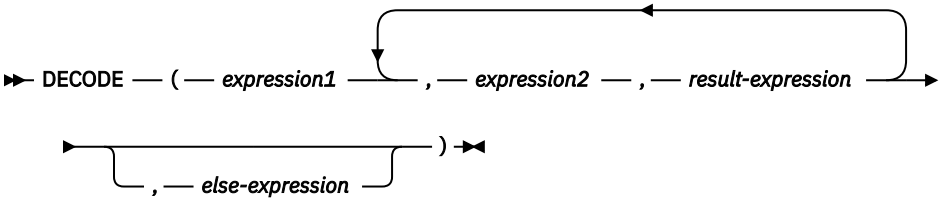
  The following table shows the decimal result and resulting precision and scale for various datetime input values.

DECIMAL(arguments)	Precision and Scale	Result
DECIMAL(ENDDT)	(8,0)	20000321.

DECIMAL(arguments)	Precision and Scale	Result
DECIMAL(ENDDT, 10)	(10,0)	20000321.
DECIMAL(ENDDT, 12, 2)	(12,2)	20000321.00
DECIMAL(ENDTM)	(6,0)	120221.
DECIMAL(ENDTM, 10)	(10,0)	120221.
DECIMAL(ENDTM, 10, 2)	(10,2)	120221.00
DECIMAL(ENDTS)	(20, 6)	20000321120221.123456
DECIMAL(ENDTS, 23)	(23, 0)	20000321120221.
DECIMAL(ENDTS, 23, 4)	(23, 4)	20000321120221.1234
DECIMAL(ENDTS0)	(14,0)	20000321120221.
DECIMAL(ENDTS9)	(23,9)	20000321120221.123456789

DECODE

The DECODE function does equality comparisons between arguments, also treating null values as equal, to determine which argument to return as the result.



The schema is SYSIBM.

The DECODE function compares each *expression2* to *expression1*. If *expression1* is equal to *expression2*, or both *expression1* and *expression2* are null, the value of the following *result-expresssion* is returned. If no *expression2* matches *expression1*, the value of *else-expression* is returned; otherwise a null value is returned.

The DECODE function is similar to the CASE expression except for the handling of null values:

- A null value of *expression1* will match a corresponding null value of *expression2*.
- If the NULL keyword is used as an argument in the DECODE function, it must be cast to an appropriate data type.

The rules for determining the result type of a DECODE expression are based on the corresponding CASE expression.

Examples

- *Example 1*: The DECODE expression:

```
DECODE (c1, 7, 'a', 6, 'b', 'c')
```

  achieves the same result as the following CASE expression:

```
CASE c1
  WHEN 7 THEN 'a'
  WHEN 6 THEN 'b'
  ELSE 'c'
END
```

- *Example 2*: The DECODE expression:

```
DECODE (c1, var1, 'a', var2, 'b')
```

  where the values of c1, var1, and var2 could be null values, achieves the same result as the following CASE expression:

```
CASE
  WHEN c1 = var1 OR (c1 IS NULL AND var1 IS NULL) THEN 'a'
  WHEN c1 = var2 OR (c1 IS NULL AND var2 IS NULL) THEN 'b'
  ELSE NULL
END
```

- *Example 3*: Consider also the following query:

```
SELECT ID, DECODE(STATUS, 'A', 'Accepted',
                          'D', 'Denied',
                          CAST(NULL AS VARCHAR(1)), 'Unknown',
                          'Other')
FROM CONTRACTS
```

  Here is the same statement using a CASE expression:

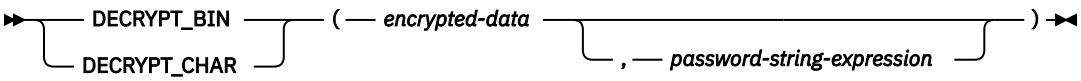
```
SELECT ID,
  CASE
    WHEN STATUS = 'A' THEN 'Accepted'
    WHEN STATUS = 'D' THEN 'Denied'
    WHEN STATUS IS NULL THEN 'Unknown'
    ELSE 'Other'
  END
FROM CONTRACTS
```

DECRYPT_BIN and DECRYPT_CHAR

The DECRYPT_BIN and DECRYPT_CHAR functions both return a value that is the result of decrypting *encrypted-data*.

Important: The DECRYPT_BIN and DECRYPT_CHAR functions are deprecated and might not appear in future releases.

Note: If using this feature on AIX, review the following for performance considerations.



The schema is SYSIBM.

The password used for decryption is either the *password-string-expression* value or the encryption password value that was assigned by the SET ENCRYPTION PASSWORD statement. To maintain the best level of security on your system, it is recommended that you do not pass the encryption password explicitly with the DECRYPT_BIN and DECRYPT_CHAR functions in your query; instead, use the SET ENCRYPTION PASSWORD statement to set the password, and use a host variable or dynamic parameter markers when you use the SET ENCRYPTION PASSWORD statement, rather than a literal string.

The DECRYPT_BIN and DECRYPT_CHAR functions can only decrypt values that are encrypted using the ENCRYPT function (SQLSTATE 428FE).

encrypted-data
An expression that returns a CHAR FOR BIT DATA or VARCHAR FOR BIT DATA value as a complete, encrypted data string. The data string must have been encrypted using the ENCRYPT function.

password-string-expression
An expression that returns a CHAR or VARCHAR value with at least 6 bytes and no more than 127 bytes (SQLSTATE 428FC). This expression must be the same password used to encrypt the data (SQLSTATE 428FD). If the value of the password argument is null or not provided, the data will

be decrypted using the encryption password value that was assigned for the session by the SET ENCRYPTION PASSWORD statement (SQLSTATE 51039).

The result of the DECRYPT_BIN function is VARCHAR FOR BIT DATA. The result of the DECRYPT_CHAR function is VARCHAR. If *encrypted-data* included a hint, the hint is not returned by the function. The length attribute of the result is the length of the data type of *encrypted-data* minus 8 bytes. The actual length of the value returned by the function will match the length of the original string that was encrypted. If *encrypted-data* includes bytes beyond the encrypted string, these bytes are not returned by the function.

If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

If the data is decrypted on a different system, which uses a code page that is different from the code page in which the data was encrypted, expansion might occur when converting the decrypted value to the database code page. In such situations, the *encrypted-data* value should be cast to a VARCHAR string with a larger number of bytes.

Examples

- *Example 1*: The following example demonstrates the use of the DECRYPT_CHAR function by showing code fragments from an embedded SQL application.

```
EXEC SQL BEGIN DECLARE SECTION;
      char hostVarCreateTableStmt[100];
      char hostVarSetEncPassStmt[200];
      char hostVarPassword[128];
      char hostVarInsertStmt1[200];
      char hostVarInsertStmt2[200];
      char hostVarSelectStmt1[200];
      char hostVarSelectStmt2[200];
EXEC SQL END DECLARE SECTION;

/* prepare the statement */
strcpy(hostVarCreateTableStmt, "CREATE TABLE EMP (SSN VARCHAR(24) FOR BIT DATA)");
EXEC SQL PREPARE hostVarCreateTableStmt FROM :hostVarCreateTableStmt;

/* execute the statement */
EXEC SQL EXECUTE hostVarCreateTableStmt;
```

- *Example 2*: Use the SET ENCRYPTION PASSWORD statement to set an encryption password for the session:

```
/* prepare the statement with a parameter marker */
strcpy(hostVarSetEncPassStmt, "SET ENCRYPTION PASSWORD = ?");
EXEC SQL PREPARE hostVarSetEncPassStmt FROM :hostVarSetEncPassStmt;

/* execute the statement for hostVarPassword = 'Pac1f1c' */
strcpy(hostVarPassword, "Pac1f1c");
EXEC SQL EXECUTE hostVarSetEncPassStmt USING :hostVarPassword;

/* prepare the statement */
strcpy(hostVarInsertStmt1, "INSERT INTO EMP(SSN) VALUES ENCRYPT('289-46-8832')");
EXEC SQL PREPARE hostVarInsertStmt1 FROM :hostVarInsertStmt1;

/* execute the statement */
EXEC SQL EXECUTE hostVarInsertStmt1;

/* prepare the statement */
strcpy(hostVarSelectStmt1, "SELECT DECRYPT_CHAR(SSN) FROM EMP");
EXEC SQL PREPARE hostVarSelectStmt1 FROM :hostVarSelectStmt1;

/* execute the statement */
EXEC SQL EXECUTE hostVarSelectStmt1;
```

This query returns the value '289-46-8832'.

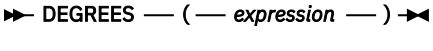
- *Example 3*: Pass the encryption password explicitly:

```
/* prepare the statement */
strcpy(hostVarInsertStmt2, "INSERT INTO EMP (SSN) VALUES
ENCRYPT('289-46-8832',?)");
EXEC SQL PREPARE hostVarInsertStmt2 FROM :hostVarInsertStmt2;

/* execute the statement for hostVarPassword = 'Pac1f1c' */
strcpy(hostVarPassword, "Pac1f1c");
EXEC SQL EXECUTE hostVarInsertStmt2 USING :hostVarPassword;

/* prepare the statement */
strcpy(hostVarSelectStmt2, "SELECT DECRYPT_CHAR(SSN,?) FROM EMP");
EXEC SQL PREPARE hostVarSelectStmt2 FROM :hostVarSelectStmt2;

/* execute the statement for hostVarPassword = 'Pac1f1c' */
strcpy(hostVarPassword, "Pac1f1c");
EXEC SQL EXECUTE hostVarSelectStmt2 USING :hostVarPassword;
```

This query returns the value '289-46-8832'.

DEGREES

The DEGREES function returns the number of degrees of the argument, which is an angle expressed in radians.



The schema is SYSIBM. (The SYSFUN version of the DEGREES function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type. If the value is of decimal floating-point data type, the operation is performed in decimal floating-point; otherwise, the value is converted to double-precision floating-point for processing by the function.

If the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*); otherwise, the result is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example

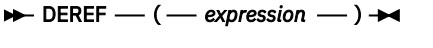
Assume that RAD is a DECIMAL(4,3) host variable with a value of 3.142.

```
VALUES DEGREES(:RAD)
```

Returns the approximate value 180.0.

DEREF

The DEREF function returns an instance of the target type of the argument.



expression
An expression that returns a value with a reference data type that has a defined scope (SQLSTATE 428DT).

The static data type of the result is the target type of the argument. The dynamic data type of the result is a subtype of the target type of the argument. The result can be null. The result is the null value if *expression* is a null value or if *expression* is a reference that has no matching OID in the target table.

The result is an instance of the subtype of the target type of the reference. The result is determined by finding the row of the target table or target view of the reference that has an object identifier that matches the reference value. The type of this row determines the dynamic type of the result. Since the type of the result can be based on a row of a subtable or subview of the target table or target view, the authorization

ID of the statement must have SELECT privilege on the target table and all of its subtables or the target view and all of its subviews (SQLSTATE 42501).

Example

Assume that EMPLOYEE is a table of type EMP, and that its object identifier column is named EMPID. Then the following query returns an object of type EMP (or one of its subtypes), for each row of the EMPLOYEE table (and its subtables). This query requires SELECT privilege on EMPLOYEE and all its subtables.

```
SELECT DEREF(EMPID) FROM EMPLOYEE
```

DIFFERENCE

Returns a value from 0 to 4 representing the difference between the sounds of two strings based on applying the SOUNDEX function to the strings. A value of 4 is the best possible sound match.

The schema is SYSFUN.

expression
An expression that returns a character string or a Boolean value. In a Unicode database, the expression can also return a graphic string. If the returned value is not a character string, it is cast to a character string before the function is evaluated. The character string to be evaluated cannot exceed 4000 bytes. This function interprets data that is passed to it as if the data were made up of ASCII characters, even if it is encoded in UTF-8.

Result

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

Example

The following code:

```
VALUES (DIFFERENCE('CONSTRAINT','CONSTANT'),SOUNDEX('CONSTRAINT'),
SOUNDEX('CONSTANT')),
(DIFFERENCE('CONSTRAINT','CONTRITE'),SOUNDEX('CONSTRAINT'),
SOUNDEX('CONTRITE'))
```

returns the following output:

```
1           2    3
----------- ---- ----
          4 C523 C523
          2 C523 C536
```

In the first row, the words have the same result from SOUNDEX while in the second row the words have only some similarity.

DIGITS

The DIGITS function returns a character-string representation of a number.

The schema is SYSIBM.

expression

An expression that returns a value of one of the following built-in data types: SMALLINT, INTEGER, BIGINT, DECIMAL, CHAR, or VARCHAR. In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to a character string before the function is executed. A CHAR or VARCHAR value is implicitly cast to DECIMAL(31,6) before evaluating the function.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result of the function is a fixed-length character string representing the absolute value of the argument without regard to its scale. The result does not include a sign or a decimal character. Instead, it consists exclusively of digits, including, if necessary, leading zeros to fill out the string. The length of the string is:

- 5 if the argument is a small integer
- 10 if the argument is a large integer
- 19 if the argument is a big integer
- *p* if the argument is a decimal number with a precision of *p*.

Examples

- *Example 1*: Assume that a table called TABLEX contains an INTEGER column called INTCOL containing 10-digit numbers. List all distinct four digit combinations of the first four digits contained in column INTCOL.

```
SELECT DISTINCT SUBSTR(DIGITS(INTCOL),1,4)
  FROM TABLEX
```

- *Example 2*: Assume that COLUMNX has the DECIMAL(6,2) data type, and that one of its values is -6.28. Then, for this value:

```
DIGITS(COLUMNX)
```

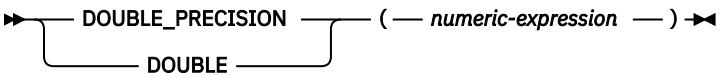
  returns the value '000628'.

  The result is a string of length six (the precision of the column) with leading zeros padding the string out to this length. Neither sign nor decimal point appear in the result.

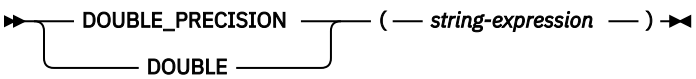
DOUBLE_PRECISION or DOUBLE

The DOUBLE_PRECISION and DOUBLE functions return a double-precision floating-point representation of either a number or a string representation of a number.

Numeric to DOUBLE



String to DOUBLE



The schema is SYSIBM.

Numeric to DOUBLE

numeric-expression
An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a double-precision floating-point column or variable. If the numeric value of the argument is not within the range of double-precision floating-point, an error is returned (SQLSTATE 22003).

String to DOUBLE

string-expression
An expression that returns a string, including FOR BIT DATA, that represents a number. The data type of this expression cannot be CLOB, BLOB, or DBCLOB (SQLSTATE 42884).

The result is the same number that would result from the statement CAST(*string-expression* AS DOUBLE PRECISION). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming a valid numeric constant (SQLSTATE 22018). If the numeric value of the argument is not within the range of double-precision floating-point, an error is returned (SQLSTATE 22003).

The result of the function is a double-precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Notes

- The CAST specification should be used to increase the portability of applications.
- FLOAT is a synonym for DOUBLE_PRECISION and DOUBLE.
- The SYSFUN version of DOUBLE (*string_expression*) continues to be available.

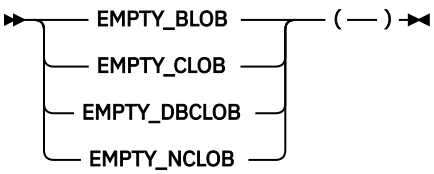
Example

Using the EMPLOYEE table, find the ratio of salary to commission for employees whose commission is not zero. The columns involved (SALARY and COMM) have DECIMAL data types. To eliminate the possibility of out-of-range results, DOUBLE is applied to SALARY so that the division is carried out in floating point:

```
SELECT EMPNO, DOUBLE(SALARY)/COMM
  FROM EMPLOYEE
  WHERE COMM > 0
```

EMPTY_BLOB, EMPTY_CLOB, EMPTY_DBCLOB, and EMPTY_NCLOB

These functions return a zero-length value with a data type of BLOB, CLOB, or DBCLOB.



The schema is SYSIBM.

The empty value functions return a zero-length value of the associated data type. There are no arguments to these functions (the empty parentheses must be specified).

- The EMPTY_BLOB function returns a zero-length value with a data type of BLOB(1).
- The EMPTY_CLOB function returns a zero-length value with a data type of CLOB(1).
- The EMPTY_DBCLOB and EMPTY_NCLOB functions return a zero-length value with a data type of DBCLOB(1).

The result of these functions can be used in assignments to provide zero-length values where needed.

The EMPTY_NCLOB function can be specified only in a Unicode database (SQLSTATE 560AA).

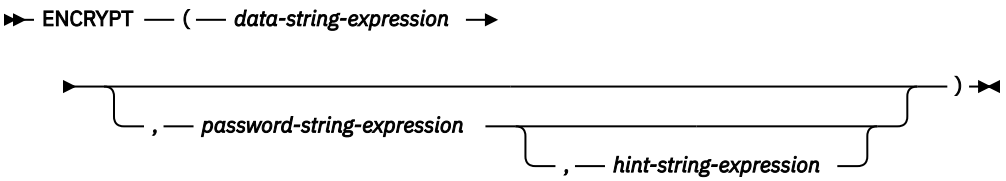
ENCRYPT

The ENCRYPT function returns a value that is the result of encrypting *data-string-expression*.

Important: The ENCRYPT function is deprecated and might not appear in future releases.

The algorithms used are not compliant with **NIST SP 800-131A**.

Note: If using this feature on AIX, review the following for performance considerations.



The schema is SYSIBM.

The password used for encryption is either the *password-string-expression* value or the encryption password value that was assigned by the SET ENCRYPTION PASSWORD statement. To maintain the best level of security on your system, it is recommended that you do not pass the encryption password explicitly with the ENCRYPT function in your query; instead, use the SET ENCRYPTION PASSWORD statement to set the password, and use a host variable or dynamic parameter markers when you use the SET ENCRYPTION PASSWORD statement, rather than a literal string.

In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

data-string-expression
An expression that returns a CHAR or a VARCHAR value that is to be encrypted. The length attribute for the data type of *data-string-expression* is limited to 32663 without a *hint-string-expression* argument, and 32631 when the *hint-string-expression* argument is specified (SQLSTATE 42815).

password-string-expression
An expression that returns a CHAR or a VARCHAR value with at least 6 bytes and no more than 127 bytes (SQLSTATE 428FC). The value represents the password used to encrypt *data-string-expression*. If the value of the password argument is null or not provided, the data is encrypted using the encryption password value that was assigned for the session by the SET ENCRYPTION PASSWORD statement (SQLSTATE 51039).

hint-string-expression
An expression that returns a CHAR or a VARCHAR value with at most 32 bytes that will help data owners remember passwords (for example, 'Ocean' as a hint to remember 'Pacific'). If a hint value is given, the hint is embedded into the result and can be retrieved using the GETHINT function. If this argument is null or not provided, no hint will be embedded in the result.

The result data type of the function is VARCHAR FOR BIT DATA.

- When the optional hint parameter is specified, the length attribute of the result is equal to the length attribute of the unencrypted data + 8 bytes + the number of bytes until the next 8-byte boundary + 32 bytes for the length of the hint.
- When the optional hint parameter is not specified, the length attribute of the result is equal to the length attribute of the unencrypted data + 8 bytes + the number of bytes until the next 8-byte boundary.

If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

Note that the encrypted result is longer than the *data-string-expression* value. Therefore, when assigning encrypted values, ensure that the target is declared with sufficient size to contain the entire encrypted value.

Notes

- ***Encryption algorithm:*** The internal encryption algorithm is RC2 block cipher with padding; the 128-bit secret key is derived from the password using an MD5 message digest.
- ***Encryption passwords and data:*** Password management is the user's responsibility. Once the data is encrypted, only the password that was used when encrypting it can be used to decrypt it (SQLSTATE 428FD).

  The encrypted result might contain null terminator and other unprintable characters. Any assignment or cast to a length that is shorter than the suggested data length might result in failed decryption in the future, and lost data. Blanks are valid encrypted data values that might be truncated when stored in a column that is too short.
- ***Administration of encrypted data:*** Encrypted data can only be decrypted on servers that support the decryption functions corresponding to the ENCRYPT function. Therefore, replication of columns with encrypted data should only be done to servers that support the DECRYPT_BIN or the DECRYPT_CHAR function.
- ***Avoid encrypted data in predicates:*** The ENCRYPT function does not always produce the same encrypted data when given the same input. Do not use encrypted data in search conditions or comparison operations. For example, do not use encrypted data in a predicate.

Examples

- *Example 1*: The following example demonstrates the use of the ENCRYPT function by showing code fragments from an embedded SQL application.

```
EXEC SQL BEGIN DECLARE SECTION;
      char hostVarCreateTableStmt[100];
      char hostVarSetEncPassStmt[200];
      char hostVarPassword[128];
      char hostVarInsertStmt1[200];
      char hostVarInsertStmt2[200];
      char hostVarInsertStmt3[200];
EXEC SQL END DECLARE SECTION;

/* prepare the statement */
strcpy(hostVarCreateTableStmt, "CREATE TABLE EMP (SSN VARCHAR(24) FOR BIT DATA)");
EXEC SQL PREPARE hostVarCreateTableStmt FROM :hostVarCreateTableStmt;

/* execute the statement */
EXEC SQL EXECUTE hostVarCreateTableStmt;
```

- *Example 2*: Use the SET ENCRYPTION PASSWORD statement to set an encryption password for the session:

```
/* prepare the statement with a parameter marker */
strcpy(hostVarSetEncPassStmt, "SET ENCRYPTION PASSWORD = ?");
EXEC SQL PREPARE hostVarSetEncPassStmt FROM :hostVarSetEncPassStmt;

/* execute the statement for hostVarPassword = 'Pac1f1c' */
strcpy(hostVarPassword, "Pac1f1c");
EXEC SQL EXECUTE hostVarSetEncPassStmt USING :hostVarPassword;

/* prepare the statement */
strcpy(hostVarInsertStmt1, "INSERT INTO EMP(SSN) VALUES ENCRYPT('289-46-8832')");
EXEC SQL PREPARE hostVarInsertStmt1 FROM :hostVarInsertStmt1;

/* execute the statement */
EXEC SQL EXECUTE hostVarInsertStmt1;
```

- *Example 3*: Pass the encryption password explicitly:

```
/* prepare the statement */
strcpy(hostVarInsertStmt2, "INSERT INTO EMP(SSN) VALUES
ENCRYPT('289-46-8832',?)");
```

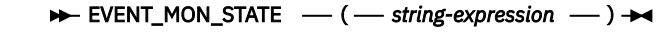
```
EXEC SQL PREPARE hostVarInsertStmt2 FROM :hostVarInsertStmt2;

/* execute the statement for hostVarPassword = 'Pac1f1c' */
strcpy(hostVarPassword, "Pac1f1c");
EXEC SQL EXECUTE hostVarInsertStmt2 USING :hostVarPassword;
```

- *Example 4*: Define a password hint:

```
/* prepare the statement */
strcpy(hostVarInsertStmt3, "INSERT INTO EMP(SSN) VALUES ENCRYPT('289-46-8832',?,'Ocean')");
EXEC SQL PREPARE hostVarInsertStmt3 FROM :hostVarInsertStmt3;

/* execute the statement for hostVarPassword = 'Pac1f1c' */
strcpy(hostVarPassword, "Pac1f1c");
EXEC SQL EXECUTE hostVarInsertStmt3 USING :hostVarPassword;
```

EVENT_MON_STATE

The EVENT_MON_STATE function returns the current state of an event monitor.



The schema is SYSIBM.

string-expression
An expression that returns a value of CHAR or VARCHAR data type. In a Unicode database, if the value is a graphic string, it is first converted to a character string before the function is executed. The value must be the name of an event monitor that is equal to an event monitor name in the SYSCAT.EVENMONITORS catalog view (SQLSTATE 42704).

The result is an integer with one of the following values:

0
The event monitor is inactive.

1
The event monitor is active.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

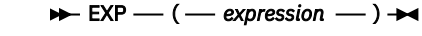
Example

The following example selects all of the defined event monitors, and indicates whether each is active or inactive:

```
SELECT EVMONNAME,
  CASE
    WHEN EVENT_MON_STATE(EVMONNAME) = 0 THEN 'Inactive'
    WHEN EVENT_MON_STATE(EVMONNAME) = 1 THEN 'Active'
  END
  FROM SYSCAT.EVENTMONITORS
```

EXP

The EXP function returns a value that is the base of the natural logarithm (e) raised to a power specified by the argument. The EXP and LN functions are inverse operations.



The schema is SYSIBM. (The SYSFUN version of the EXP function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type. If the value is of decimal floating-point data type, the operation is performed in decimal floating-point; otherwise, the value is converted to double-precision floating-point for processing by the function.

If the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*); otherwise, the result is a double-precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Notes

- **Results involving DECFLOAT special values:** For decimal floating-point values, the special values are treated as follows:
  - EXP(NaN) returns NaN.
  - EXP(-NaN) returns -NaN.
  - EXP(Infinity) returns Infinity.
  - EXP(-Infinity) returns 0.
  - EXP(sNaN) returns NaN and a warning.
  - EXP(-sNaN) returns -NaN and a warning.

Example

Assume that E is a DECIMAL(10,9) host variable with a value of 3.453789832.

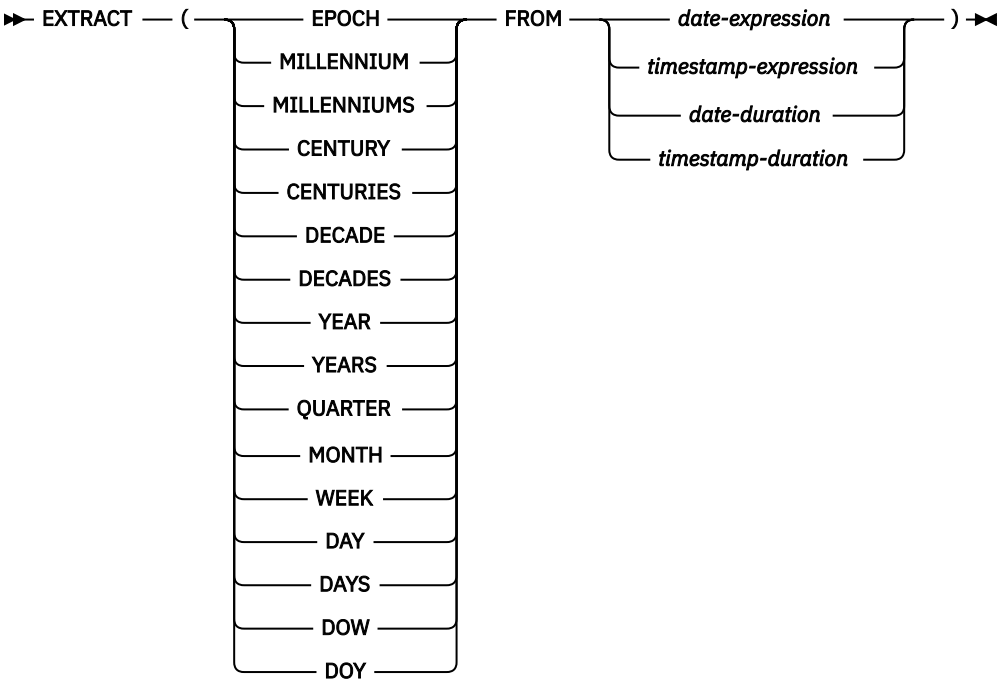
```
VALUES EXP(:E)
```

Returns the DOUBLE value +3.16200000069145E+001.

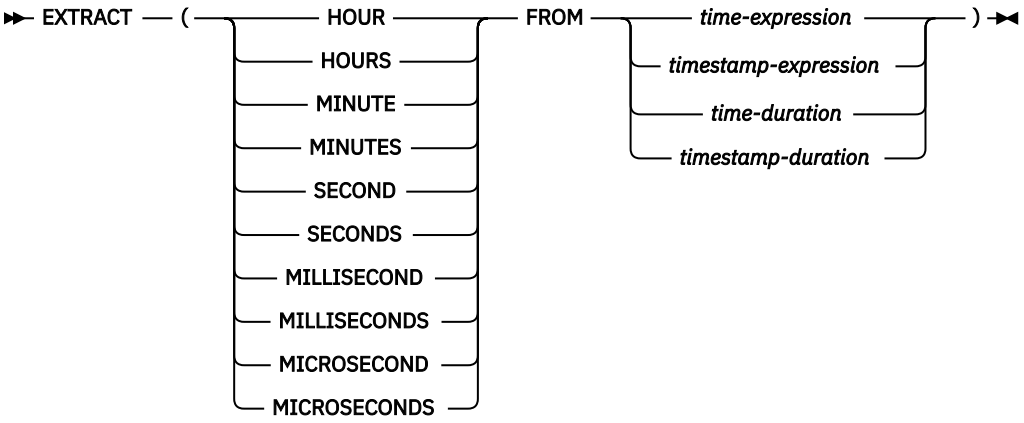
EXTRACT

The EXTRACT function returns a portion of a datetime based on its arguments.

Extract date values



Extract time values



The schema is SYSIBM.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Extract date values

EPOCH
The number of seconds since 1970-01-01 00:00:00.00 for the specified date or timestamp expression is returned. A duration cannot be specified (SQLCODE SQL0171, SQLSTATE 22546). The value is positive if the expression represents a date or timestamp that is after 1970-01-01 00:00:00.00; the value is negative for a date or timestamp that is before 1970-01-01 00:00:00.00.

MILLENNIUM or MILLENNIUMS
The ordinal number of the full 1000-year period of the specified date or timestamp expression or duration is returned; for example, 2 for a date between 01 Jan 2000 and 31 Dec 2999

CENTURY or CENTURIES
The ordinal number of the full 100-year period of the specified date or timestamp expression or duration is returned; for example, 20 for a date between 01 Jan 2000 and 31 Dec 2099. Not to be confused with the ordinal system that counts dates up to the year 100 as being in the "first century", dates between 01 Jan 2000 and 31 Dec 2099 as being in the "21st century", etc.

DECADE or DECADES
The ordinal number of the full 10-year period of the specified date or timestamp expression or duration is returned; for example, 201 for a date between 01 Jan 2010 and 31 Dec 2019.

YEAR or YEARS
The years portion of the specified date or timestamp expression or duration is returned. The result is identical to that returned by the YEAR scalar function.

QUARTER
The quarter (1, 2, 3, or 4) of the year of the specified date or timestamp expression or duration is returned.

MONTH
The number (1 - 12) of the month of the specified date or timestamp expression or duration is returned. The result is identical to that returned by the MONTH scalar function.

WEEK
The number (1 - 53) of the week of the year of the specified date or timestamp expression or duration is returned. The value uses the ISO-8601 definition of a week, which begins on Monday; as a result, some years might have 53 weeks, and sometimes the first few days of January can be included as part of the 52nd or 53rd week of the previous year.

DAY or DAYS
The number (1 - 31) of the day of the specified date or timestamp expression or duration is returned. The result is identical to that returned by the DAY scalar function.

DOW
A number (1 for Sunday, 2 for Monday, ..., 7 for Saturday) indicating the day of the week of the specified date or timestamp expression is returned. A duration cannot be specified (SQLCODE SQL0171, SQLSTATE 22546).

DOY
A number (1 - 366) indicating the day of the year of the specified date or timestamp expression is returned. A duration cannot be specified (SQLCODE SQL0171, SQLSTATE 22546).

date-expression
An expression that returns the value of either a built-in DATE or built-in character string data type.

If a date expression is a character string, it must be a valid string representation of a date that is not a CLOB. In a Unicode database, if a date expression is a graphic string, it is first converted to a character string before the function is executed.

timestamp-expression
An expression that returns the value of either a built-in TIMESTAMP or built-in character string data type.

If a timestamp expression is a character string, it must be a valid string representation of a timestamp that is not a CLOB. In a Unicode database, if a timestamp expression is a graphic string, it is first converted to a character string before the function is executed.

date-duration
A DECIMAL(8,0) number that specifies a date duration (see "Datetime operations and durations" on page 145).

timestamp-duration
A DECIMAL(14+*s*,*s*) number that specifies a timestamp duration, where *s* represents the number of digits of fractional seconds ranging from 0 to 12 (see "Datetime operations and durations" on page 145).

Extract time values

HOUR or HOURS
The hours portion of the specified time or timestamp expression or duration is returned. The result is identical to that returned by the HOUR scalar function.

MINUTE or MINUTES
The minutes portion of the specified time or timestamp expression or duration is returned. The result is identical to that returned by the MINUTE scalar function.

SECOND or SECONDS
The seconds portion of the specified time or timestamp expression or duration is returned. The result is identical to that returned by:

- SECOND(*expression*, 6) when the data type of the expression or duration is a TIME value or a string representation of a TIME or TIMESTAMP
- SECOND(*expression*, *s*) when the data type of the expression or duration is a TIMESTAMP(*s*) value

MILLISECOND or MILLISECONDS
The seconds portion of the specified timestamp expression or duration, including fractional parts to one thousandth of a second, multiplied by 1000 (0 - 59999) is returned. A time expression or duration cannot be specified (SQLCODE SQL0180, SQLSTATE 22007).

MICROSECOND or MICROSECONDS
The seconds portion of the specified timestamp expression or duration, including fractional parts to one millionth of a second, multiplied by 1000000 (0 - 59999999) is returned. A time expression or duration cannot be specified (SQLCODE SQL0180, SQLSTATE 22007).

time-expression
An expression that returns the value of either a built-in TIME or built-in character string data type.

If a time expression is a character string, it must be a valid string representation of a time that is not a CLOB. In a Unicode database, if a time expression is a graphic string, it is first converted to a character string before the function is executed.

timestamp-expression
An expression that returns the value of a built-in DATE, TIMESTAMP or character string data type.

If *timestamp-expression* is a DATE, it is converted to a TIMESTAMP(0) value, assuming a time of exactly midnight (00.00.00).

If a timestamp expression is a character string, it must be a valid string representation of a timestamp or date that is not a CLOB. In a Unicode database, if a timestamp expression is a graphic string, it is first converted to a character string before the function is executed. The string is converted to a TIMESTAMP(6) value.

time-duration
A DECIMAL(6,0) number that specifies a time duration (see "Datetime operations and durations" on page 145).

timestamp-duration
A DECIMAL(14+*s*,*s*) number that specifies a timestamp duration, where *s* represents the number of digits of fractional seconds ranging from 0 to 12 (see "Datetime operations and durations" on page 145).

The data type of the result of the function depends on the part of the datetime value that is specified:

- If MILLENNIUM, CENTURY, DECADE, YEAR, QUARTER, MONTH, WEEK, DAY, DOW, DOY, HOUR, or MINUTE is specified, the data type of the result is INTEGER.
- If SECOND is specified with a TIMESTAMP(*p*) value, the data type of the result is DECIMAL(2+*p*, *p*) where *p* is the fractional seconds precision.
- If SECOND is specified with a TIME value or a string representation of a TIME or TIMESTAMP, the data type of the result is DECIMAL(8,6).
- If MILLISECOND, MILLISECONDS, MICROSECOND, or MICROSECONDS is specified with a TIMESTAMP(p) value, the data type of the result is INTEGER.
- If EPOCH is specified, the data type of result is BIGINT.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

In a table with the name PROJECT:

- Column col1 contains the date value '1988-12-25'. The following statement returns the integer value 12:

```
SELECT EXTRACT(MONTH FROM col1) FROM PROJECT;
```

- Column col2 contains the timestamp value '2007-02-14 12:15:06.123456'. The following statement returns the integer value 6123:

```
SELECT EXTRACT(MILLISECONDS FROM col2) FROM PROJECT;
```

  The following statement returns the integer value 6123456:

```
SELECT EXTRACT(MICROSECONDS FROM col2) FROM PROJECT;
```

- Column col3 contains the date value '2013-02-14'. The following statement returns the integer value 201:

```
SELECT EXTRACT(DECADE FROM col3) FROM PROJECT;
```

- Column col4 has a data type of DECIMAL(8,0) and contains the value 12301000. Because the data type is DECIMAL(8,0), this value will be interpreted as a date duration (1230 years, 10 months, 00 days). The following statement returns the integer value 10:

```
SELECT EXTRACT(MONTH FROM col4) FROM PROJECT;
```

- Column col5 has a data type of DECIMAL(6,0) and contains the value 123010. Because the data type is DECIMAL(6,0), this value will be interpreted as a time duration (12 hours, 30 minutes, 10 seconds). The following statement returns the integer value 10:

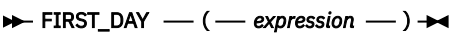
```
SELECT EXTRACT(SECONDS FROM col5) FROM PROJECT;
```

- Column col6 has a data type of DECIMAL(16,2) and contains the value 12301000123010.45. Because the data type is DECIMAL(16,2), this value will be interpreted as a timestamp duration (1230 years, 10 months, 00 days, 12 hours, 30 minutes, 10.45 seconds). The following statement returns the integer value 10450:

```
SELECT EXTRACT(MILLISECOND FROM col6) FROM PROJECT;
```

FIRST_DAY

The FIRST_DAY function returns a date or timestamp that represents the first day of the month of the argument.



The schema is SYSIBM.

expression
An expression that specifies the datetime value to compute the first day of the month. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

The result of the function is a timestamp with the same precision as *expression*, if *expression* is a timestamp. Otherwise, the result of the function is a date. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

1. Set the host variable FIRST_OF_MONTH with the first day of the current month.

```
SET :FIRST_OF_MONTH = FIRST_DAY(CURRENT_DATE)
```

   The host variable FIRST_OF_MONTH is set with the value representing the beginning of the current month. If the current day is 2000-02-10, then FIRST_OF_MONTH is set to 2000-02-01.

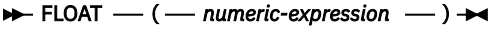
2. Set the host variable FIRST_OF_MONTH with the first day of the month in IBM European standard format for the given date.

```
SET :FIRST_OF_MONTH = CHAR(FIRST_DAY(DATE '1965-07-07'), EUR)
```

   The host variable FIRST_OF_MONTH is set with the value '01.07.1965'.

FLOAT

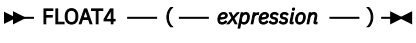
The FLOAT function returns a floating-point representation of a number. FLOAT is a synonym for DOUBLE.



The schema is SYSIBM.

FLOAT4

The FLOAT4 function returns a single-precision floating-point representation of either a number or a string representation of a number.

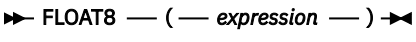


The schema is SYSIBM.

The FLOAT4 scalar function is a synonym for the REAL scalar function.

FLOAT8

The FLOAT8 function returns a double-precision floating-point representation of either a number or a string representation of a number.

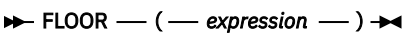


The schema is SYSIBM.

The FLOAT8 scalar function is a synonym for the DOUBLE scalar function.

FLOOR

Returns the largest integer value less than or equal to the argument.



The schema is SYSIBM. (The SYSFUN version of the FLOOR function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type.

The result of the function has the same data type and length attribute as the argument except that the scale is 0 if the argument is DECIMAL. For example, an argument with a data type of DECIMAL(5,5) returns DECIMAL(5,0).

The result can be null if the argument can be null or if the argument is not a decimal floating-point number and the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

Notes

- **Results involving DECFLOAT special values:** For decimal floating-point values, the special values are treated as follows:
  - FLOOR(NaN) returns NaN.
  - FLOOR(-NaN) returns -NaN.
  - FLOOR(Infinity) returns Infinity.
  - FLOOR(-Infinity) returns -Infinity.
  - FLOOR(sNaN) returns NaN and a warning.
  - FLOOR(-sNaN) returns -NaN and a warning.

Examples

- *Example 1*: Use the FLOOR function to truncate any digits to the right of the decimal point.

```
SELECT FLOOR(SALARY)
  FROM EMPLOYEE
```

- *Example 2*: Use the FLOOR function on both positive and negative numbers.

```
VALUES FLOOR(3.5), FLOOR(3.1),
  FLOOR(-3.1), FLOOR(-3.5)
```

This example returns 3., 3., -4., and -4., respectively.

FROM_UTC_TIMESTAMP

The FROM_UTC_TIMESTAMP scalar function returns a TIMESTAMP that is converted from Coordinated Universal Time to the time zone specified by the time zone string. FROM_UTC_TIMESTAMP is a statement deterministic function.

The schema is SYSIBM.

expression
An expression that specifies the timestamp that is in the Coordinated Universal Time time zone. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. If *expression* does not contain time information, a time of midnight (00.00.00) is used for the argument. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported through implicit casting. If expression is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

timezone-expression
An expression that specifies the time zone that the expression is to be adjusted to. The expression must return a value that is a built-in character string, numeric, or datetime data type. In a Unicode database, the expression can also be a graphic string data type. Numeric and datetime data types are supported through implicit casting. If the expression is not a VARCHAR, it is cast to VARCHAR before the function is evaluated. The expression must not be a FOR BIT DATA subtype (SQLSTATE 42815). *timezone-expression* must not be null if *expression* is not null (SQLSTATE 42815).

The value of the *timezone-expression* must be a time zone name from the Internet Assigned Numbers Authority (IANA) time zone database. The standard format for a time zone name in the IANA database is `Area/Location`, where:

- `Area` is the English name of a continent, ocean, or the special area 'Etc'.
- `Location` is the English name of a location within the area; usually a city, or small island.

Examples:

- "America/Toronto"
- "Asia/Sakhalin"
- "Etc/UTC" (which represents Coordinated Universal Time)

For complete details on the valid set of time zone names and the rules that are associated with those time zones, refer to the IANA time zone database. The database server uses version **2010c** of the IANA time zone database. Contact IBM support if a newer version of the IANA time zone database is required.

The result of the function is a timestamp with the same precision as *expression*, if *expression* is a timestamp. If expression is a DATE, the result of the function is a TIMESTAMP(0). Otherwise, the result of the function is a TIMESTAMP(6).

The result can be null; if the *expression* is null, the result is the null value. The *timezone-expression* cannot be null if a not-null value was supplied for the *expression* (SQLSTATE 42815).

The result is the *expression*, adjusted from the Coordinated Universal Time time zone to the time zone specified by the *timezone-expression*. If the *timezone-expression* returns a value that is not a time zone in the IANA time zone database, then the value of *expression* is returned without being adjusted.

The timestamp adjustment is done by first applying the raw offset from Coordinated Universal Time of the *timezone-expression*. If Daylight Saving Time is in effect at the adjusted timestamp for the time zone that is specified by the *timezone-expression*, then the Daylight Saving Time offset is also applied to the timestamp.

Time zones that use Daylight Saving Time have ambiguities at the transition dates. When a time zone changes from standard time to Daylight Saving Time, a range of time does not occur as it is skipped during the transition. When a time zone changes from Daylight Saving Time to standard time, a range of time occurs twice. Ambiguous timestamps are treated as if they occurred when standard time was in effect for the time zone.

Examples

1. Convert the Coordinated Universal Time timestamp '2011-12-25 09:00:00.123456' to the 'Asia/Tokyo' time zone. The following returns a TIMESTAMP with the value '2011-12-25 18:00:00.123456'.

```
FROM_UTC_TIMESTAMP(TIMESTAMP '2011-12-25 09:00:00.123456', 'Asia/Tokyo')
```

2. Convert the Coordinated Universal Time timestamp '2014-11-02 06:55:00' to the 'America/Toronto' time zone. The following returns a TIMESTAMP with the value '2014-11-02 01:55:00'.

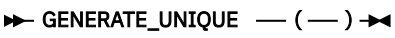
```
FROM_UTC_TIMESTAMP(TIMESTAMP'2014-11-02 06:55:00', 'America/Toronto')
```

3. Convert the Coordinated Universal Time timestamp '2015-03-02 06:05:00' to the 'America/Toronto' time zone. The following returns a TIMESTAMP with the value '2015-03-02 01:05:00'.

```
FROM_UTC_TIMESTAMP(TIMESTAMP'2015-03-02 06:05:00', 'America/Toronto')
```

GENERATE_UNIQUE

The GENERATE_UNIQUE function returns a bit data character string 13 bytes long (CHAR(13) FOR BIT DATA) that is unique compared to any other execution of the same function.



The schema is SYSIBM.

The system clock is used to generate the internal Universal Time, Coordinated (UTC) timestamp along with the database partition number on which the function executes. Adjustments that move the actual system clock backward could result in duplicate values.

The function is defined as non-deterministic.

There are no arguments to this function (the empty parentheses must be specified).

The result of the function is a unique value that includes the internal form of the Universal Time, Coordinated (UTC) and the database partition number where the function was processed. The result cannot be null.

The result of this function can be used to provide unique values in a table. Each successive value will be greater than the previous value, providing a sequence that can be used within a table. The value includes the database partition number where the function executed so that a table partitioned across multiple database partitions also has unique values in some sequence. The sequence is based on the time the function was executed.

This function differs from using the special register CURRENT TIMESTAMP in that a unique value is generated for each row of a multiple row insert statement, an insert statement with a fullselect, or an INSERT operation in a MERGE statement.

The timestamp value that is part of the result of this function can be determined using the TIMESTAMP scalar function with the result of GENERATE_UNIQUE as an argument.

Example

Create a table that includes a column that is unique for each row. Populate this column using the GENERATE_UNIQUE function. Notice that the UNIQUE_ID column has "FOR BIT DATA "specified to identify the column as a bit data character string.

```
CREATE TABLE EMP_UPDATE
  (UNIQUE_ID CHAR(13) FOR BIT DATA,
  EMPNO CHAR(6),
  TEXT VARCHAR(1000))
  INSERT INTO EMP_UPDATE
    VALUES (GENERATE_UNIQUE(), '000020', 'Update entry...'),
    (GENERATE_UNIQUE(), '000050', 'Update entry...')
```

This table will have a unique identifier for each row provided that the UNIQUE_ID column is always set using GENERATE_UNIQUE. This can be done by introducing a trigger on the table.

```
CREATE TRIGGER EMP_UPDATE_UNIQUE
  NO CASCADE BEFORE INSERT ON EMP_UPDATE
  REFERENCING NEW AS NEW_UPD
  FOR EACH ROW
  SNEW_UPD.UNIQUE_ID = GENERATE_UNIQUE()
```

With this trigger defined, the previous INSERT statement could be issued without the first column as follows.

```
INSERT INTO EMP_UPDATE (EMPNO, TEXT)
  VALUES ('000020', 'Update entry 1...'),
  ('000050', 'Update entry 2...')
```

The timestamp (in UTC) for when a row was added to EMP_UPDATE can be returned using:

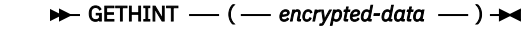
```
SELECT TIMESTAMP (UNIQUE_ID), EMPNO, TEXT
  FROM EMP_UPDATE
```

Therefore, there is no need to have a timestamp column in the table to record when a row is inserted.

GETHINT

The GETHINT function will return the password hint if one is found in the *encrypted-data*.

Important: The GETHINT function is deprecated and might not appear in future releases.



The schema is SYSIBM.

A password hint is a phrase that will help data owners remember passwords; for example, 'Ocean' as a hint to remember 'Pacific'. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

encrypted-data
An expression that returns a CHAR FOR BIT DATA or VARCHAR FOR BIT DATA value that is a complete, encrypted data string. The data string must have been encrypted using the ENCRYPT function (SQLSTATE 428FE).

The result of the function is VARCHAR(32 OCTETS). The result can be null; if the hint parameter was not added to the *encrypted-data* by the ENCRYPT function or the first argument is null, the result is the null value.

Example

In this example the hint 'Ocean' is stored to help the user remember the encryption password 'Pacific'.

```
INSERT INTO EMP (SSN) VALUES ENCRYPT('289-46-8832', 'Pacific','Ocean');
SELECT GETHINT(SSN)
  FROM EMP;
```

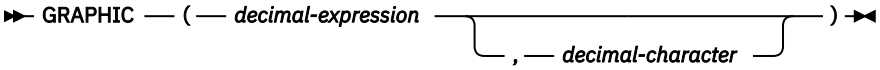
The value returned is 'Ocean'.

GRAPHIC

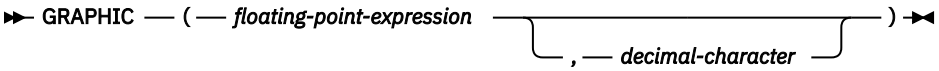
The GRAPHIC function returns a fixed-length graphic string representation of a value of a different data type.

Integer to graphic

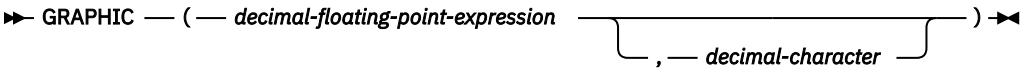
Decimal to graphic



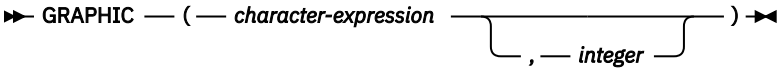
Floating-point to graphic



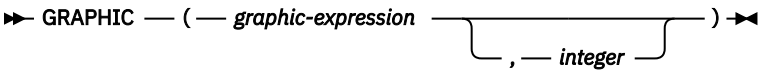
Decimal floating-point to graphic



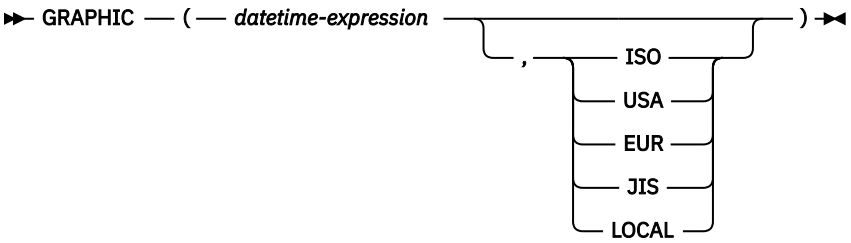
Character to graphic



Graphic to graphic



Datetime to graphic



Boolean to vargraphic

The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature.

Integer to graphic

integer-expression
An expression that returns a value that is of an integer data type (SMALLINT, INTEGER, or BIGINT).

The result is a fixed-length graphic string representation of *integer-expression* in the form of an SQL integer constant. The result consists of *n* double-byte characters, which represent the significant digits in the argument, and is preceded by a minus sign if the argument is negative. The result is left-aligned.

- If the first argument is a small integer, the length of the result is 6.
- If the first argument is a large integer, the length of the result is 11.
- If the first argument is a big integer, the length of the result is 20.

If the number of double-byte characters in the result is less than the defined length of the result, the result is padded on the right with blanks.

The code page of the result is the DBCS code page of the section.

Decimal to graphic

decimal-expression
An expression that returns a value that is a decimal data type. The DECIMAL scalar function can be used to change the precision and scale.

decimal-character
Specifies the double-byte character constant that is used to delimit the decimal digits in the result graphic string. The double-byte character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a fixed-length graphic string representation of *decimal-expression* in the form of an SQL decimal constant. The length of the result is 2+*p*, where *p* is the precision of *decimal-expression*. Leading zeros are not included. Trailing zeros are included. If *decimal-expression* is negative, the first double-byte character of the result is a minus sign; otherwise, the first double-byte character is a digit or the decimal character. If the scale of *decimal-expression* is zero, the decimal character is not returned. If the number of double-byte characters in the result is less than the defined length of the result, the result is padded on the right with blanks.

The code page of the result is the DBCS code page of the section.

Floating-point to graphic

floating-point-expression
An expression that returns a value that is a floating-point data type (DOUBLE or REAL).

decimal-character
Specifies the double-byte character constant that is used to delimit the decimal digits in the result graphic string. The double-byte character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a fixed-length graphic string representation of *floating-point-expression* in the form of an SQL floating-point constant. The length of the result is 24. The result is the smallest number of double-byte characters that can represent the value of *floating-point-expression* such that the mantissa consists of a single digit other than zero followed by a period and a sequence of digits. If *floating-point-expression* is negative, the first double-byte character of the result is a minus sign; otherwise, the first double-byte character is a digit. If *floating-point-expression* is zero, the result is

0E0. If the number of double-byte characters in the result is less than 24, the result is padded on the right with blanks.

The code page of the result is the DBCS code page of the section.

Decimal floating-point to graphic

decimal-floating-point-expression
An expression that returns a value that is a decimal floating-point data type (DECFLOAT).

decimal-character
Specifies the double-byte character constant that is used to delimit the decimal digits in the result graphic string. The double-byte character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a fixed-length graphic string representation of *decimal-floating-point-expression* in the form of an SQL decimal floating-point constant. The length attribute of the result is 42. The result is the smallest number of double-byte characters that can represent the value of *decimal-floating-point-expression*. If *decimal-floating-point-expression* is negative, the first double-byte character of the result is a minus sign; otherwise, the first double-byte character is a digit. If *decimal-floating-point-expression* is zero, the result is 0.

If the value of *decimal-floating-point-expression* is the special value Infinity, sNaN, or NaN, the strings G'INFINITY', G'SNAN', and G'NAN', respectively, are returned. If the special value is negative, the first double-byte character of the result is a minus sign. The decimal floating-point special value sNaN does not result in a warning when converted to a string. If the number of double-byte characters in the result is less than 42, the result is padded on the right with blanks.

The code page of the result is the DBCS code page of the section.

Character to graphic
In Unicode databases:

character-expression
An expression that returns a value that is a built-in character string data type. The expression must not be a FOR BIT DATA subtype (SQLSTATE 42846).

integer
An integer constant that specifies the length attribute for the resulting fixed-length graphic string. The value must be between 0 and the maximum length for the GRAPHIC data type in the string units of the result.

If the second argument is not specified:

- If the *character-expression* is the empty string constant, the length attribute of the result is 0.
- Otherwise, the length attribute of the result is the lower of the following values:
  - The maximum length for the GRAPHIC data type in the string units of the result
  - The length attribute of the first argument

The result is a fixed-length graphic string that is converted from *character-expression*. The length attribute of the result is determined by the value of *integer*.

The actual length of the result is the same as the length attribute of the result.

If the length of *character-expression* that is converted to a graphic string is less than the length attribute of the result, the result is padded with blanks up to the length attribute of the result.

If the length of *character-expression* that is converted to a graphic string is greater than the length attribute of the result, several scenarios exist:

- If the string unit of the result is CODEUNITS32, truncation is performed. If only blank characters are truncated and *character-expression* is CHAR or VARCHAR, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004).
- If *integer* is specified, truncation is performed. If only blank characters are truncated and *character-expression* is CHAR or VARCHAR, no warning is returned. Otherwise, a warning is returned

(SQLSTATE 01004). When the output string is truncated, such that the last character is a high surrogate, that surrogate is converted to the blank character (X'0020'). Do not rely on this behavior because it might change in a future release.

- If *integer* is not specified, an error is returned (SQLSTATE 22001).

For details about the conversion process, see "VARGRAPHIC " on page 573.

In non-Unicode databases:

character-expression
An expression that returns a value that is a built-in CHAR or VARCHAR data type.

The result is a fixed-length graphic string that is converted from *character-expression*. The length attribute of the result is the minimum of the length attribute of *character-expression* and the maximum length for the GRAPHIC data type.

The actual length of the result is the same as the length attribute of the result. If the length of *character-expression* that is converted to a graphic string is less than the length attribute of the result, the result is padded with blanks up to the length attribute of the result. If the length of *character-expression* that is converted to a graphic string is greater than the length attribute of the result, an error is returned (SQLSTATE 22001).

For details about the conversion process, see "VARGRAPHIC " on page 573.

Graphic to graphic

graphic-expression
An expression that returns a built-in value that is a graphic string data type.

integer
An integer constant that specifies the length attribute for the resulting fixed-length graphic string. The value must be between 0 and the maximum length for the GRAPHIC data type in the string units of the result.

If the second argument is not specified:

- If the *graphic-expression* is the empty string constant, the length attribute of the result is 0.
- Otherwise, the length attribute of the result is the lower of the following values:
  - The maximum length for the GRAPHIC data type in the string units of the result
  - The length attribute of the first argument

The result is a fixed-length graphic string. The length attribute of the result is determined by the value of *integer*.

The actual length of the result is the same as the length attribute of the result.

If the length of *graphic-expression* is less than the length attribute of the result, the result is padded with blanks up to the length attribute of the result.

If the length of *graphic-expression* is greater than the length attribute of the result, several scenarios exist:

- If the string unit of the result is CODEUNITS32, truncation is performed. If only blank characters are truncated and *graphic-expression* is GRAPHIC or VARGRAPHIC, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004).
- If *integer* is specified, truncation is performed. If only blank characters are truncated and *graphic-expression* is GRAPHIC or VARGRAPHIC, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004). In a Unicode database, when the output string is truncated, such that the last character is a high surrogate, that surrogate is converted to the blank character (X'0020'). Do not rely on this behavior because it might change in a future release.
- If *integer* is not specified and *graphic-expression* is VARGRAPHIC, truncation behavior is as follows:
  - If only blank characters must be truncated, truncation is performed with no warning returned.
  - If non-blank characters must be truncated, an error is returned (SQLSTATE 22001).

- If *integer* is not specified and *graphic-expression* is DBCLOB, an error is returned (SQLSTATE 22001).

Datetime to graphic

datetime-expression
An expression that is of one of the following data types:

DATE
The result is the graphic string representation of the date in the format specified by the second argument. The length of the result is 10. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

TIME
The result is the graphic string representation of the time in the format specified by the second argument. The length of the result is 8. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

TIMESTAMP
The result is the graphic string representation of the timestamp. If the data type of *datetime-expression* is TIMESTAMP(0), the length of the result is 19. If the data type of *datetime-expression* is TIMESTAMP(*n*), where *n* is between 1 and 12, the length of the result is 20+*n*. Otherwise, the length of the result is 26.

The code page of the string is the code page of the section.

Boolean to graphic

boolean-expression
An expression that returns a Boolean value (TRUE or FALSE). The result is either 'TRUE ' (note the blank after the E) or 'FALSE'.

Result

The GRAPHIC function returns a fixed-length graphic string representation of:

- An integer number (Unicode database only), if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number (Unicode database only), if the first argument is a decimal number
- A double-precision floating-point number (Unicode database only), if the first argument is a DOUBLE or REAL
- A decimal floating-point number (Unicode database only), if the argument is a decimal floating-point number (DECFLOAT)
- A character string, if the first argument is any type of character string
- A graphic string, if the first argument is any type of graphic string
- A datetime value (Unicode database only), if the first argument is a DATE, TIME, or TIMESTAMP
- A Boolean value (TRUE or FALSE)

In a non-Unicode database, the string units of the result is double bytes. Otherwise, the string units of the result are determined by the data type of the first argument.

- CODEUNITS16, if the first argument is character string or a graphic string with string units of OCTETS or CODEUNITS16.
- CODEUNITS32, if the first argument is character string or a graphic string with string units of CODEUNITS32.
- Determined by the default string unit of the environment, if the first argument is not a character string or a graphic string.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Notes

- **Increasing portability of applications:** If the first argument is numeric, or if the first argument is a string and the length argument is specified, use the "CAST specification" on page 152 instead of this function to increase the portability of your applications.

Examples

- *Example 1*: The EDLEVEL column is defined as SMALLINT. The following returns the value as a fixed-length graphic string.

```
SELECT GRAPHIC(EDLEVEL)
  FROM EMPLOYEE
  WHERE LASTNAME = 'HAAS'
```

  Results in the value G'18 '.

- *Example 2*: The SALARY and COMM columns are defined as DECIMAL with a precision of 9 and a scale of 2. Return the total income for employee Haas using the comma decimal character.

```
SELECT GRAPHIC(SALARY + COMM, ',')
  FROM EMPLOYEE
  WHERE LASTNAME = 'HAAS'
```

  Results in the value G'56970,00 '.

- *Example 3*: The following statement returns a string of data type GRAPHIC with the value 'TRUE '.

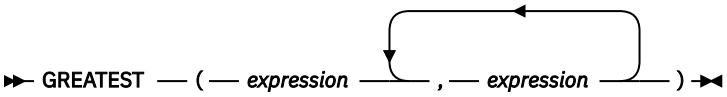
```
values GRAPHIC(3=3)
```

- *Example 4*: The following statement returns a string of data type GRAPHIC with the value 'FALSE'.

```
values GRAPHIC(3>3)
```

GREATEST

The GREATEST function returns the maximum value in a set of values.



The schema is SYSIBM.

expression

An expression that returns a value of any built-in data type or user-defined data type that is comparable with the data type of the other argument. The data type cannot be a LOB, distinct type based on a LOB, XML, array, cursor, row, or structured type.

Result

The result of the function is the largest argument value. The result can be null if at least one argument can be null; the result is the null value if any argument is null.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by the data types of all the arguments as explained in "Rules for result data types".

Notes

- The GREATEST scalar function is a synonym for the MAX scalar function.

- The GREATEST function cannot be used as a source function when creating a user-defined function. Because this function accepts any comparable data types as arguments, it is not necessary to create additional signatures to support user-defined data types.

Example

Assume that table T1 contains three columns C1, C2, and C3 with values 1, 7, and 4, respectively. The query:

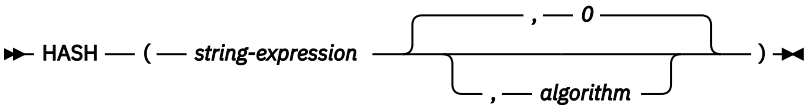
```
SELECT GREATEST (C1, C2, C3) FROM T1
```

returns 7.

If column C3 has a value of null instead of 4, the same query returns the null value.

HASH

The HASH function returns a 128-bit, 160-bit, 256-bit or 512-bit hash of the input data, depending on the algorithm selected, and is intended for cryptographic purposes.



The schema is SYSIBM.

string-expression
An expression that represents the string value to be hashed. This expression must return a built-in character string, graphic string, binary string, numeric value, Boolean value, or datetime value. If the value is not a character, graphic, or binary string, it is implicitly cast to VARCHAR before the function is evaluated.

algorithm
An expression that returns a value that indicates which algorithm is to be used for hashing. The expression must return a value that has a built-in numeric, a CHAR, or a VARCHAR data type. In a Unicode database, the expression can also return a GRAPHIC or VARGRAPHIC data type. If the value is not an INTEGER, it is cast to INTEGER before the function is evaluated. If no algorithm is specified, the default algorithm value of 0 is used.

Table 60 on page 359 shows the algorithm used, the result size, and the number of return values for each algorithm value.

Table 60. Resulting size from each algorithm

Algorithm value	Algorithm	Result size	Number of return values
0	MD5	128 bit	2^{128}
1	SHA1	160 bit	2^{160}
2	SHA2_256	256 bit	2^{256}
3	SHA2_512	512 bit	2^{512}

Note that security flaws have been identified in both the SHA1 and MD5 algorithms. You can find acceptable hash algorithms in applicable compliance documentation, such as National Institute of Standards and Technology (NIST) Special Publication 800-131A.

Result

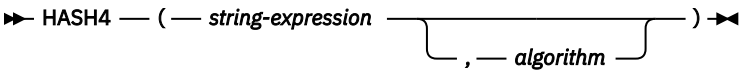
The data type of the result is VARBINARY. If any argument can be null, the result can be null. If any argument is null, the result is the null value.

Example

```
values  hash('Charlie at IBM', 1)
result is x'D6E42303462491FC696EAC53C1B086A5034735A7'
```

HASH4

The HASH4 function returns the 32-bit checksum hash of the input data. The function provides 2^{32} distinct return values and is intended for data retrieval (lookups).



The schema is SYSIBM.

string-expression
An expression that represents the string value to be hashed. This expression must return a built-in character string, graphic string, binary string, numeric value, Boolean value, or datetime value. If the value is not a character, graphic, or binary string, it is implicitly cast to VARCHAR before the function is evaluated.

algorithm
An expression that specifies the algorithm to use for hashing. The expression must return a value that is a built-in numeric, a CHAR, or a VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported using implicit casting. If the expression is not a INTEGER, it is cast to INTEGER before evaluating the function.

The algorithm value can be either `0` or `1`. `0` is the default value and indicates the Adler algorithm. `1` indicates the CRC32 algorithm. The Adler algorithm provides a faster checksum hash; however, it has poor coverage when the messages are less than a few hundred bytes (poor coverage means that two different integers hash to the same value, referred to as a "collision"). In this case, use the CRC32 algorithm, or switch to hash8 instead.

Result

The data type of the result is INTEGER. If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

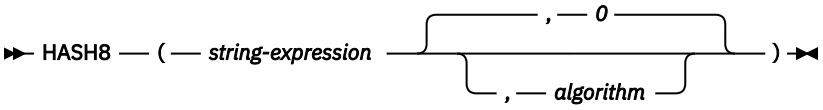
Example

The following example gives a hashed value of a string of text:

```
values hash4('Charlie at IBM', 0)
The result is 622396582
```

HASH8

The HASH8 function returns the 64-bit hash of an input string. The function provides 2^{64} distinct return values and is intended for data retrieval (that is, lookups). The result for a particular input string differs depending on the endianness (big-endian or little-endian) of your system.



The schema is SYSIBM.

string-expression
An expression that represents the string value to be hashed. This expression must return a built-in character string, graphic string, binary string, numeric value, Boolean value, or datetime value. If the value is not a character, graphic, or binary string, it is implicitly cast to VARCHAR before the function is evaluated.

algorithm
An expression that returns a value that indicates which algorithm is to be used for hashing. The expression must return a value that has a built-in numeric, a CHAR, or a VARCHAR data type. In a Unicode database, the expression can also return a GRAPHIC or VARGRAPHIC data type. If the value is not an INTEGER, it is cast to INTEGER before the function is evaluated. The returned value must be 0, which indicates the Jenkins algorithm.

Result

The data type of the result is BIGINT. If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

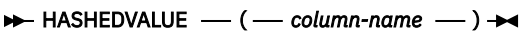
Example

```
values hash8('Charlie at IBM', 0)
```

- On a little-endian system, the result is 4570902652830829618.
- On a big-endian system, the result is 7187665777530874019.

HASHEDVALUE

The HASHEDVALUE function returns the distribution map index of the row obtained by applying the partitioning function on the distribution key value of the row.



The schema is SYSIBM.

column-name
The qualified or unqualified name of a column in a table. The column can have any data type.

If the column is a column of a view, the expression in the view for the column must reference a column of the underlying base table and the view must be deletable. A nested or common table expression follows the same rules as a view.

An example application for this function is in a SELECT clause, where it returns the distribution map index for each row of the table that was used to form the result of the SELECT statement.

The distribution map index returned on transition variables and tables is derived from the current transition values of the distribution key columns. For example, in a before insert trigger, the function will return the projected distribution map index given the current values of the new transition variables. However, the values of the distribution key columns may be modified by a subsequent before insert

trigger. Thus, the final distribution map index of the row when it is inserted into the database may differ from the projected value.

The specific row (and table) for which the distribution map index is returned by the HASHEDVALUE function is determined from the context of the SQL statement that uses the function.

Result

The data type of the result is INTEGER in the range 0 to 32767. For a table with no distribution key, the result is always 0. A null value is never returned. Since row-level information is returned, the results are the same, regardless of which column is specified for the table.

Notes

- The HASHEDVALUE function cannot be used on replicated tables, within check constraints, or in the definition of generated columns (SQLSTATE 42881).
- The HASHEDVALUE function cannot be used as a source function when creating a user-defined function. Because it accepts any data type as an argument, it is not necessary to create additional signatures to support user-defined distinct types.
- The HASHEDVALUE function cannot be used as part of an expression-based key in a CREATE INDEX statement.
- ***Syntax alternatives:*** For compatibility with previous versions of Db2 products, the function name PARTITION is a synonym for HASHEDVALUE.

Examples

- *Example 1*: List the employee numbers (EMPNO) from the EMPLOYEE table for all rows with a distribution map index of 100.

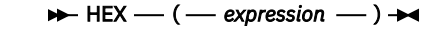
```
SELECT EMPNO FROM EMPLOYEE
  WHERE HASHEDVALUE(PHONENO) = 100
```

- *Example 2*: Log the employee number and the projected distribution map index of the new row into a table called EMPINSERTLOG2 for any insertion of employees by creating a before trigger on the table EMPLOYEE.

```
CREATE TRIGGER EMPINSLOGTRIG2
  BEFORE INSERT ON EMPLOYEE
  REFERENCING NEW AW NEWTABLE
  FOR EACH ROW
  INSERT INTO EMPINSERTLOG2
    VALUES(NEWTABLE.EMPNO, HASHEDVALUE(NEWTABLE.EMPNO))
```

HEX

The HEX function returns a hexadecimal representation of a value as a character string.



The schema is SYSIBM.

expression
An expression that returns a value of any built-in data type that is not XML, with a maximum length of 16 336 bytes.

The result of the function is a character string with string units of OCTETS. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The code page is the section code page.

The result is a string of hexadecimal digits. The first two represent the first byte of the argument, the next two represent the second byte of the argument, and so forth. If the argument is a datetime value or

a numeric value the result is the hexadecimal representation of the internal form of the argument. The hexadecimal representation that is returned may be different depending on the application server where the function is executed. Cases where differences would be evident include:

- Character string arguments when the HEX function is performed on an ASCII client with an EBCDIC server or on an EBCDIC client with an ASCII server.
- Numeric arguments (in some cases) when the HEX function is performed where client and server systems have different byte orderings for numeric values.

The type and length of the result vary based on the type, length, and string units of the character and graphic string arguments.

Table 61. Data type of the result as a function of the data types of the argument data type and the length attribute

Argument data type[1]	**Length attribute**[2]	**Result data type**
CHAR(A) or BINARY(A)	A<128	CHAR(A*2)
CHAR(A) or BINARY(A)	A>127	VARCHAR(A*2)
VARCHAR(A), VARBINARY(A), CLOB(A), or BLOB(A)	A<16337	VARCHAR(A*2)
GRAPHIC(A)	A<64	CHAR(A*2*2)
GRAPHIC(A)	A>63	VARCHAR(A*2*2)
VARGRAPHIC(A) or DBCLOB(A)	A<8169	VARCHAR(A*2*2)
CHAR(A CODEUNITS32)	A<64	VARCHAR(A*4*2)
VARCHAR(A CODEUNITS32) or CLOB(A CODEUNITS32)	A<4085	VARCHAR(A*4*2)
GRAPHIC(A CODEUNITS32)	A<64	VARCHAR(A*2*2*2)
VARGRAPHIC(A CODEUNITS32) or DBCLOB(A CODEUNITS32)	A<4085	VARCHAR(A*2*2*2)

1. If string units are not specified, then the string units for the data type are not CODEUNITS32.
2. The maximum length attributes reflect a data type limit or the limit of 16336 bytes for the input argument.

Examples

Assume the use of a database application server on AIX® or Linux® for the following examples.

- *Example 1*: Using the DEPARTMENT table set the host variable HEX_MGRNO (char(12)) to the hexadecimal representation of the manager number (MGRNO) for the "PLANNING" department (DEPTNAME).

```
SELECT HEX(MGRNO)
  INTO :HEX_MGRNO
  FROM DEPARTMENT
  WHERE DEPTNAME = 'PLANNING'
```

  HEX_MGRNO will be set to "303030303230" when using the sample table (character value is "000020").
- *Example 2*: Suppose COL_1 is a column with a data type of char(1) and a value of "B". The hexadecimal representation of the letter "B" is X'42'. HEX(COL_1) returns a two byte long string "42".
- *Example 3*: Suppose COL_3 is a column with a data type of decimal(6,2) and a value of 40.1. An eight byte long string "0004010C" is the result of applying the HEX function to the internal representation of the decimal value, 40.1.

HEXTORAW

The HEXTORAW function returns a bit string representation of a hexadecimal character string.

The schema is SYSIBM.

In a Unicode database, if a supplied argument is a graphic string, it is converted to a character string before the function is executed.

character-expression
An expression that returns a value that is a built-in character string that is not a CLOB (SQLSTATE 42815). The length must be an even number of characters from the ranges '0' to '9', 'a' to 'f', and 'A' to 'F' (SQLSTATE 42815).

The result of the function is a VARBINARY. The length attribute of the result is half the length attribute of *character-expression* and the actual length is half the length of the actual length of *character-expression*. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Example

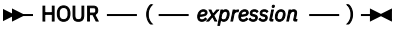
Represent a string of hexadecimal characters in binary form.

```
HEXTORAW('ef01abC9')
```

The result is a VARBINARY(4) data type with a value of BX'EF01ABC9'.

HOUR

The HOUR function returns the hour part of a value.



The schema is SYSIBM.

expression
An expression that returns a value of one of the following built-in data types: a DATE, a TIME, a TIMESTAMP, a character string, or an exact numeric data type.

If *expression* is a character string, it must not be a CLOB and its value must be a valid string representation of a datetime value. For the valid formats of string representations of datetime values, see "String representations of datetime values" in "Datetime values".

If *expression* is an exact numeric value, it must be a time duration or timestamp duration. For information about valid time durations and timestamp durations, see "Datetime operands and durations".

Only Unicode databases support an expression that is a valid graphic string representation of a datetime value that is not a DBCLOB. The graphic string is converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a TIME, TIMESTAMP or valid string representation of a time or timestamp:
  - The result is the hour part of the value, which is an integer between 0 and 24.
- If the argument is a DATE or valid string representation of a date:
  - The result is 0.

- If the argument is a time duration or timestamp duration:
  - The result is the hour part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Example

Using the CL_SCHED sample table, select all the classes that start in the afternoon.

```
SELECT * FROM CL_SCHED
  WHERE HOUR(STARTING) BETWEEN 12 AND 17
```

HOURS_BETWEEN

The HOURS_BETWEEN function returns the number of full hours between the specified arguments.

The schema is SYSIBM.

expression1
An expression that specifies the first datetime value to compute the number of full hours between two datetime values. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression1* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

expression2
An expression that specifies the second datetime value to compute the number of full hours between two datetime values. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression2* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

If there is less than a full hour between *expression1* and *expression2*, the result is zero. If *expression1* is later than *expression2*, the result is positive. If *expression1* is earlier than *expression2*, the result is negative. In NPS compatibility mode, this function always returns a positive number. If *expression1* or *expression2* contains time information, this information is also used to determine the number of full hours. If *expression1* or *expression2* does not contain time information, a time of midnight (00.00.00) is used for the argument that is missing time information.

The result of the function is an INTEGER. If either argument can be null, the result can be null. If either argument is null, the result is the null value.

Examples

1. Set the host variable NUM_HOURS to the number of full hours between 2012-03-01-01.00.00 and 2012-02-28-00.00.00.

   ```
   SET :NUM_HOURS = HOURS_BETWEEN(TIMESTAMP '2012-03-01-01.00.00',
                                  TIMESTAMP '2012-02-28-00.00.00')
   ```

   The host variable NUM_HOURS is set to 49 because an additional day is incurred for February 29, 2012.

2. Set the host variable NUM_HOURS to the number of full hours between 2013-09-11-23.59.59 and 2013-09-01-00.00.00.

   ```
   SET :NUM_HOURS = HOURS_BETWEEN(TIMESTAMP '2013-09-11-23.59.59',
                                  TIMESTAMP '2013-09-01-00.00.00')
   ```

The host variable NUM_HOURS is set to 263 because there is 1 second less than a full 264 hours between the arguments. It is positive because the first argument is later than the second argument.

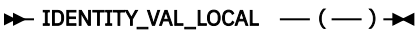
3. Set the host variable NUM_HOURS to the number of full hours between 2013-09-01-00.00.00 and 2013-09-11-23.59.59.

```
SET :NUM_HOURS = HOURS_BETWEEN(TIMESTAMP '2013-09-01-00.00.00',
                               TIMESTAMP '2013-09-11-23.59.59')
```

The host variable NUM_HOURS is set to -263 because there is 1 second less than a full 264 hours between the arguments. It is negative because the first argument is earlier than the second argument.

IDENTITY_VAL_LOCAL

The IDENTITY_VAL_LOCAL function is a non-deterministic function that returns the most recently assigned value for an identity column, where the assignment occurred as a result of a single INSERT statement using a VALUES clause.



The schema is SYSIBM.

The function has no input parameters.

The result is a DECIMAL(31,0), regardless of the actual data type of the corresponding identity column.

The value returned by the function is the value assigned to the identity column of the table identified in the most recent single row insert operation. The INSERT statement must contain a VALUES clause on a table containing an identity column. The INSERT statement must also be issued at the same level; that is, the value must be available locally at the level it was assigned, until it is replaced by the next assigned value. (A new level is initiated each time a trigger or routine is invoked.)

The assigned value is either a value supplied by the user (if the identity column is defined as GENERATED BY DEFAULT), or an identity value generated by the database manager.

It is recommended that a SELECT FROM data-change-table-reference statement be used to obtain the assigned value for an identity column. See "table-reference" in "subselect" for more information.

The function returns a null value if a single row INSERT statement with a VALUES clause has not been issued at the current processing level against a table containing an identity column.

The result of the function is not affected by the following operations:

- A single row INSERT statement with a VALUES clause for a table without an identity column
- A multiple row INSERT statement with a VALUES clause
- An INSERT statement with a fullselect
- A ROLLBACK TO SAVEPOINT statement

Notes

- Expressions in the VALUES clause of an INSERT statement are evaluated before the assignments for the target columns of the insert operation. Thus, an invocation of an IDENTITY_VAL_LOCAL function inside the VALUES clause of an INSERT statement will use the most recently assigned value for an identity column from a previous insert operation. The function returns the null value if no previous single row INSERT statement with a VALUES clause for a table containing an identity column has been executed within the same level as the IDENTITY_VAL_LOCAL function.
- The identity column value of the table for which the trigger is defined can be determined within a trigger by referencing the trigger transition variable for the identity column.
- The result of invoking the IDENTITY_VAL_LOCAL function from within the trigger condition of an insert trigger is a null value.

- It is possible that multiple before or after insert triggers exist for a table. In this case, each trigger is processed separately, and identity values assigned by one triggered action are not available to other triggered actions using the IDENTITY_VAL_LOCAL function. This is true even though the multiple triggered actions are conceptually defined at the same level.
- It is not generally recommended to use the IDENTITY_VAL_LOCAL function in the body of a before insert trigger. The result of invoking the IDENTITY_VAL_LOCAL function from within the triggered action of a before insert trigger is the null value. The value for the identity column of the table for which the trigger is defined cannot be obtained by invoking the IDENTITY_VAL_LOCAL function within the triggered action of a before insert trigger. However, the value for the identity column can be obtained in the triggered action by referencing the trigger transition variable for the identity column.
- The result of invoking the IDENTITY_VAL_LOCAL function from within the triggered action of an after insert trigger is the value assigned to an identity column of the table identified in the most recent single row insert operation invoked in the same triggered action that had a VALUES clause for a table containing an identity column. (This applies to both FOR EACH ROW and FOR EACH STATEMENT after insert triggers.) If a single row INSERT statement with a VALUES clause for a table containing an identity column was not executed within the same triggered action, before the invocation of the IDENTITY_VAL_LOCAL function, the function returns a null value.
- Because IDENTITY_VAL_LOCAL is a non-deterministic function, the result of invoking this function within the SELECT statement of a cursor can vary for each FETCH statement.
- The assigned value is the value actually assigned to the identity column (that is, the value that would be returned on a subsequent SELECT statement). This value is not necessarily the value provided in the VALUES clause of the INSERT statement, or a value generated by the database manager. The assigned value could be a value specified in a SET transition variable statement, within the body of a before insert trigger, for a trigger transition variable associated with the identity column.
- ***Scope of IDENTITY_VAL_LOCAL***: The IDENTITY_VAL_LOCAL value persists until the next insert in the current session into a table that has an identity column defined on it, or the application session ends. The value is unaffected by COMMIT or ROLLBACK statements. The IDENTITY_VAL_LOCAL value cannot be directly set and is a result of inserting a row into a table.

  A technique commonly used, especially for performance, is for an application or product to manage a set of connections and route transactions to an arbitrary connection. In these situations, the availability of the IDENTITY_VAL_LOCAL value should be relied on only until the end of the transaction. Examples of where this type of situation can occur include applications that use XA protocols, use connection pooling, use the connection concentrator, and use HADR to achieve failover.
- The value returned by the function following a failed single row INSERT statement with a VALUES clause into a table with an identity column is unpredictable. It could be the value that would have been returned from the function had it been invoked before the failed insert operation, or it could be the value that would have been assigned had the insert operation succeeded. The actual value returned depends on the point of failure, and is therefore unpredictable.

Examples

- *Example 1:* Create two tables, T1 and T2, each with an identity column named C1. Start the identity sequence for table T2 at 10. Insert some values for C2 into T1.

```
CREATE TABLE T1
  (C1 INTEGER GENERATED ALWAYS AS IDENTITY,
   C2 INTEGER)

CREATE TABLE T2
  (C1 DECIMAL(15,0) GENERATED BY DEFAULT AS IDENTITY (START WITH 10),
   C2 INTEGER)

INSERT INTO T1 (C2) VALUES (5)

INSERT INTO T1 (C2) VALUES (6)

SELECT * FROM T1
```

  This query returns:

```
C1          C2
----------- -----------
          1           5
          2           6
```

Insert a single row into table T2, where column C2 gets its value from the IDENTITY_VAL_LOCAL function.

```
   INSERT INTO T2 (C2) VALUES (IDENTITY_VAL_LOCAL())

   SELECT * FROM T2
```

This query returns:

```
C1                C2
----------------- -----------
              10.           2
```

- *Example 2:* In a nested environment involving a trigger, use the IDENTITY_VAL_LOCAL function to retrieve the identity value assigned at a particular level, even though there might have been identity values assigned at lower levels. Assume that there are three tables, EMPLOYEE, EMP_ACT, and ACCT_LOG. There is an after insert trigger defined on EMPLOYEE that results in additional inserts into the EMP_ACT and ACCT_LOG tables.

```
   CREATE TABLE EMPLOYEE
     (EMPNO SMALLINT GENERATED ALWAYS AS IDENTITY (START WITH 1000),
      NAME CHAR(30),
      SALARY DECIMAL(5,2),
      DEPTNO SMALLINT)

   CREATE TABLE EMP_ACT
     (ACNT_NUM SMALLINT GENERATED ALWAYS AS IDENTITY (START WITH 1),
      EMPNO SMALLINT)

   CREATE TABLE ACCT_LOG
     (ID SMALLINT GENERATED ALWAYS AS IDENTITY (START WITH 100),
      ACNT_NUM SMALLINT,
      EMPNO SMALLINT)

   CREATE TRIGGER NEW_HIRE
     AFTER INSERT ON EMPLOYEE
     REFERENCING NEW AS NEW_EMP
     FOR EACH ROW
     BEGIN ATOMIC
       INSERT INTO EMP_ACT (EMPNO) VALUES (NEW_EMP.EMPNO);
       INSERT INTO ACCT_LOG (ACNT_NUM, EMPNO)
         VALUES (IDENTITY_VAL_LOCAL(), NEW_EMP.EMPNO);
     END
```

The first triggered insert operation inserts a row into the EMP_ACT table. The statement uses a trigger transition variable for the EMPNO column of the EMPLOYEE table to indicate that the identity value for the EMPNO column of the EMPLOYEE table is to be copied to the EMPNO column of the EMP_ACT table. The IDENTITY_VAL_LOCAL function could not be used to obtain the value assigned to the EMPNO column of the EMPLOYEE table, because an INSERT statement has not been issued at this level of the nesting. If the IDENTITY_VAL_LOCAL function were invoked in the VALUES clause of the INSERT statement for the EMP_ACT table, it would return a null value. The insert operation against the EMP_ACT table also results in the generation of a new identity value for the ACNT_NUM column.

The second triggered insert operation inserts a row into the ACCT_LOG table. The statement invokes the IDENTITY_VAL_LOCAL function to indicate that the identity value assigned to the ACNT_NUM column of the EMP_ACT table in the previous insert operation in the triggered action is to be copied to the ACNT_NUM column of the ACCT_LOG table. The EMPNO column is assigned the same value as the EMPNO column of the EMPLOYEE table.

After the following INSERT statement and all of the triggered actions have been processed:

```
   INSERT INTO EMPLOYEE (NAME, SALARY, DEPTNO)
     VALUES ('Rupert', 989.99, 50)
```

the contents of the three tables are as follows:

```
   SELECT EMPNO, SUBSTR(NAME,1,10) AS NAME, SALARY, DEPTNO
     FROM EMPLOYEE

EMPNO  NAME       SALARY  DEPTNO
------ ---------- ------- ------
  1000 Rupert      989.99     50

   SELECT ACNT_NUM, EMPNO
     FROM EMP_ACT

ACNT_NUM EMPNO
-------- ------
       1   1000

   SELECT * FROM ACCT_LOG

ID     ACNT_NUM EMPNO
------ -------- ------
   100        1   1000
```

The result of the IDENTITY_VAL_LOCAL function is the most recently assigned value for an identity column at the same nesting level. After processing the original INSERT statement and all of the triggered actions, the IDENTITY_VAL_LOCAL function returns a value of 1000, because this is the value that was assigned to the EMPNO column of the EMPLOYEE table.

IFNULL

The IFNULL function returns the first non-null expression in a list of two expressions.

The schema is SYSIBM.

The IFNULL function is identical to the “COALESCE ” on page 308, except that IFNULL is limited to two arguments.

INITCAP

The INITCAP function returns a string with the first character of each *word* converted to uppercase, using the UPPER function semantics, and the other characters converted to lowercase, using the LOWER function semantics.

The schema is SYSIBM.

A *word* is delimited by any of the following characters:

Table 62. Word delimiter characters

Character or range of characters	Unicode code points or range of Unicode code points
(blank)	U+0020
! " # $ % & ' ( ) * + , -. /	U+0021 to U+002F
: ; < = > ? @	U+003A to U+0040
[\ ] ^ _ `	U+005B to U+0060
{ | } ~	U+007B to U+007E

Table 62. Word delimiter characters (continued)	
Character or range of characters	**Unicode code points or range of Unicode code points**
Control characters, including the following SQL control characters: • tab • new line • form feed • carriage return • line feed	U+0009, U+000A, U+000B, U+000C, U+000D, U+0085

Note: A character listed in the preceding table might not have an allocated code point in a particular database code page.

string-expression
An expression that returns a CHAR or VARCHAR data type. In a Unicode database, the expression can return a GRAPHIC or VARGRAPHIC data type.

The data type of the result depends on the data type of *string-expression*, as described in the following table:

Table 63. Data type of string-expression compared to the data type of the result	
Data type of *string-expression*	**Data type of the result**
CHAR or VARCHAR	VARCHAR
GRAPHIC or VARGRAPHIC	VARGRAPHIC

The length attribute of the result is the same as the length attribute of *string-expression*.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

- *Example 1*: Input the string "a prospective book title" to return the string "A Prospective Book Title".

```
VALUES INITCAP ('a prospective book title')
1
------------------------
A Prospective Book Title
```

- *Example 2*: Input the string "YOUR NAME" to return the string "Your Name".

```
VALUES INITCAP ('YOUR NAME')
1
---------
Your Name
```

- *Example 3*: Input the string "my_résumé" to return the string "My_Résumé".

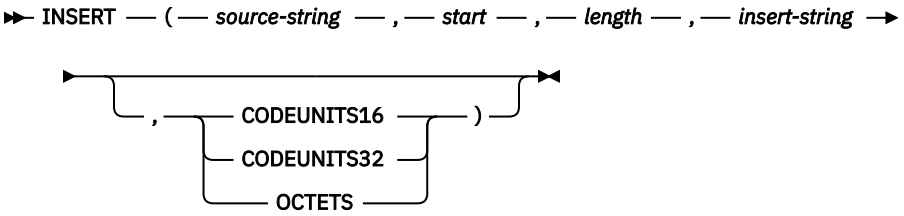
```
VALUES INITCAP ('my_résumé')
1
---------
My_Résumé
```

- *Example 4*: Input the string "élégant" to return the string "Élégant".

```
VALUES INITCAP ('FORMAT:élégant')
1
--------------
Format:Élégant
```

INSERT

The INSERT function returns a string where, beginning at *start* in *source-string*, *length* of the specified code units have been deleted and *insert-string* has been inserted.



The schema is SYSIBM. The SYSFUN version of the INSERT function continues to be available.

The INSERT function is identical to the OVERLAY function, except that the length argument is mandatory.

source-string
An expression that specifies the source string. The expression must return a value that is a built-in string, numeric, boolean, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function.

start
An expression that returns an integer value. The integer value specifies the starting point within the source string where the deletion of code units and the insertion of another string is to begin. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The integer value is the starting point in code units using the specified string units. The integer value must be between 1 and the actual length of *source-string* in implicit or explicit string units plus one (SQLSTATE 22001). If OCTETS is specified and the result is graphic data, the value must be an odd number between 1 and the actual octet length of *source-string* plus one (SQLSTATE 428GC or 22011).

length
An expression that specifies the number of code units (in the specified string units) that are to be deleted from the source string, starting at the position identified by *start*. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value must be positive integer or zero (SQLSTATE 22011). If OCTETS is specified and the result is graphic data, the value must be an even number or zero (SQLSTATE 428GC).

insert-string
An expression that specifies the string to be inserted into *source-string*, starting at the position identified by *start*. The expression must return a value that is a built-in string, numeric, boolean, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of *start* and *length*.

CODEUNITS16 specifies that *start* and *length* are expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *start* and *length* are expressed in 32-bit UTF-32 code units. OCTETS specifies that *start* and *length* are expressed in bytes.

If the string unit is specified as CODEUNITS16 or CODEUNITS32, and the result is a binary string or bit data, an error is returned (SQLSTATE 428GC). If the string unit is specified as CODEUNITS16 or OCTETS, and the string units of source-string is CODEUNITS32, an error is returned (SQLSTATE 428GC). If the string unit is specified as OCTETS, the operation is performed in the code page of the *source-string*. If a string unit is not explicitly specified, the string unit of the *source-string* determines the unit that is used. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

The data type of the result depends on the data types of *source-string* and *insert-string*, as shown in the following tables of supported type combinations. The string unit of the result is the string unit of *source-string*. If either *source-string* or *insert-string* is defined as FOR BIT DATA the other argument cannot be defined with string units of CODEUNITS32.The second table applies only to Unicode databases.

Table 64. Data type of the result as a function of the data types of source-string and insert-string

source-string	***insert-string***	**Result**
CHAR or VARCHAR	CHAR or VARCHAR	VARCHAR
GRAPHIC or VARGRAPHIC	GRAPHIC or VARGRAPHIC	VARGRAPHIC
CLOB	CHAR, VARCHAR, or CLOB	CLOB
CHAR or VARCHAR	CLOB	CLOB
DBCLOB	GRAPHIC, VARGRAPHIC, or DBCLOB	DBCLOB
GRAPHIC or VARGRAPHIC	DBCLOB	DBCLOB
CHAR or VARCHAR	CHAR FOR BIT DATA or VARCHAR FOR BIT DATA	VARCHAR FOR BIT DATA
CHAR FOR BIT DATA or VARCHAR FOR BIT DATA	CHAR, VARCHAR, CHAR FOR BIT DATA, or VARCHAR FOR BIT DATA	VARCHAR FOR BIT DATA
BINARY or VARBINARY	BINARY or VARBINARY	VARBINARY
BLOB	BINARY, VARBINARY, or BLOB	BLOB
BINARY or VARBINARY	BLOB	BLOB

Note: If *source-string* or *insert-string* is a binary data type and the other is a FOR BIT DATA string, the argument that is not a binary data type is handled as if it was cast to the corresponding binary data type.

Table 65. Data type of the result as a function of the data types of source-string and insert-string (Unicode databases only)

source-string	***insert-string***	**Result**
CHAR or VARCHAR	GRAPHIC or VARGRAPHIC	VARCHAR
GRAPHIC or VARGRAPHIC	CHAR or VARCHAR	VARGRAPHIC
CLOB	GRAPHIC, VARGRAPHIC, or DBCLOB	CLOB
DBCLOB	CHAR, VARCHAR, or CLOB	DBCLOB

A *source-string* can have a length of 0; in this case, *start* must be 1 (as implied by the bounds for *start* described previously), and the result of the function is a copy of the *insert-string*.

An *insert-string* can also have a length of 0. This has the effect of deleting the code units identified by *start* and *length* from the *source-string*.

The length attribute of the result is the length attribute of *source-string* plus the length attribute of *insert-string* when the string units of the *source-string* and *insert-string* are the same or the result string units is CODEUNITS32. Special cases are listed in the following table.

Table 66. Data type of the result as a function of the data types of source-string and insert-string (special cases)

source-string		***insert-string***		**Result**	
Data type	**String units**	**Data type**	**String units**	**Length attribute**	**String units**

Table 66. Data type of the result as a function of the data types of source-string and insert-string (special cases) (continued)					
source-string		***insert-string***		**Result**	
Character string with length attribute A	OCTETS	Graphic string with length attribute B	CODEUNITS16	A+3*B	OCTETS
Character string with length attribute A	OCTETS	Graphic string with length attribute B	CODEUNITS32	A+4*B	OCTETS
Character string with length attribute A	OCTETS	Character with length attribute B	CODEUNITS32	A+4*B	OCTETS
Graphic string with length attribute A	CODEUNITS16	Character with length attribute B	OCTETS	A+B	CODEUNITS16
Graphic string with length attribute A	CODEUNITS16	Character with length attribute B	CODEUNITS32	A+2*B	CODEUNITS16
Graphic string with length attribute A	CODEUNITS16	Graphic string with length attribute B	CODEUNITS32	A+2*B	CODEUNITS16

The actual length of the result depends on the actual length of *source-string*, the actual length of the of the deleted string, the actual length of the *insert-string*, and string units used for the *start* and *length* arguments. For example, if the string arguments are character strings in OCTETS and the OCTETS is used as the fourth argument, the actual length of the result is A1 - MIN((A1 - V2 + 1), V3) + A4, where:

- A1 is the actual length of *source-string*
- V2 is the value of *start*
- V3 is the value of *length*
- A4 is the actual length of *insert-string*

If the actual length of the result string exceeds the maximum for the return data type, an error is returned (SQLSTATE 54006).

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Create the strings "INSISTING", "INSISERTING", and "INSTING" from the string "INSERTING" by inserting text into the middle of the existing text.

```
SELECT INSERT('INSERTING',4,2,'IS'),
  INSERT('INSERTING',4,0,'IS'),
  INSERT('INSERTING',4,2,'')
  FROM SYSIBM.SYSDUMMY1
```

- *Example 2*: Create the strings "XXINSERTING", "XXNSERTING", "XXSERTING", and "XXERTING" from the string "INSERTING" by inserting text before the existing text, using 1 as the starting point.

```
SELECT INSERT('INSERTING',1,0,'XX'),
  INSERT('INSERTING',1,1,'XX'),
  INSERT('INSERTING',1,2,'XX'),
  INSERT('INSERTING',1,3,'XX')
  FROM SYSIBM.SYSDUMMY1
```

- *Example 3*: Create the string "ABCABCXX" from the string "ABCABC" by inserting text after the existing text. Because the source string is 6 characters long, set the starting position to 7 (one plus the length of the source string).

```
SELECT INSERT('ABCABC',7,0,'XX')
  FROM SYSIBM.SYSDUMMY1
```

- *Example 4*: Change the string "Hegelstraße" to "Hegelstrasse".

```
SELECT INSERT('Hegelstraße',10,1,'ss',CODEUNITS16)
  FROM SYSIBM.SYSDUMMY1
```

- *Example 5*: The following example works with the Unicode string "&N~AB", where "&" is the musical symbol G clef character, and "~ " is the combining tilde character. This string is shown in different Unicode encoding forms in the following example:

	"&"	"N"	"~"	"A"	"B"
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'

Assume that the variables UTF8_VAR and UTF16_VAR contain the UTF-8 and the UTF-16BE representations of the string, respectively. Use the INSERT function to insert a "C" into the Unicode string "&N~AB".

```
SELECT INSERT(UTF8_VAR, 1, 4, 'C', CODEUNITS16),
  INSERT(UTF8_VAR, 1, 4, 'C', CODEUNITS32),
  INSERT(UTF8_VAR, 1, 4, 'C', OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values "CAB", "CB", and "CN~AB", respectively.

```
SELECT INSERT(UTF8_VAR, 5, 1, 'C', CODEUNITS16),
  INSERT(UTF8_VAR, 5, 1, 'C', CODEUNITS32),
  INSERT(UTF8_VAR, 5, 1, 'C', OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values "&N~CB", "&N~AC", and "&C~AB", respectively.

```
SELECT INSERT(UTF16_VAR, 1, 4, 'C', CODEUNITS16),
  INSERT(UTF16_VAR, 1, 4, 'C', CODEUNITS32),
  INSERT(UTF16_VAR, 1, 4, 'C', OCTETS)
FROM SYSIBM.SYSDUMMY1
```

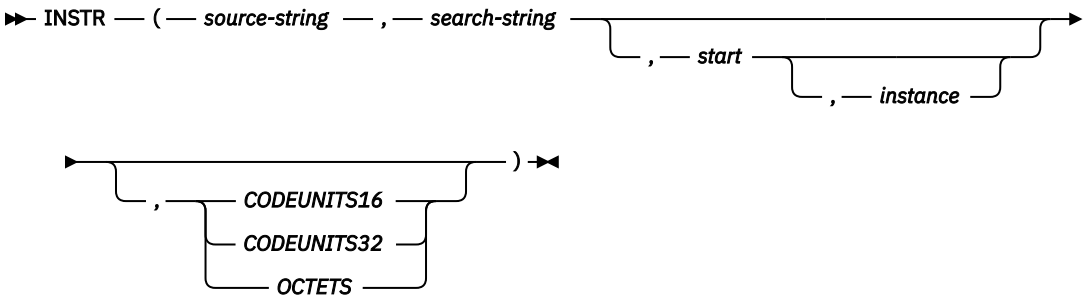
returns the values "CAB", "CB", and "CN~AB", respectively.

```
SELECT INSERT(UTF16_VAR, 5, 2, 'C', CODEUNITS16),
  INSERT(UTF16_VAR, 5, 1, 'C', CODEUNITS32),
  INSERT(UTF16_VAR, 5, 4, 'C', OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values "&N~C", "&N~AC", and "&CAB", respectively.

INSTR

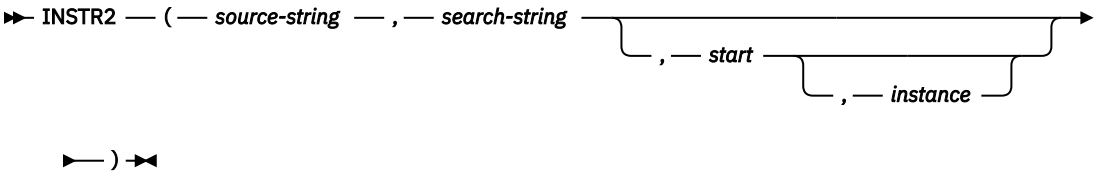
The INSTR function returns the starting position of a string (the *search string*) within another string (the *source string*). The INSTR scalar function is a synonym for the LOCATE_IN_STRING scalar function.



The schema is SYSIBM

INSTR2

The INSTR2 function returns the starting position, in 16-bit UTF-16 string units (CODEUNITS16), of a string within another string.



The schema is SYSIBM.

source-string
An expression that specifies the string in which the search is to take place.

search-string
An expression that specifies the string that is the object of the search.

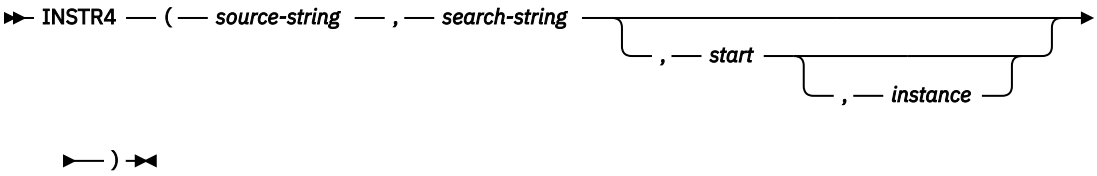
start
An expression that specifies the position within *source-string* at which the search for a match is to start.

instance
An expression that specifies which instance of *search-string* to search for within *source-string*.

The INSTR2 scalar function invoked with character or graphic strings as the first two arguments is equivalent to invoking the LOCATE_IN_STRING function with CODEUNITS16 specified. The INSTR2 scalar function invoked with binary strings as the first two arguments is equivalent to invoking the LOCATE_IN_STRING function without a string units argument.

INSTR4

The INSTR4 function returns the starting position, in 32-bit UTF-32 string units (CODEUNITS32), of a string within another string.



The schema is SYSIBM.

source-string
An expression that specifies the string in which the search is to take place.

search-string
An expression that specifies the string that is the object of the search.

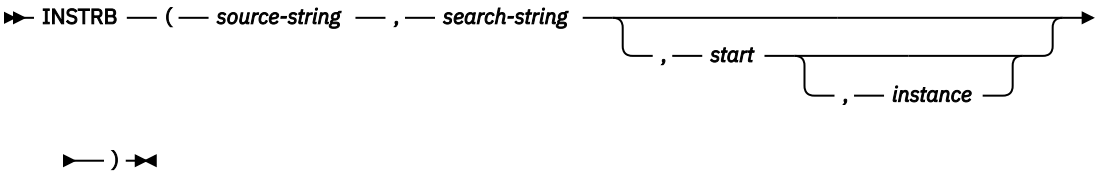
start
An expression that specifies the position within *source-string* at which the search for a match is to start.

instance
An expression that specifies which instance of *search-string* to search for within *source-string*.

The INSTR4 scalar function invoked with character or graphic strings as the first two arguments is equivalent to invoking the LOCATE_IN_STRING function with CODEUNITS32 specified. The INSTR4 scalar function invoked with binary strings as the first two arguments is equivalent to invoking the LOCATE_IN_STRING function without a string units argument.

INSTRB

The INSTRB function returns the starting position, in bytes, of a string within another string.



The schema is SYSIBM.

source-string
An expression that specifies the string in which the search is to take place.

search-string
An expression that specifies the string that is the object of the search.

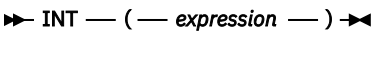
start
An expression that specifies the position within the source string at which the search for a match is to start.

instance
An expression that specifies which instance of the search string to search for within the source string.

The INSTRB scalar function invoked with character or graphic strings as the first two arguments is equivalent to invoking the LOCATE_IN_STRING function with OCTETS specified. The INSTRB scalar function invoked with binary strings as the first two arguments is equivalent to invoking the LOCATE_IN_STRING function without a string units argument.

INT

The INT function returns a large integer (a binary integer with a precision of 31 bits) representation of a value of a different data type.



The schema is SYSIBM.

The INT scalar function is a synonym for the INTEGER scalar function.

INTERVAL

The INTERVAL function converts a character string representation of an interval to a decimal duration.

The schema is SYSIBM.

Syntax

string-constant
A character string representation of an interval, for example:

```
'4 years 2 months 3 days'
'3 day 4 year 2 month'
'-4y -2 m -3d'
'-2 hr -21 min -34sec'
'4years 2months 3 days 2 hours 21minutes 34seconds 75 milliseconds 27 microseconds'
'2 mons 3 days 4 yrs 2 hrs 20 mins 30 secs 75 ms 27 us'
```

Note:

- The input must be a string constant, not an expression or column value (sqlcode SQL0171N with SQLSTATE=42815).
- A blank between each value and its unit keyword is optional.
- Values must be either all positive or all negative.
- The unit keywords are case-insensitive.
- The order of the unit keywords is unimportant.
- A unit keyword cannot be used more than once.
- The number that is specified for each unit must be a whole number.
- The following statements are equivalent:

```
INTERVAL (string-constant)
CAST (string-constant as INTERVAL)
```

The specified interval can be of one of the following types:

time interval
The input value does not contain units other than hours, minutes, or seconds, and cannot exceed the equivalent of `'99h 99m 99s'` (sqlcode SQL0105N with SQLSTATE=42604). A minutes value that exceeds 99 is converted to hours, and a seconds value that exceeds 99 is converted to minutes. For example, 100 seconds is converted to 1 minute and 40 seconds.

date interval
The input value does not contain units other than years, months, or days, and cannot exceed the equivalent of `'9999y 99m 99d'` (sqlcode SQL0105N with SQLSTATE=42604). A months value that exceeds 99 is converted to years, and a days value that exceeds 99 is converted to 30-day months. For example, 100 days is converted to 3 months and 10 days.

timestamp interval
The input value cannot exceed the equivalent of `'9999y 99m 99d 99h 99m 99s 999999us'` (sqlcode SQL0105N with SQLSTATE=42604). The millisecond and microsecond values are combined into a single microsecond value before they are evaluated. If the resulting value exceeds 999999 microseconds, it is converted to seconds. For example, `'88 ms 5000000 us'` is converted to 5088000 microseconds. This exceeds 999999 microseconds, so it is converted to 5.088000 seconds. A seconds value that exceeds 99 is converted to minutes. A minutes value that exceeds 99 is converted to hours. An hours value that exceeds 99 is converted to days. A days value that exceeds 99 is converted to 30-day months. A months value that exceeds 99 is converted to years.

Table 67. Keywords for representing time units

Keywords	**Maximum value allowed when specifying a...**		
	time interval	**date interval**	**timestamp interval**
year, years, yrs, yr, y	(not allowed)	9999	9999
month, months, mons, mon	(not allowed)	119999	119999
day, days, d	(not allowed)	3599999	3599999
hour, hours, hrs, hr, h	99	(not allowed)	86399999
minute, minutes, mins, min, m	5999	(not allowed)	2147483647
second, seconds, secs, sec, s	359999	(not allowed)	2147483647
millisecond, milliseconds, ms	(not allowed)	(not allowed)	2147483647

Table 67. Keywords for representing time units (continued)			
Keywords	**Maximum value allowed when specifying a...**		
	time interval	**date interval**	**timestamp interval**
microsecond, microseconds, us	(not allowed)	(not allowed)	2147483647

Result

The data type of the result depends on the type of the input interval. If the input value represents:

- A date interval, the result is a date duration, which is a DECIMAL(8,0) value.
- A time interval, the result is a time duration, which is a DECIMAL(6,0) value.
- A timestamp interval, the result is a timestamp duration, which is a DECIMAL value. The precision of this value depends on whether, after combining millisecond and microsecond values into a single microsecond value, and after then converting any microseconds in excess of 999999 to seconds, the number of microseconds remaining is zero:
  - If so, the result is DECIMAL(14,0)
  - If not, the result is DECIMAL(20,6)

Examples

- The following statement returns the DECIMAL(8,0) value 40203:

```
interval('4years 2months 3days')
```

- The following statement returns the DECIMAL(6,0) value -122030:

```
interval('-12 hours -20 minutes -30 seconds')
```

- The following statement returns the DECIMAL(20,6) value 40801092630.007055:

```
interval('4 years 9 hour 26min 30 seconds 7 ms 55us 8months 1d')
```

- The following statement returns the DECIMAL(14,0) value 22035:

```
interval('2 hours 20 minutes 30 seconds 1500 ms 3500000 us')
```

  The millisecond (1500) and microsecond (3500000) values are combined into a single microsecond value (5000000) before they are evaluated. The resulting value exceeds 999999 microseconds, so it is converted to 5 seconds, which are added to the specified 30 seconds for a total of 35 seconds.

- The following statement returns the DECIMAL(8,0) value 90714, which corresponds to 9 years, 7 months, and 14 days.

```
interval('1 years 100 months 104 days')
```

  Because the number of days exceeds 99, the 104 days are converted to 3 30-day months, plus a remainder of 14 days. The 3 months are added to the specified 100 months for a total of 103 months. Because the number of months exceeds 99, the 103 months are converted to 8 12-month years, plus a remainder of 7 months. The 8 years are added to the specified 1 year for a total of 9 years.

- The following statement returns the DECIMAL(6,0) value 230120, which corresponds to 23 hours, 1 minute, and 20 seconds.

```
interval('20 hours 181 minutes 20 seconds')
```

  Because the number of minutes exceeds 99, the 181 minutes are converted to 3 60-minute hours, plus a remainder of 1 minute. The 3 hours are added to the specified 20 hours for a total of 23 hours.

Related reference

"Datetime operations and durations" on page 145
Datetime values can be incremented, decremented, and subtracted. These operations can involve decimal numbers called durations.

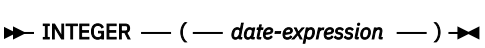
INTEGER

The INTEGER function returns a large integer (a binary integer with a precision of 31 bits) representation of a value of a different data type.

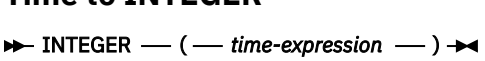
Numeric to INTEGER

String to INTEGER

Date to INTEGER



Time to INTEGER



Boolean to INTEGER

The schema is SYSIBM.

Numeric to INTEGER

numeric-expression
An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a large integer column or variable. The fractional part of the argument is truncated. If the whole part of the argument is not within the range of integers, an error is returned (SQLSTATE 22003).

String to INTEGER

string-expression
An expression that returns a value that is a character-string or Unicode graphic-string representation of a number with a of length not greater than the maximum length of a character constant.

The result is the same number that would result from CAST(*string-expresssion* AS INTEGER). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018). If the whole part of the argument is not within the range of integers, an error is returned (SQLSTATE 22003). The data type of *string-expression* must not be CLOB or DBCLOB (SQLSTATE 42884).

Date to INTEGER

date-expression
An expression that returns a value of the DATE data type. The result is an INTEGER value representing the date as *yyyymmdd*.

Time to INTEGER

time-expression
An expression that returns a value of the TIME data type. The result is an INTEGER value representing the time as *hhmmss*.

Boolean to INTEGER

boolean-expression
An expression that returns a Boolean value (TRUE or FALSE). The result is either 1 (for TRUE) or 0 (for FALSE).

Result

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Notes

- **Increasing portability of applications:** If the first argument is numeric, or if the first argument is a string and the length argument is specified, use the "CAST specification" on page 152 instead of this function to increase the portability of your applications.

Examples

- *Example 1*: Using the EMPLOYEE table, select a list containing salary (SALARY) divided by education level (EDLEVEL). Truncate any decimal in the calculation. The list should also contain the values used in the calculation and employee number (EMPNO). The list should be in descending order of the calculated value.

```
SELECT INTEGER (SALARY / EDLEVEL), SALARY, EDLEVEL, EMPNO
  FROM EMPLOYEE
  ORDER BY 1 DESC
```

- *Example 2*: Using the EMPLOYEE table, select the EMPNO column in integer form for further processing in the application.

```
SELECT INTEGER(EMPNO) FROM EMPLOYEE
```

- *Example 3*: Assume that the column BIRTHDATE (whose data type is DATE) has an internal value equivalent to '1964-07-20'.

```
INTEGER(BIRTHDATE)
```

  results in the value `19 640 720`.

- *Example 4*: Assume that the column STARTTIME (whose data type is TIME) has an internal value equivalent to '12:03:04'.

```
INTEGER(STARTTIME)
```

  results in the value `120 304`.

- *Example 5*: The following statement returns the value 1 of data type INTEGER.

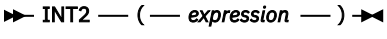
```
values INTEGER(TRUE)
```

- *Example 6*: The following statement returns the value 0 of data type INTEGER.

```
values INTEGER(3>3)
```

INT2

The INT2 function returns a small integer (a binary integer with a precision of 15 bits) representation of a value of a different data type.

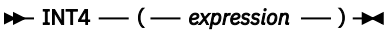


The schema is SYSIBM.

The INT2 scalar function is a synonym for the SMALLINT scalar function.

INT4

The INT4 function returns a large integer (a binary integer with a precision of 31 bits) representation of a value of a different data type.

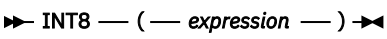


The schema is SYSIBM.

The INT4 scalar function is a synonym for the INTEGER scalar function.

INT8

The INT8 function returns a big integer (a binary integer with a precision of 63 bits) representation of a value of a different data type.



The schema is SYSIBM.

The INT8 scalar function is a synonym for the BIGINT scalar function.

INTNAND, INTNOR, INTNXOR, and INTNNOT

These bitwise functions operate on the "two's complement" representation of the integer value of the input arguments and return the result as a corresponding base 10 integer value.

The schema is SYSIBM.

In each function, the placeholder *N* represents the byte size of the integer data type that the function operates on and returns, as shown in Table 69 on page 382.

Table 68. The bit manipulation functions		
Function	**Description**	**A bit in the two's complement representation of the result is:**
INTNAND	Performs a bitwise AND operation.	1 only if the corresponding bits in both arguments are 1.
INTNOR	Performs a bitwise OR operation.	1 unless the corresponding bits in both arguments are zero.
INTNXOR	Performs a bitwise exclusive OR operation.	1 unless the corresponding bits in both arguments are the same.
INTNNOT	Performs a bitwise NOT operation.	Opposite of the corresponding bit in the argument.

Table 69. Meaning of placeholder N	
Value of *N*	**Data type function operates on and returns**
2	SMALLINT
4	INTEGER
8	BIGINT

expression* or *expression1* or *expression2

The arguments must be integer values represented by the data types SMALLINT, INTEGER, BIGINT, DECFLOAT, DECIMAL, REAL, or DOUBLE. If the input argument is not of the same data type as represented by N, then the input is implicitly cast to the data type represented by *N*. As a result, if a value larger than the maximum value supported by the data type represented by *N* is passed as input to the function, then an overflow can occur (SQLSTATE=22003).

If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Due to differences in internal representation between data types and on different hardware platforms, using functions (such as HEX) or host language constructs to view or compare internal representations of BIT function results and arguments is data type-dependent and not portable. The data type- and platform-independent way to view or compare BIT function results and arguments is to use the actual integer values.

Examples

- *Example 1*: A value larger than the maximum supported by 2 byte SMALLINT is passed as input to the function INT2AND.

```
select INT2AND(1234567,1) from SYSIBM.SYSDUMMY1
SQL0413N  Overflow occurred during numeric data type conversion.
SQLSTATE=22003
```

- *Example 2*: Assume BIGINT columns col1 and col2 have values 137266 and 123825 respectively.

```
SELECT INT8AND(col1,col2) from TAB1
returns the value 48
```

- *Example 3*: Assume SMALLINT columns col1 and col2 have values 12 and 13 respectively.

```
SELECT INT2AND(col1,col2) from TAB1
returns the value 12
```

ISNULL

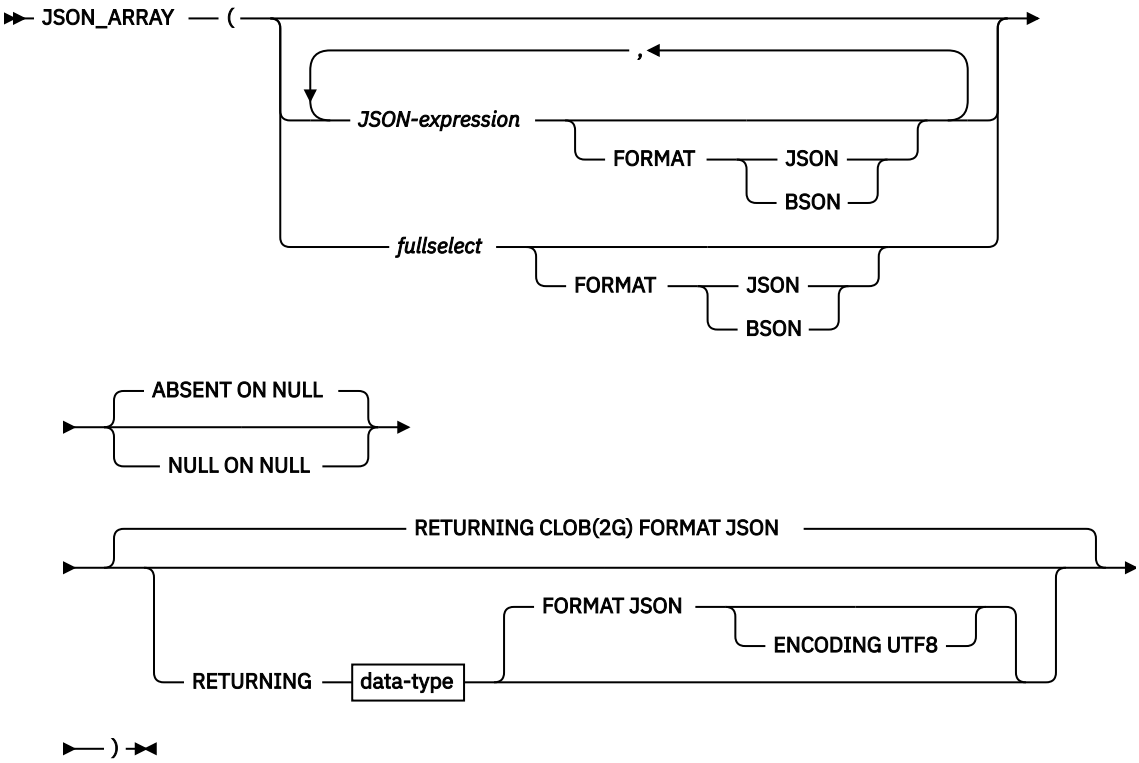
The ISNULL function returns the first non-null expression in a list of two expressions.

The schema is SYSIBM.

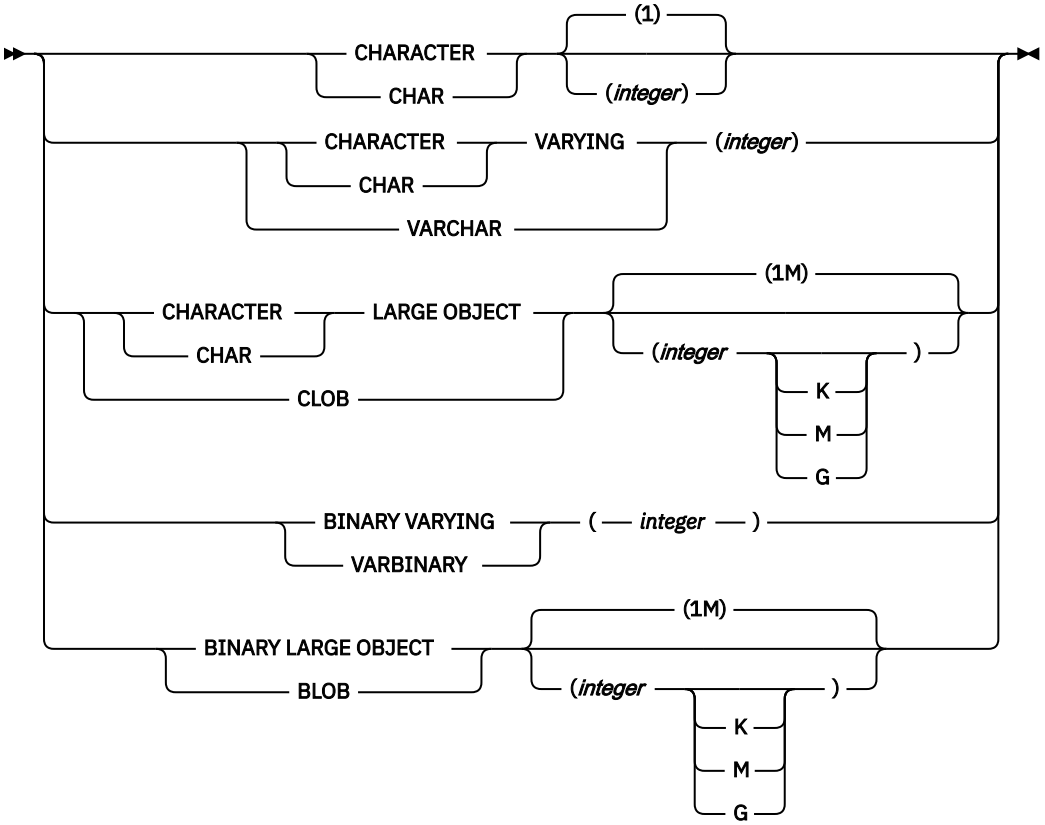
The ISNULL function is identical to the “COALESCE ” on page 308, except that ISNULL is limited to two arguments.

JSON_ARRAY

The JSON_ARRAY function generates a JSON array by explicitly listing the array elements by using *JSON-expression*, or by using a query.



data-type



Although the schema for this function is SYSIBM, the function cannot be specified as a qualified name.

JSON-expression
The expression that is used to generate a value in the JSON array.

The result type of this expression can be any built-in data type, except the following data types (SQLSTATE 42815):

- GRAPHIC
- VARGRAPHIC
- DBCLOB
- BINARY
- CHAR FOR BIT DATA
- VARCHAR FOR BIT DATA
- XML
- A user-defined type that is sourced on any of the previously listed data types

If the generated value is numeric, it cannot be Infinity, NaN, or sNaN (SQLSTATE 22023).

If FORMAT JSON or FORMAT BSON is not specified, and the generated value is not numeric, any special characters (for example, backslash or double quotation marks) within the result string are escaped.

If FORMAT JSON or FORMAT BSON is not specified and *JSON-expression* is binary string type, it is interpreted as FORMAT BSON.

FORMAT JSON
JSON-expression is formatted as JSON data.

If *JSON-expression* is a character string data type, it is treated as JSON data.

If *JSON-expression* is a binary string data type, it is interpreted as UTF-8 data.

FORMAT BSON
Specifies that *JSON-expression* is formatted as the BSON representation of JSON data (SQLSTATE 22032). *JSON-expression* must be a binary string data type (SQLSTATE 42815).

fullselect
Specifies a fullselect that returns a single column to be used to generate the values in the array (SQLSTATE 42823). The value of each row is used to generate a value in the JSON array.

The result type of this column cannot be any of the following data types (SQLSTATE 42815):

- GRAPHIC
- VARGRAPHIC
- DBCLOB
- BINARY
- CHAR FOR BIT DATA
- VARCHAR FOR BIT DATA
- XML
- A user-defined type that is sourced on any of the previously listed data types

If the generated value is numeric, it cannot be Infinity, NaN, or sNaN (SQLSTATE 22023).

If FORMAT JSON or FORMAT BSON is not specified and the generated value is not numeric, any special characters (for example, backslash or double quotation marks) within the result string are escaped.

If FORMAT JSON or FORMAT BSON is not specified and the *fullselect* is binary string type, it is interpreted as FORMAT BSON.

FORMAT JSON
fullselect is formatted as JSON data.

If *fullselect* is a character string data type, it is treated as JSON data.

If *fullselect* is a binary string data type, it is interpreted as UTF-8 data.

FORMAT BSON
Specifies that *fullselect* is formatted as the BSON representation of JSON data (SQLSTATE 22032). *fullselect* must be a binary string data type (SQLSTATE 42815).

ON NULL
Specifies what to return when an array element produced by *JSON-expression* or *fullselect* is the null value.

ABSENT ON NULL
A null array element is not included in the JSON array. This clause is the default.

NULL ON NULL
A null array element is included in the JSON array.

RETURNING data-type
The data type of the result can be CHAR, VARCHAR, CLOB, VARBINARY, or BLOB (SQLSTATE 42815). The default is CLOB (2 GB).

See "CREATE TABLE " on page 1351 for the description of built-in data types.

FORMAT JSON
The returned data is formatted as JSON data.

ENCODING UTF8
Specifies the encoding to use when *data-type* is a binary string type. This clause is allowed only for binary string types. The default for binary strings is UTF-8.

Notes

- If parameter markers are not explicitly cast to a supported data type, an error is returned (SQLSTATE 42815)

Examples

1. Generate a JSON array that contains the values 'Washington', 'Jefferson', and 'Hamilton'.

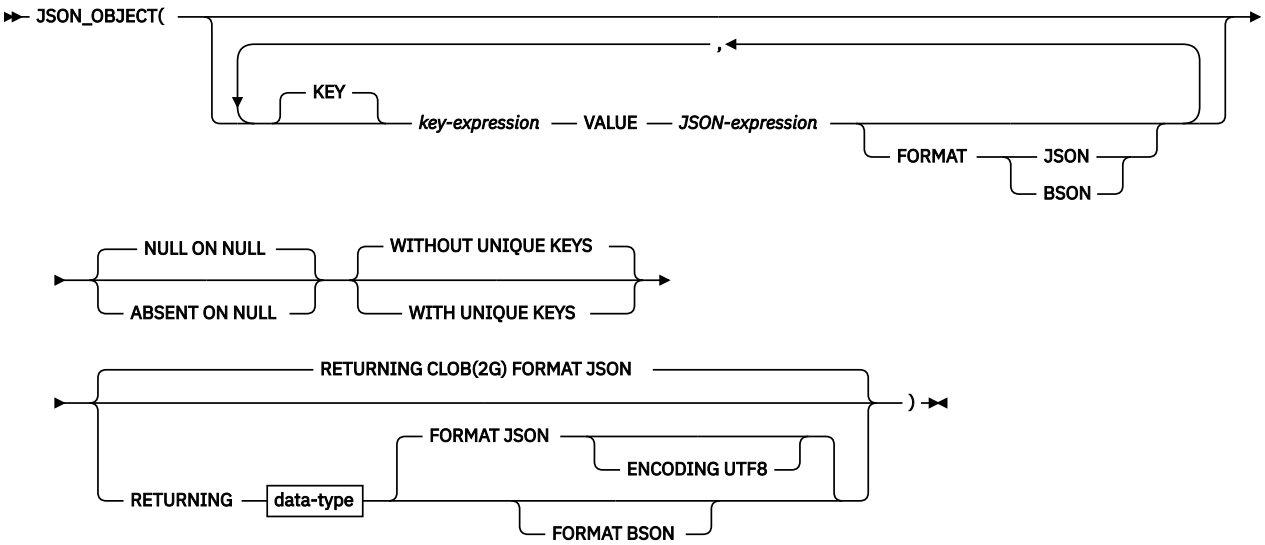
```
VALUES JSON_ARRAY('Washington', 'Jefferson', 'Hamilton');

1
-------------------------------------
["Washington","Jefferson","Hamilton"]
```

2. Generate a JSON array that includes all department numbers.

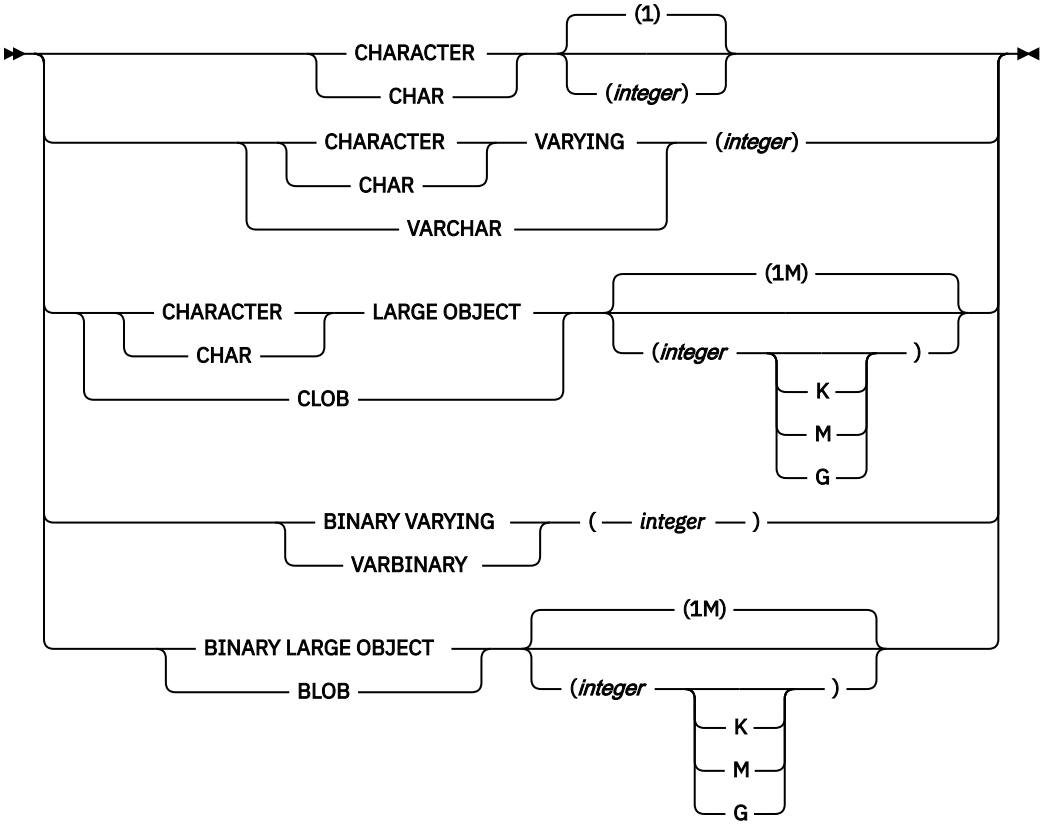
```
VALUES JSON_ARRAY(SELECT DEPTNO FROM DEPT);

1
------------------------------
["F22","G22","H22","I22","J22"]
```

JSON_OBJECT

The JSON_OBJECT function generates a JSON object by using the specified key:value pairs. If no key:value pairs are provided, an empty object is returned.



data-type



Although the schema for this function is SYSIBM, the function cannot be specified as a qualified name.

key-expression
The name of the JSON key. The name must not be null (SQLSTATE 22004). The result of *key-expression* must be a built-in character string data type, except the following data types (SQLSTATE 42815):

- GRAPHIC
- VARGRAPHIC
- DBCLOB
- CHAR FOR BIT DATA
- VARCHAR FOR BIT DATA

JSON-expression
The expression that is used to generate the JSON value that is associated with *key-expression*.

The result type of this expression can be any built-in data type, except the following data types (SQLSTATE 42815):

- GRAPHIC
- VARGRAPHIC
- DBCLOB
- BINARY
- CHAR FOR BIT DATA
- VARCHAR FOR BIT DATA
- XML
- A user-defined type that is sourced on any of the previously listed data types

If the generated value is numeric, it cannot be Infinity, NaN, or sNaN (SQLSTATE 22023).

If FORMAT JSON or FORMAT BSON is not specified, and the generated value is not numeric, any special characters (for example, backslash or double quotation marks) within the result string are escaped.

If FORMAT JSON or FORMAT BSON is not specified and the expression is binary string type, it is interpreted as FORMAT BSON.

FORMAT JSON
JSON-expression is formatted as JSON data.

If *JSON-expression* is a character string data type, it is treated as JSON data.

If *JSON-expression* is a binary string data type, it is interpreted as UTF-8 data.

FORMAT BSON
Specifies that *JSON-expression* is formatted as the BSON representation of JSON data (SQLSTATE 22032). *JSON-expression* must be a binary string data type (SQLSTATE 42815).

ON NULL
Specifies what to return when *JSON-expression* is the null value.

NULL ON NULL
A character string that represents the null value is returned. This clause is the default.

ABSENT ON NULL
The key:value pair is omitted from the JSON object.

WITHOUT UNIQUE KEYS or WITH UNIQUE KEYS
Specifies whether the key values for the resulting JSON object must be unique.

WITHOUT UNIQUE KEYS
The resulting JSON object is not checked for duplicate keys. This clause is the default.

WITH UNIQUE KEYS
The resulting JSON object must have unique key values (SQLSTATE 22037).

Generating a JSON object with unique keys is considered best practice. If *key-expression* generates unique key names, omit WITH UNIQUE KEYS to improve performance.

RETURNING data-type
The data type of the result can be CHAR, VARCHAR, CLOB, VARBINARY, or BLOB (SQLSTATE 42815). The default is CLOB (2 GB).

See “CREATE TABLE ” on page 1351 for the description of built-in data types.

FORMAT JSON
The returned data is formatted as JSON data.

ENCODING UTF8
Specifies the encoding to use when *data-type* is a binary string type. This clause is allowed only for binary string types. The default for binary strings is UTF-8.

FORMAT BSON
The returned data is formatted as the BSON representation of JSON data (SQLSTATE 22032). *data-type* must be a binary string data type (SQLSTATE 42815).

Notes

- If parameter markers are not explicitly cast to a supported data type, an error is returned (SQLSTATE 42815)

Examples

1. Generate a JSON object for a person’s name.

```
VALUES JSON_OBJECT(KEY 'first' VALUE 'John', KEY 'last' VALUE 'Doe')

1
```

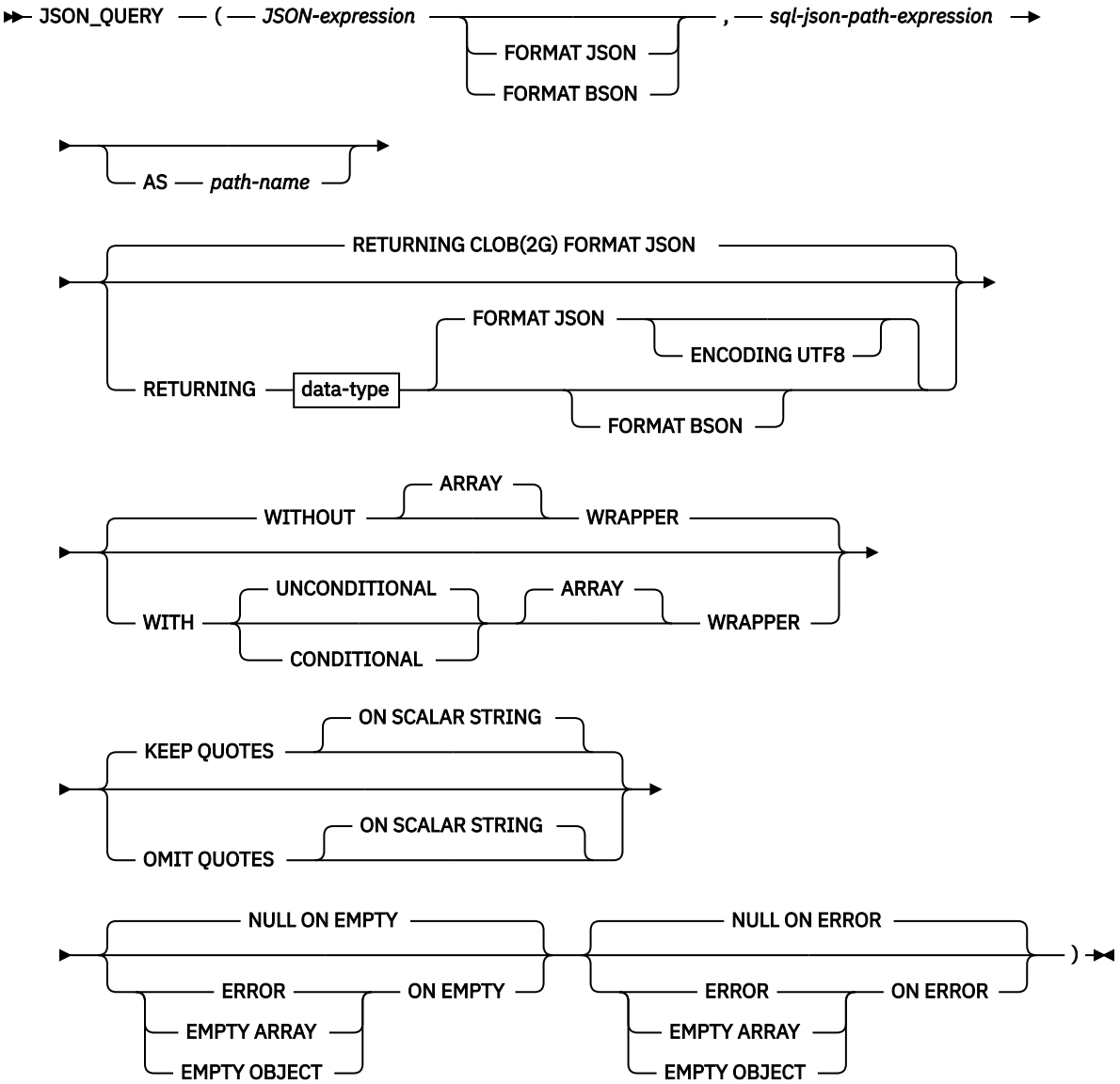
```
----------------------------
{"first":"John","last":"Doe"}
```

2. Generate a JSON object that contains the surname, hire date, and salary for the employee with an employee number of '000020'.

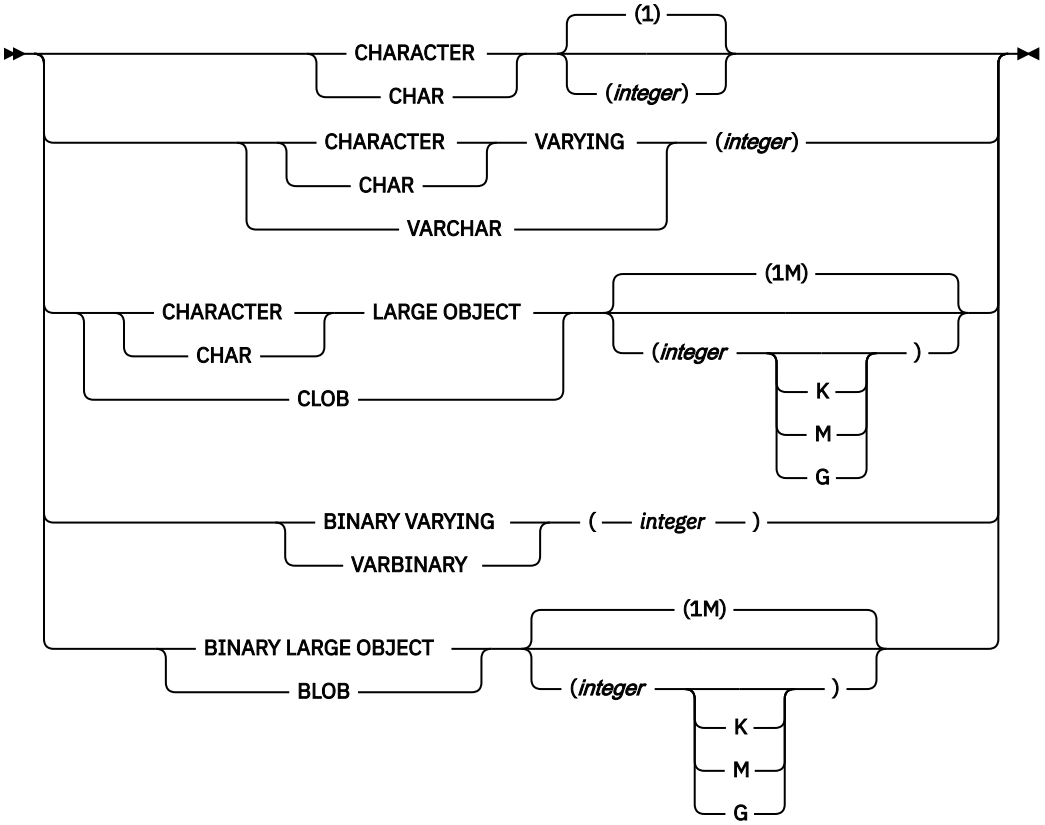
```
SELECT JSON_OBJECT(KEY 'Last name' VALUE LASTNAME,
                   KEY 'Hire date' VALUE HIREDATE,
                   KEY 'Salary' VALUE SALARY)
   FROM EMPLOYEE
   WHERE EMPNO  = '000020';

1
--------------------------------------------------------------------
{"Last name":"THOMPSON","Hire date":"1973-10-10","Salary":41250.00}
```

JSON_QUERY

The JSON_QUERY function returns an SQL/JSON value from the specified JSON text by using an SQL/JSON path expression.



data-type



Although the schema for this function is SYSIBM, the function cannot be specified as a qualified name.

JSON-expression
An expression that returns a value that is a built-in string data type, except the following data types (SQLSTATE 42815):

- GRAPHIC
- VARGRAPHIC
- DBCLOB
- BINARY
- CHAR FOR BIT DATA
- VARCHAR FOR BIT DATA
- A user-defined type that is sourced on any of the previously listed data types

If a character value is returned, it must contain correctly formatted JSON data (SQLSTATE 22032). If a binary data type is returned, it is interpreted according to the explicit or implicit FORMAT clause.

FORMAT JSON
JSON-expression is formatted as JSON data.

If *JSON-expression* is a character string data type, it is treated as JSON data.

If *JSON-expression* is a binary string data type, it is interpreted as UTF-8 data.

FORMAT BSON
Specifies that *JSON-expression* is formatted as the BSON representation of JSON data (SQLSTATE 22032). *JSON-expression* must be a binary string data type (SQLSTATE 42815).

sql-json-path-expression
An expression that returns a value that is a built-in character string data type. The string is interpreted as an SQL/JSON path expression and is used to locate a JSON value within the JSON data that

is specified by *JSON-expression*. For more information about the SQL/JSON path expression, see "sql-json-path-expression" on page 179.

AS *path-name*
Specifies a name to be used to identify *sql-json-path-expression*.

RETURNING *data-type*
The data type of the result can be CHAR, VARCHAR, CLOB, VARBINARY, or BLOB (SQLSTATE 42815). The default is CLOB (2 GB).

See "CREATE TABLE " on page 1351 for the description of built-in data types.

FORMAT JSON
The returned data is formatted as JSON data.

ENCODING UTF8
Specifies the encoding to use when *data-type* is a binary string type. This clause is allowed only for binary string types. The default for binary strings is UTF-8.

FORMAT BSON
The returned data is formatted as the BSON representation of JSON data (SQLSTATE 22032). *data-type* must be a binary string data type (SQLSTATE 42815).

WITHOUT ARRAY WRAPPER or WITH ARRAY WRAPPER
Specifies whether the output value is wrapped in a JSON array.

WITHOUT ARRAY WRAPPER
The result is not wrapped. This clause is the default. Using a strict SQL/JSON path definition that resolves to a sequence of two or more SQL/JSON elements results in an error (SQLSTATE 2203A). Using a lax SQL/JSON path definition with the ON EMPTY that resolves to a sequence of two or more SQL/JSON elements will result in an error (SQLSTATE 22035).

WITH UNCONDITIONAL ARRAY WRAPPER
The result is enclosed in square brackets to create a JSON array.

WITH CONDITIONAL ARRAY WRAPPER
Indicates that the result is enclosed in square brackets to create a JSON array for either of the following scenarios:

- More than one SQL/JSON element is returned.
- A single SQL/JSON element that is not a JSON array or a JSON object is returned.

KEEP QUOTES or OMIT QUOTES
Specifies whether the surrounding quotation marks should be removed when a scalar string is returned.

KEEP QUOTES
Quotation marks are not removed from scalar strings. This clause is the default.

OMIT QUOTES
Quotation marks are removed from scalar strings. When OMIT QUOTES is specified, the WITH ARRAY WRAPPER clause cannot be specified (SQLSTATE 42601).

ON EMPTY
Specifies the behavior when an empty sequence is returned by *sql-json-path-expression*.

NULL ON EMPTY
A null value is returned. This clause is the default.

ERROR ON EMPTY
An error is returned.

EMPTY ARRAY ON EMPTY
An empty array is returned.

EMPTY OBJECT ON EMPTY
An empty object is returned.

ON ERROR
Specifies the behavior when an error is encountered by JSON_QUERY.

NULL ON ERROR
A null value is returned. This clause is the default.

ERROR ON ERROR
An error is returned.

EMPTY ARRAY ON ERROR
An empty array is returned.

EMPTY OBJECT ON ERROR
An empty object is returned.

The result can be null. If *JSON-expression* is null, the result is the null value.

Notes

- If parameter markers are not explicitly cast to a supported data type, an error is returned (SQLSTATE 42815)

Example

1. Return the JSON object that is associated with the name key from JSON text.

```
VALUES JSON_QUERY('{"id":"701", "name":{"first":"John", "last":"Doe"}}', '$.name');
```

The result is the following string that represents a JSON object:

```
{"first":"John", "last":"Doe"}
```

See the example at "sql-json-path-expression" on page 179 for different array wrapper options with JSON_QUERY.

JSON_TO_BSON

The JSON_TO_BSON function converts a string that contains data that is formatted for JSON to a binary string that contains data that is formatted as BSON.

Although the schema for this function is SYSIBM, the function cannot be specified as a qualified name.

JSON-expression
Specifies an expression that returns a character string value. It must contain formatted JSON data (SQLSTATE 22032).

If *JSON-expression* can be null, the result can be null; if *JSON-expression* is null, the result is the null value.

Notes

- If parameter markers are not explicitly cast to a supported data type, an error is returned (SQLSTATE 42815)

Examples

1. Convert a valid JSON document to BSON format and insert it into in a table.

```
INSERT INTO TESTJSON VALUES (JSON_TO_BSON('{"Name":"George"}'));
```

This example inserts x'16000000024E616D65000700000047656F7267650000' into the table named TESTJSON.

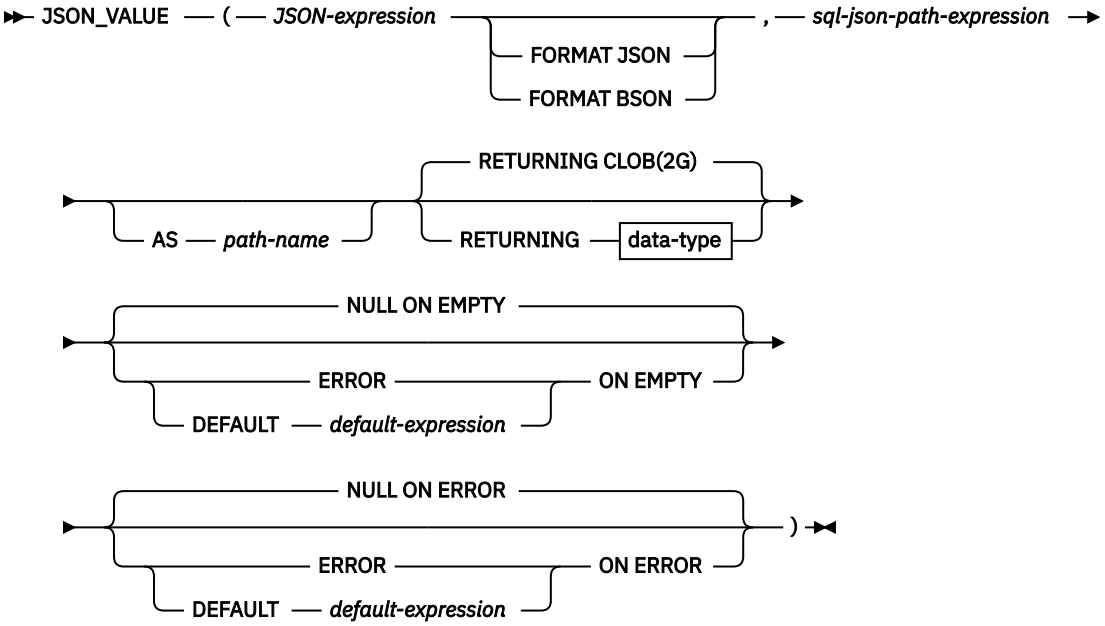
2. Convert an incorrectly formatted JSON document, which is missing a value for the key "Name".

```
INSERT INTO TESTJSON VALUES (JSON_TO_BSON('{"Name":, "Age" : 32}'));

SQL16402N  JSON data is not valid.  SQLSTATE=22032
```

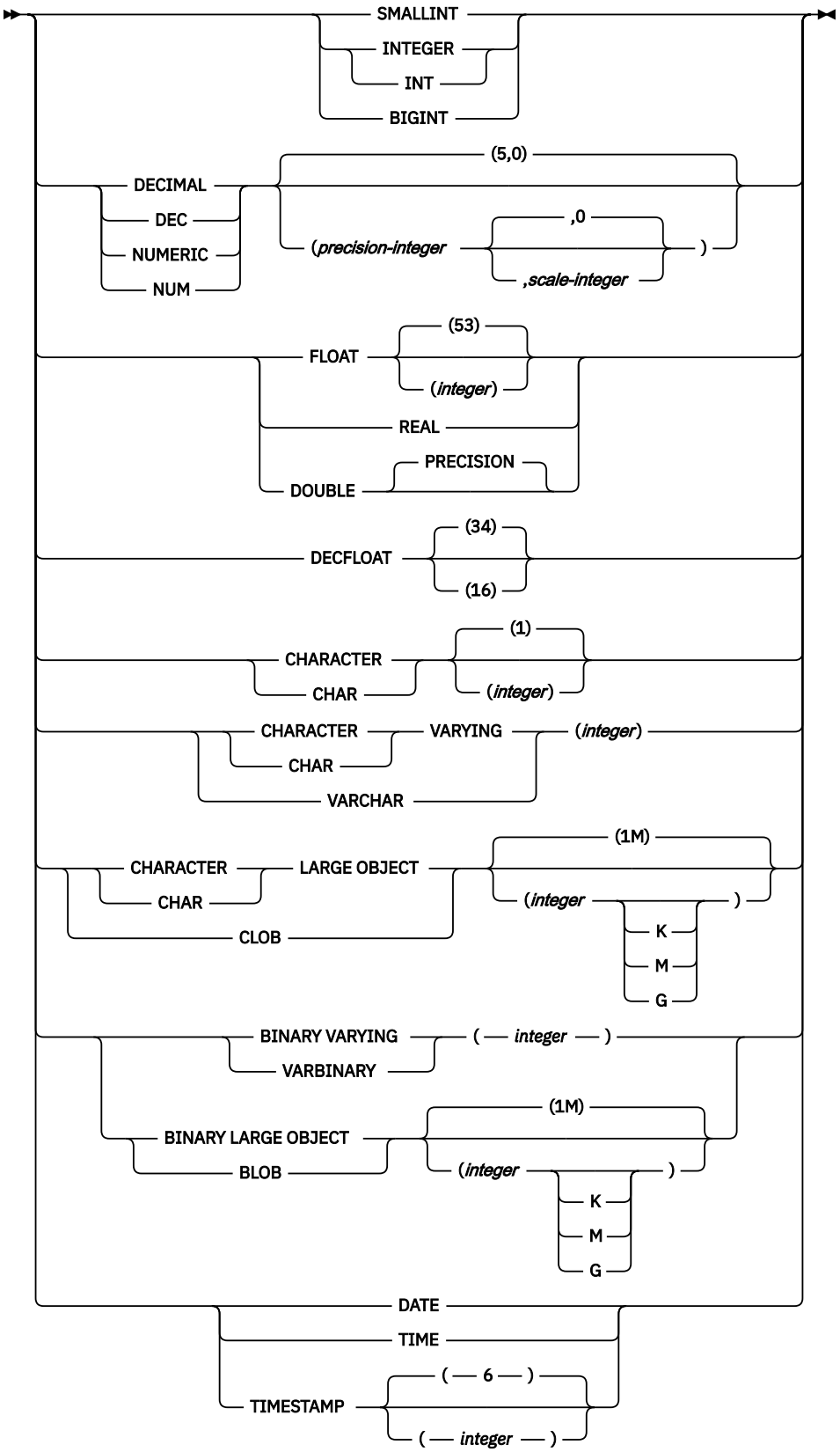
This example results in an error.

JSON_VALUE

The JSON_VALUE function returns an SQL scalar value from JSON text, by using an SQL/JSON path expression.



data-type



Although the schema for this function is SYSIBM, the function cannot be specified as a qualified name.

JSON-expression

An expression that returns a value that is a built-in string data type, except the following data types (SQLSTATE 42815):

- GRAPHIC
- VARGRAPHIC
- DBCLOB
- BINARY
- CHAR FOR BIT DATA
- VARCHAR FOR BIT DATA
- A user-defined type that is sourced on any of the previously listed data types

If a character value is returned, it must contain correctly formatted JSON data (SQLSTATE 22032). If a binary data type is returned, it is interpreted according to the explicit or implicit FORMAT clause.

FORMAT JSON
JSON-expression is formatted as JSON data.

If *JSON-expression* is a character string data type, it is treated as JSON data.

If *JSON-expression* is a binary string data type, it is interpreted as UTF-8 data.

FORMAT BSON
Specifies that *JSON-expression* is formatted as the BSON representation of JSON data (SQLSTATE 22032). *JSON-expression* must be a binary string data type (SQLSTATE 42815).

sql-json-path-expression
An expression that returns a value that is a built-in character string data type. The string is interpreted as an SQL/JSON path expression and is used to locate a JSON value within the JSON data that is specified by *JSON-expression*. For more information about the SQL/JSON path expression, see "sql-json-path-expression" on page 179.

AS *path-name*
Specifies a name to be used to identify *sql-json-path-expression*.

RETURNING *data-type*
Specifies the data type of the result. The default is CLOB (2 GB). The default encoding used when data-type is a binary string type is UTF-8.

See "CREATE TABLE " on page 1351 for the description of built-in data types.

ON EMPTY
Specifies the behavior when an empty sequence is returned by *sql-json-path-expression*.

NULL ON EMPTY
A null value is returned. This clause is the default.

ERROR ON EMPTY
An error is returned.

DEFAULT *default-expression* ON EMPTY
The value that is specified by *default-expression* is returned. The data type of *default-expression* must be the same as the returning data type (SQLSTATE 42815).

ON ERROR
Specifies the behavior when an error is encountered by JSON_VALUE.

NULL ON ERROR
A null value is returned. This clause is the default.

ERROR ON ERROR
An error is returned.

DEFAULT *default-expression* ON ERROR
The value that is specified by *default-expression* is returned. The data type of *default-expression* must be the same as the returning data type (SQLSTATE 42815).

The result can be null. If *JSON-expression* is null, the result is the null value.

Notes

- If parameter markers are not explicitly cast to a supported data type, an error is returned (SQLSTATE 42815)

Examples

1. Return a value from JSON text as an integer.

```
VALUES (JSON_VALUE('{"id":"987"}', 'strict $.id' RETURNING INTEGER));
```

   The result is 987.

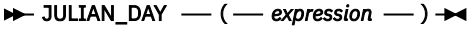
2. Get the value for the bonus field from JSON text. Return it as an integer.

```
VALUES (JSON_VALUE('{"pay":{"salary":94250.00,"bonus":800.00,"comm":3300.00}}',
                   'strict $.pay.bonus' RETURNING INTEGER));
```

   The result is 800.

JULIAN_DAY

Returns an integer value representing the number of days from January 1, 4713 B.C. (the start of the Julian date calendar) to the date value specified in the argument.



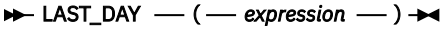
The schema is SYSFUN.

expression
An expression that returns a value of one of the following built-in data types: a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

LAST_DAY

The LAST_DAY scalar function returns a date or timestamp value that represents the last day of the month of the argument.



The schema is SYSIBM.

expression
An expression that specifies the starting date. The expression must return a value of one of the following built-in data types: a DATE or a TIMESTAMP.

The result of the function has the same data type as *expression*, unless *expression* is a string, in which case the result data type is DATE. The result can be null; if the value of *date-expression* is null, the result is the null value.

Any hours, minutes, seconds or fractional seconds information included in *expression* is not changed by the function.

Examples

- *Example 1*: Set the host variable *END_OF_MONTH* with the last day of the current month.

```
SET :END_OF_MONTH = LAST_DAY(CURRENT_DATE);
```

  The host variable *END_OF_MONTH* is set with the value representing the end of the current month. If the current day is 2000-02-10, then *END_OF_MONTH* is set to 2000-02-29.

- *Example 2*: Set the host variable *END_OF_MONTH* with the last day of the month in EUR format for the given date.

```
SET :END_OF_MONTH = CHAR(LAST_DAY('1965-07-07'), EUR);
```

  The host variable *END_OF_MONTH* is set with the value '31.07.1965'.

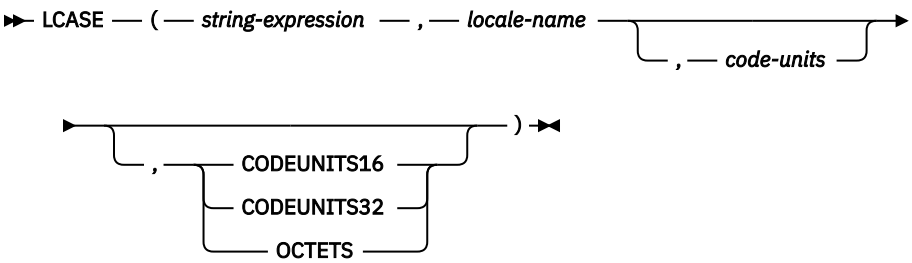
LCASE

The LCASE function returns a string in which all the SBCS characters of the input string have been converted to lowercase characters. The LCASE scalar function is a synonym for the LOWER scalar function.

The schema is SYSIBM.

LCASE (locale sensitive)

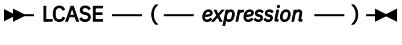
The LCASE function returns a string in which all characters have been converted to lowercase characters using the rules associated with the specified locale. The LCASE scalar function is a synonym for the LOWER scalar function.



The schema is SYSIBM.

LCASE (SYSFUN schema)

The LCASE function returns a string in which all SBCS characters have been converted to lowercase characters, that is, the characters A-Z have been converted to the characters a-z. Characters with diacritical marks are not converted. Consequently, a statement of the form LCASE(UCASE(*string*)) will not necessarily return the same result as LCASE(*string*).



The schema is SYSFUN.

expression
An expression that returns a built-in character string or Boolean value. In a Unicode database, the expression can also return a graphic string, in which case it is first converted to a character string before the function is evaluated. For a VARCHAR, the maximum length is 4000 bytes. For a CLOB, the maximum length is 1,048,576 bytes.

Result

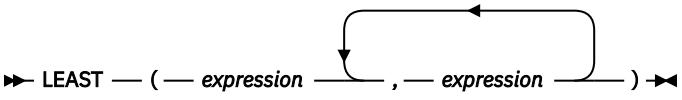
The data type of the result depends on the data type of the input expression:

- VARCHAR(4000) if the input expression is VARCHAR or CHAR
- CLOB(1M) if the input expression is CLOB or LONG VARCHAR

The result can be null; if the input expression is null, the result is the null value.

LEAST

The LEAST function returns the minimum value in a set of values.



The schema is SYSIBM.

expression
An expression that returns a value of any built-in data type or user-defined data type that is comparable with the data type of the other argument. The data type cannot be a LOB, distinct type base on a LOB, XML, array, cursor, row, or structured type.

Result

The result of the function is the smallest argument value. The result can be null if at least one argument can be null; the result is the null value if any argument is null.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by the data types of all the arguments as explained in "Rules for result data types".

Notes

- The LEAST scalar function is a synonym for the MIN scalar function.
- The LEAST function cannot be used as a source function when creating a user-defined function. Because this function accepts any comparable data types as arguments, it is not necessary to create additional signatures to support user-defined data types.

Example

Assume that table T1 contains three columns C1, C2, and C3 with values 1, 7, and 4, respectively. The query:

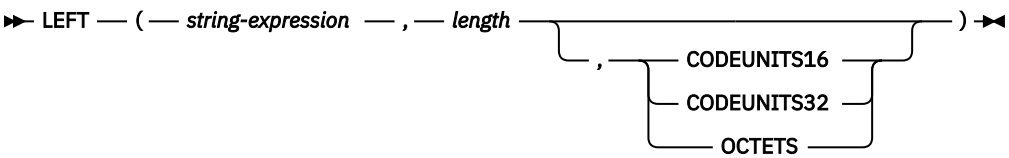
```
SELECT LEAST (C1, C2, C3) FROM T1
```

returns 1.

If column C3 has a value of null instead of 4, the same query returns the null value.

LEFT

The LEFT function returns the leftmost string of *string-expression* of length *length*, expressed in the specified string unit.



The schema is SYSIBM. The SYSFUN version of the LEFT function continues to be available.

string-expression
An expression that specifies the string from which the result is derived. The expression must return a built-in string, numeric value, Boolean value, or datetime value. If the value is not a string, it is implicitly cast to VARCHAR before the function is evaluated. Zero or more contiguous code points of the string comprise a substring of the string.

The *string-expression* is padded on the right with the necessary number of padding characters so that the specified substring of *string-expression* always exists. The character used for padding is the same character that is used to pad the string in contexts where padding would occur. For more information about padding, see "String assignments" in "Assignments and comparisons".

length

An expression that specifies the length of the result. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. *length* must be greater than or equal to 0 (SQLSTATE 22011). If OCTETS is specified and the result is graphic data, the value must be an even number (SQLSTATE 428GC).

If *length* is not a constant and a string unit is not specified, *length* must be less than or equal to the length attribute of *string-expression* (SQLSTATE 22011).

If *length* is not a constant and a string unit is specified, *length* must be less than or equal to the corresponding value from the following table (SQLSTATE 22011):

Table 70. Maximum value of length when length is not a constant and a string unit is specified

String unit of *string-expression*	**Specified string unit**	**Maximum value of *length*** **L = length attribute of *string-expression***
OCTETS	OCTETS	L
OCTETS	CODEUNITS16	L/2
OCTETS	CODEUNITS32	L/4
CODEUNITS16	OCTETS	L*2
CODEUNITS16	CODEUNITS16	L
CODEUNITS16	CODEUNITS32	L/2
CODEUNITS32	OCTETS	L*4
CODEUNITS32	CODEUNITS16	L*2
CODEUNITS32	CODEUNITS32	L

If *length* is a constant and the data type of *string-expression* is:

CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC
length must be less than or equal to 32,672 OCTETS, 16,336 CODEUNITS16 or 8168 CODEUNITS32 (SQLSTATE 22011).

CLOB or DBCLOB
length must be less than or equal to 2,147,483,647 OCTETS, 1,073,741,823 CODEUNITS16, or 53,6870,911 CODEUNITS32 (SQLSTATE 22011).

BLOB
length must be less than or equal to 2,147,483,647 OCTETS (SQLSTATE 22011).

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of *length*.

CODEUNITS16 specifies that *length* is expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *length* is expressed in 32-bit UTF-32 code units. OCTETS specifies that *length* is expressed in bytes.

If the string unit is specified as CODEUNITS16 or CODEUNITS32, and *string-expression* is a binary string or bit data, an error is returned (SQLSTATE 428GC). If the string unit is specified as OCTETS and *string-expression* is a graphic string, *length* must be an even number; otherwise, an error is returned (SQLSTATE 428GC). If a string unit is not explicitly specified, the string unit of *string-expression* determines the unit that is used. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

Result

The result of the function is a varying-length string that depends on the data type of *string-expression*:

- VARCHAR if *string-expression* is CHAR or VARCHAR
- CLOB if *string-expression* is CLOB
- VARGRAPHIC if *string-expression* is GRAPHIC or VARGRAPHIC
- DBCLOB if *string-expression* is DBCLOB
- VARBINARY if *string-expression* is BINARY or VARBINARY
- BLOB if *string-expression* is BLOB

The string unit of the result is the string unit of *string-expression*.The length attribute of the result depends on how *length* and string unit are specified:

- If *length* is not a constant, then the length attribute of the result is the same as the length attribute of *string-expression*.
- If *length* is a constant and a string unit is not specified, then the length attribute of the result is the maximum of length and the length attribute of *string-expression*.
- If *length* is a constant and a string unit is specified, then the length attribute of the result is shown in Table 2:

Table 71. Length attribute of result when length is a constant and a string unit is specified

String unit of *string-expression*	**Specified string unit**	**Maximum value of *length*** **L = length attribute of *string-expression***
OCTETS	OCTETS	max(L, *length*)
OCTETS	CODEUNITS16	max(L, *length* * 2)
OCTETS	CODEUNITS32	max(L, *length* * 4)
CODEUNITS16	OCTETS	max(L, *length* / 2)
CODEUNITS16	CODEUNITS16	max(L, *length*)
CODEUNITS16	CODEUNITS32	max(L, *length* * 2)
CODEUNITS32	OCTETS	max(L, *length* / 4)
CODEUNITS32	CODEUNITS16	max(L, *length* / 2)
CODEUNITS32	CODEUNITS32	max(L, *length*)

The actual length of the result (in string units) is *length*.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Assume that variable ALPHA has a value of "ABCDEF". The following statement:

```
SELECT LEFT(ALPHA,3)
  FROM SYSIBM.SYSDUMMY1
```

  returns "ABC", which are the three leftmost characters in ALPHA.

- *Example 2*: Assume that variable NAME, which is defined as VARCHAR(50), has a value of "KATIE AUSTIN", and that the integer variable FIRSTNAME_LEN has a value of 5. The following statement:

```
SELECT LEFT(NAME,FIRSTNAME_LEN)
  FROM SYSIBM.SYSDUMMY1
```

  returns the value "KATIE".

- *Example 3*: The following statement returns a zero-length string.

```
SELECT LEFT('ABCABC',0)
  FROM SYSIBM.SYSDUMMY1
```

- *Example 4*: The FIRSTNME column in the EMPLOYEE table is defined as VARCHAR(12). Find the first name of an employee whose last name is "BROWN" and return the first name in a 10-byte string.

```
SELECT LEFT(FIRSTNME, 10)
  FROM EMPLOYEE
  WHERE LASTNAME = 'BROWN'
```

  returns a VARCHAR(12) string that has the value "DAVID" followed by five blank characters.

- *Example 5*: In a Unicode database, FIRSTNAME is a VARCHAR(12) column. One of its values is the 6-character string "Jürgen". When FIRSTNAME has this value:

```
Function...                      Returns...

LEFT(FIRSTNAME,2,CODEUNITS32)   'Jü' -- x'4AC3BC'
LEFT(FIRSTNAME,2,CODEUNITS16)   'Jü' -- x'4AC3BC'
LEFT(FIRSTNAME,2,OCTETS)        'J'  -- x'4A20', a truncated string
```

- *Example 6*: The following example works with the Unicode string "&N~AB", where "&" is the musical symbol G clef character, and "~" is the combining tilde character. This string is shown in different Unicode encoding forms in the following example:

	"&"	"N"	"~"	"A"	"B"
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'

  Assume that the variable UTF8_VAR, with a length attribute of 20 bytes, contains the UTF-8 representation of the string.

```
SELECT LEFT(UTF8_VAR, 2, CODEUNITS16),
  LEFT(UTF8_VAR, 2, CODEUNITS32),
  LEFT(UTF8_VAR, 2, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

  returns the values "&", "&N", and "bb", respectively, where "b" represents the blank character.

```
SELECT LEFT(UTF8_VAR, 5, CODEUNITS16),
  LEFT(UTF8_VAR, 5, CODEUNITS32),
  LEFT(UTF8_VAR, 5, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

  returns the values "&N~A", "&N~AB", and "&N", respectively.

```
SELECT LEFT(UTF8_VAR, 10, CODEUNITS16),
  LEFT(UTF8_VAR, 10, CODEUNITS32),
```

```
    LEFT(UTF8_VAR, 10, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values "&N~ABbbbb", "&N~ABbbbbb", and "&N~ABb", respectively, where "b" represents the blank character.

Assume that the variable UTF16_VAR, with a length attribute of 20 code units, contains the UTF-16BE representation of the string.

```
SELECT LEFT(UTF16_VAR, 2, CODEUNITS16),
  LEFT(UTF16_VAR, 2, CODEUNITS32),
  HEX (LEFT(UTF16_VAR, 2, OCTETS))
FROM SYSIBM.SYSDUMMY1
```

returns the values "&", "&N", and X'D834', respectively, where X'D834' is an unmatched high surrogate.

```
SELECT LEFT(UTF16_VAR, 5, CODEUNITS16),
  LEFT(UTF16_VAR, 5, CODEUNITS32),
  LEFT(UTF16_VAR, 6, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

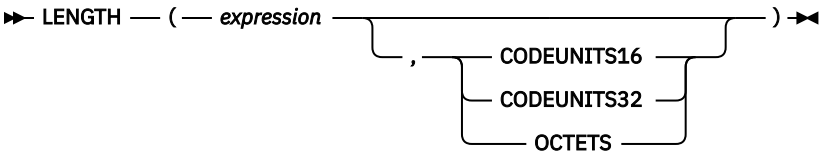
returns the values "&N~A", "&N~AB", and "&N", respectively.

```
SELECT LEFT(UTF16_VAR, 10, CODEUNITS16),
  LEFT(UTF16_VAR, 10, CODEUNITS32),
  LEFT(UTF16_VAR, 10, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values "&N~ABbbbb", "&N~ABbbbbb", and "&N~A", respectively, where "b" represents the blank character.

LENGTH

The LENGTH function returns the length of *expression* in the implicit or explicit string unit.



The schema is SYSIBM.

expression
An expression that returns a value that is a built-in data type. If *expression* can be null, the result can be null; if *expression* is null, the result is the null value.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of the result. CODEUNITS16 specifies that the result is to be expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that the result is to be expressed in 32-bit UTF-32 code units. OCTETS specifies that the result is to be expressed in bytes.

If a string unit is explicitly specified, and if *expression* is not string data, an error is returned (SQLSTATE 428F5). If a string unit is specified as CODEUNITS16 or CODEUNITS32, and *expression* is a binary string or bit data, an error is returned (SQLSTATE 428GC). If a string unit is specified as OCTETS and *expression* is a binary string, an error is returned (SQLSTATE 42815). For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

If a string unit argument is not explicitly specified and if *expression* is a character or graphic string, the string units of *expression* determines the string unit that is used for the result. Otherwise, the value returned specifies the length in bytes.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The length of character and graphic strings includes trailing blanks. The length of binary strings includes binary zeros. The length of varying-length strings is the actual length and not the maximum length. The length of all other values is the number of bytes used to represent the value:

- 2 for small integer (SMALLINT)
- 4 for large integer (INTEGER)
- 8 for big integer (BIGINT)
- (*p*/2)+1 for decimal numbers with precision *p*
- 8 for DECFLOAT(16)
- 16 for DECFLOAT(34)
- The length of the string for binary strings
- The length of the string for character strings
- 4 for single-precision floating-point
- 8 for double-precision floating-point
- 4 for DATE
- 3 for TIME
- 7+(*p*+1)/2 for TIMESTAMP(*p*)

Examples

- *Example 1*: Assume that the host variable ADDRESS is a varying-length character string with a value of '895 Don Mills Road'.

```
LENGTH(:ADDRESS)
```

  returns the value 18.

- *Example 2*: Assume that START_DATE is a column of type DATE.

```
LENGTH(START_DATE)
```

  returns the value 4.

- *Example 3*: The following example returns the value 10.

```
LENGTH(CHAR(START_DATE, EUR))
```

- *Example 4*: The following examples work with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown in different Unicode encoding forms in the following example:

	'&'	'N'	'~'	'A'	'B'
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'
UTF-32BE	X'0001D11E'	X'0000004E'	X'00000303'	X'00000041'	X'00000042'

  Assume that the variable UTF8_VAR contains the UTF-8 representation of the string.

```
SELECT LENGTH(UTF8_VAR, CODEUNITS16),
  LENGTH(UTF8_VAR, CODEUNITS32),
  LENGTH(UTF8_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

  returns the values 6, 5, and 9, respectively.

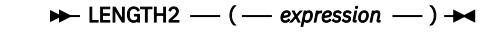
  Assume that the variable UTF16_VAR contains the UTF-16BE representation of the string.

```
SELECT LENGTH(UTF16_VAR, CODEUNITS16),
   LENGTH(UTF16_VAR, CODEUNITS32),
   LENGTH(UTF16_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 6, 5, and 12, respectively.

LENGTH2

The LENGTH2 function returns the length of *expression* in 16-bit UTF-16 string units (CODEUNITS16).



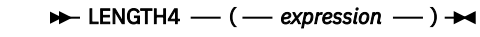
The schema is SYSIBM.

expression
An expression that returns a value that is a built-in data type.

The LENGTH2 scalar function invoked with a character or graphic string as the argument is equivalent to invoking the LENGTH function with CODEUNITS16 specified. The LENGTH2 scalar function invoked with any other data type as the argument is equivalent to invoking the LENGTH function without a string units argument.

LENGTH4

The LENGTH4 function returns the length of *expression* in 32-bit UTF-32 string units (CODEUNITS32).



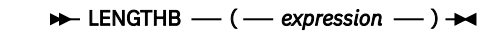
The schema is SYSIBM.

expression
An expression that returns a value that is a built-in data type.

The LENGTH4 scalar function invoked with a character or graphic string as the argument is equivalent to invoking the LENGTH function with CODEUNITS32 specified. The LENGTH4 scalar function invoked with any other data type as the argument is equivalent to invoking the LENGTH function without a string units argument.

LENGTHB

The LENGTHB function returns the length of *expression* in bytes.



The schema is SYSIBM.

expression
An expression that returns a value that is a built-in data type.

The LENGTHB scalar function invoked with a character or graphic string as the argument is equivalent to invoking the LENGTH function with OCTETS specified. The LENGTHB scalar function invoked with any other data type as the argument is equivalent to invoking the LENGTH function without a string units argument.

LN

The LN function returns the natural logarithm of a number. The LN and EXP functions are inverse operations.

The schema is SYSIBM. (The SYSFUN version of the LN function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type. If the value is of decimal floating-point data type, the operation is performed in decimal floating-point; otherwise, the value is converted to double-precision floating-point for processing by the function. The value of the argument must be greater than zero (SQLSTATE 22003).

If the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*); otherwise, the result is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Notes

- ***Results involving DECFLOAT special values:*** For decimal floating-point values, the special values are treated as follows:
  - LN(NaN) returns NaN.
  - LN(-NaN) returns -NaN.
  - LN(Infinity) returns Infinity.
  - LN(-Infinity) returns NaN and a warning.
  - LN(sNaN) returns NaN and a warning.
  - LN(-sNaN) returns -NaN and a warning.
  - LN(DECFLOAT('0') returns -Infinity.
- ***Syntax alternatives:*** For compatibility with other SQL dialects, LOG can be specified in place of LN. However, because some database managers and applications use LOG to mean a common logarithm rather than a natural logarithm, use LN instead of LOG whenever possible.

Example

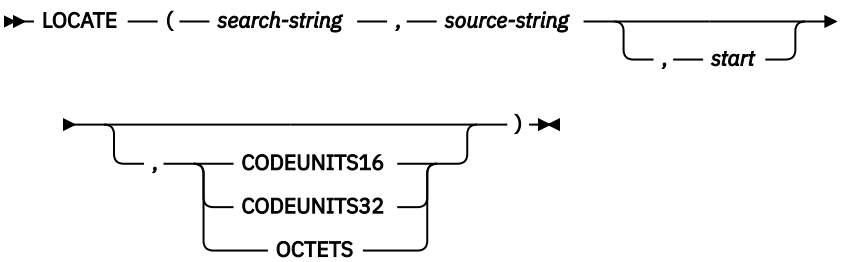
Assume that NATLOG is a DECIMAL(4,2) host variable with a value of 31.62.

```
VALUES LN(:NATLOG)
```

Returns the approximate value 3.45.

LOCATE

The LOCATE function returns the starting position of the first occurrence of one string (called the *search-string*) within another string (called the *source-string*).



The schema is SYSIBM. The SYSFUN version of the LOCATE function continues to be available, but it is not sensitive to the database collation.

If the *search-string* is not found and neither argument is null, the result is zero. If the *search-string* is found, the result is a number from 1 to the actual length of the *source-string*. The search is done using the collation of the database, unless *search-string* or *source-string* is defined as a binary string or as FOR BIT DATA , in which case the search is done using a binary comparison.

If the optional *start* is specified, it indicates the character position in the *source-string* at which the search is to begin. An optional string unit can be specified to indicate in what units the *start* and result of the function are expressed.

If the *search-string* has a length of zero, the result returned by the function is 1. Otherwise, if the *source-string* has a length of zero, the result returned by the function is 0. Otherwise:

- If the value of *search-string* is equal to an identical length of substring of contiguous positions within the value of *source-string*, the result returned by the function is the starting position of the first such substring within the *source-string* value.
- Otherwise, the result returned by the function is 0.

search-string
An expression that specifies the string that is the object of the search. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, binary string, numeric, Boolean, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, or binary string data type, it is implicitly cast to VARCHAR before evaluating the function. The expression cannot be specified by a LOB file reference variable.

source-string
An expression that specifies the string in which the search is to take place. The expression must return a value that is a built-in string, numeric, Boolean, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function.

start
An expression that specifies the position within *source-string* at which the search is to start. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value of the integer must be greater than or equal to zero. If *start* is specified, the LOCATE function is similar to:

```
POSITION(search-string,
  SUBSTRING(source-string, start, string-unit),
  string-unit) + start - 1
```

where *string-unit* is either CODEUNITS16, CODEUNITS32, or OCTETS.

If *start* is not specified, the search begins at the first position of the source string, and the LOCATE function is similar to:

```
POSITION(search-string, source-string, string-unit)
```

If OCTETS is specified and *source-string* is graphic data, the value of the integer must be odd (SQLSTATE 428GC).

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of *start* and the result. CODEUNITS16 specifies that *start* and the result are to be expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *start* and the result are to be expressed in 32-bit UTF-32 code units. OCTETS specifies that *start* and the result are to be expressed in bytes.

If a string unit is specified as CODEUNITS16 or CODEUNITS32, and *search-string* or *source-string* is a binary string or FOR BIT DATA, an error is returned (SQLSTATE 428GC). If the string unit is specified as CODEUNITS16 or OCTETS, and the string units of *source-string* is CODEUNITS32, an error is returned (SQLSTATE 428GC).

If a string unit is not explicitly specified and if *source-string* is a character or graphic string, the string units of *source-string* determines the unit that is used for the result and for *start* (if specified). Otherwise, they are expressed in bytes.

If a locale-sensitive UCA-based collation is used for this function, then the CODEUNITS16 option offers the best performance characteristics.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

The first and second arguments must have compatible string types. For more information about compatibility, see "Rules for string conversions". In a Unicode database, if one string argument is character (not FOR BIT DATA) and the other string argument is graphic, then the *search-string* is converted to the data type of the *source-string* for processing. If one argument is character FOR BIT DATA, the other argument must not be graphic (SQLSTATE 42846).

The result of the function is a large integer. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Find the location of the first occurrence of the character "N" in the string "DINING".

```
SELECT LOCATE('N', 'DINING')
FROM SYSIBM.SYSDUMMY1
```

  The result is the value 3.

- *Example 2*: For all the rows in the table named IN_TRAY, select the RECEIVED column, the SUBJECT column, and the starting position of the string "GOOD" within the NOTE_TEXT column.

```
SELECT RECEIVED, SUBJECT, LOCATE('GOOD', NOTE_TEXT)
FROM IN_TRAY
WHERE LOCATE('GOOD', NOTE_TEXT) <> 0
```

- *Example 3*: Locate the character "ß" in the string "Jürgen lives on Hegelstraße", and set the host variable LOCATION with the position, as measured in CODEUNITS32 units, within the string.

```
SET :LOCATION = LOCATE('ß', 'Jürgen lives on Hegelstraße', 1, CODEUNITS32)
```

  The value of host variable LOCATION is set to 26.

- *Example 4*: Locate the character '"ß" in the string "Jürgen lives on Hegelstraße", and set the host variable LOCATION with the position, as measured in CODEUNITS16 units, within the string.

```
SET :LOCATION = LOCATE('ß', 'Jürgen lives on Hegelstraße', 1, CODEUNITS16)
```

  The value of host variable LOCATION is set to 26.

- *Example 5*: Locate the character "ß" in the string "Jürgen lives on Hegelstraße", and set the host variable LOCATION with the position, as measured in OCTETS, within the string.

```
SET :LOCATION = LOCATE('ß', 'Jürgen lives on Hegelstraße', 1, OCTETS)
```

  The value of host variable LOCATION is set to 27.

- *Example 6*: The following examples work with the Unicode string "&N~AB", where "&" is the musical symbol G clef character, and "~" is the non-spacing combining tilde character. This string is shown in different Unicode encoding forms in the following example:

	"&"	"N"	"~"	"A"	"B"
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'

  Assume that the variable UTF8_VAR contains the UTF-8 representation of the string.

```
SELECT LOCATE('~', UTF8_VAR, CODEUNITS16),
  LOCATE('~', UTF8_VAR, CODEUNITS32),
  LOCATE('~', UTF8_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

  returns the values 4, 3, and 6, respectively.

Assume that the variable UTF16_VAR contains the UTF-16BE representation of the string.

```
SELECT LOCATE('~', UTF16_VAR, CODEUNITS16),
   LOCATE('~', UTF16_VAR, CODEUNITS32),
   LOCATE('~', UTF16_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 4, 3, and 7, respectively.

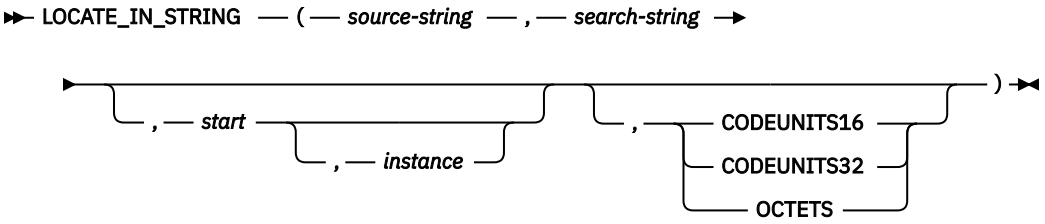
- *Example 7*: In a Unicode database created with the case insensitive collation CLDR181_LEN_S1, find the position of the word "Brown" in the phrase "The quick brown fox".

```
SET :LOCATION = LOCATE('Brown', 'The quick brown fox', CODEUNITS16)
```

The value of the host variable LOCATION is set to 11.

LOCATE_IN_STRING

The LOCATE_IN_STRING function returns the starting position of a string (called the *search-string*) within another string (called the *source-string*).



The schema is SYSIBM

If the *search-string* is not found and neither argument is null, the result is zero. If the *search-string* is found, the result is a number from 1 to the actual length of the *source-string*. The search is done using the collation of the database, unless *search-string* or *source-string* is defined as a binary string or as FOR BIT DATA, in which case the search is done using a binary comparison.

If the optional *start* is specified, it indicates the character position in the *source-string* at which the search is to begin. If the *start* is specified, an instance number can also be specified. The *instance* argument is used to determine the position of a specific occurrence of *search-string* within *source-string*. An optional string unit can be specified to indicate in what units the *start* and result of the function are expressed.

If the *search-string* has a length of zero, the result returned by the function is 1. If the *source-string* has a length of zero, the result returned by the function is 0. If neither condition exists, and if the value of *search-string* is equal to an identical length of a substring of contiguous positions within the value of *source-string*, the result returned by the function is the starting position of that substring within the *source-string* value; otherwise, the result returned by the function is 0.

source-string
An expression that specifies the string in which the search is to take place. The expression must return a value that is a built-in string, numeric, Boolean, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function.

search-string
An expression that specifies the string that is the object of the search. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, binary string, numeric, Boolean, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, or binary string data type, it is implicitly cast to VARCHAR before evaluating the function. The expression cannot be specified by a LOB file reference variable.

start
An expression that specifies the position within *source-string* at which the search for a match is to start. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC or

VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

If the value of the integer is greater than zero, the search begins at *start* and continues for each position to the end of the string. If the value of the integer is less than zero, the search begins at LENGTH(*source-string*) + *start* + 1 and continues for each position to the beginning of the string.

If *start* is not specified, the default is 1. If OCTETS is specified and *source-string* is graphic data, the value of the integer must be odd (SQLSTATE 428GC). If the value of the integer is zero, an error is returned (SQLSTATE 42815).

instance
An expression that specifies which instance of *search-string* to search for within *source-string*. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. If *instance* is not specified, the default is 1. The value of the integer must be greater than or equal to 1 (SQLSTATE 42815).

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of *start* and the result. CODEUNITS16 specifies that *start* and the result are to be expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *start* and the result are to be expressed in 32-bit UTF-32 code units. OCTETS specifies that *start* and the result are to be expressed in bytes.

If a string unit is specified as CODEUNITS16 or CODEUNITS32, and *search-string* or *source-string* is a binary string or FOR BIT DATA, an error is returned (SQLSTATE 428GC). If the string unit is specified as CODEUNITS16 or OCTETS, and the string units of *source-string* is CODEUNITS32, an error is returned (SQLSTATE 428GC).

If a string unit is not explicitly specified and if *source-string* is a character or graphic string, the string units of *source-string* determines the unit that is used for the result and for *start* (if specified). Otherwise, they are expressed in bytes.

If a locale-sensitive UCA-based collation is used for this function, then the CODEUNITS16 option offers the best performance characteristics.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

The first and second arguments must have compatible string types. For more information about compatibility, see "Rules for string conversions". In a Unicode database, if one string argument is character (not FOR BIT DATA) and the other string argument is graphic, then the *search-string* is converted to the data type of the *source-string* for processing. If one argument is character FOR BIT DATA, the other argument must not be graphic (SQLSTATE 42846).

At each search position, a match is found when the substring at that position and LENGTH(*search-string*) - 1 values to the right of the search position in *source-string*, is equal to *search-string*.

The result of the function is a large integer. The result is the starting position of the instance of *search-string* within *source-string*. The value is relative to the beginning of the string (regardless of the specification of *start*). If any argument can be null, the result can be null; if any argument is null, the result is the null value.

INSTR can be used as a synonym for LOCATE_IN_STRING.

The INSTRB scalar function is equivalent to invoking the LOCATE_IN_STRING function with OCTETS (where allowed) specified to indicate that start position and the result are expressed in bytes.

Examples

- *Example 1*: Locate the character "ß" in the string "Jürgen lives on Hegelstraße" by searching from the end of the string, and set the host variable POSITION with the position, as measured in CODEUNITS32 units, within the string.

```
SET :POSITION = LOCATE_IN_STRING('Jürgen lives on Hegelstraße',
                                 'ß',-1,CODEUNITS32);
```

  The value of host variable POSITION is set to 26.

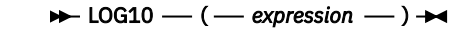
- *Example 2*: Find the location of the third occurrence of the character "N" in the string "WINNING" by searching from the start of the string and then set the host variable POSITION with the position of the character, as measured in bytes, within the string.

```
SET :POSITION =
LOCATE_IN_STRING('WINNING','N',1,3,OCTETS);
```

  The value of host variable POSITION is set to 6.

LOG10

The LOG10 function returns the common logarithm (base 10) of a number.



The schema is SYSIBM. (The SYSFUN version of the LOG10 function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type. If the value is of decimal floating-point data type, the operation is performed in decimal floating-point; otherwise, the value is converted to double-precision floating-point for processing by the function. The value of the argument must be greater than zero (SQLSTATE 22003).

If the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*); otherwise, the result is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Notes

- ***Results involving DECFLOAT special values:*** For decimal floating-point values, the special values are treated as follows:
  - LOG10(NaN) returns NaN.
  - LOG10(-NaN) returns -NaN.
  - LOG10(Infinity) returns Infinity.
  - LOG10(-Infinity) returns NaN and a warning.
  - LOG10(sNaN) returns NaN and a warning.
  - LOG10(-sNaN) returns -NaN and a warning.
  - LOG10(DECFLOAT('0') returns -Infinity.

Example

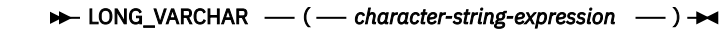
Assume that L is a DECIMAL(4,2) host variable with a value of 31.62.

```
VALUES LOG10(:L)
```

Returns the DOUBLE value +1.49996186559619E+000.

LONG_VARCHAR

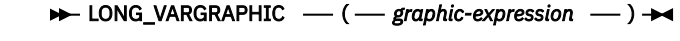
The LONG_VARCHAR function is deprecated and might be removed in a future release.



The function is compatible with earlier Db2 versions.

LONG_VARGRAPHIC

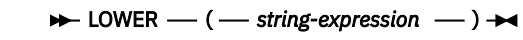
The LONG_VARGRAPHIC function is deprecated and might be removed in a future release.



The function is compatible with earlier Db2 versions.

LOWER

The LOWER function returns a string in which all the SBCS characters have been converted to lowercase characters.



The schema is SYSIBM. (The SYSFUN version of this function continues to be available with support for CLOB arguments.)

string-expression
An expression that returns a built-in character string or Boolean value. In a Unicode database, the expression can also return a graphic string, in which case it is first converted to a character string before the function is evaluated.

Result

The characters A-Z are converted to the characters a-z, and other characters are converted to their lowercase equivalents, if they exist. For example, in code page 850, É maps to é. If the code point length of the result character is not the same as the code point length of the source character, the source character is not converted. Because not all characters are converted, LOWER(UPPER(*string-expression*)) does not necessarily return the same result as LOWER(*string-expression*).

The result of the function has the same data type, string unit, and length attribute as the argument. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

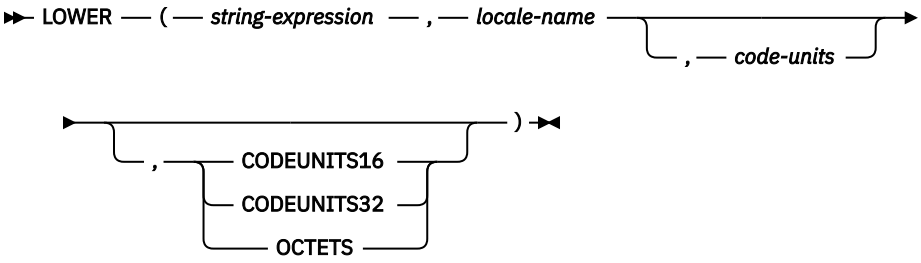
Ensure that the characters in the value of column JOB in the EMPLOYEE table are returned in lowercase characters.

```
SELECT LOWER(JOB)
  FROM EMPLOYEE
  WHERE EMPNO = '000020';
```

The result is the value 'manager'.

LOWER (locale sensitive)

The LOWER function returns a string in which all characters have been converted to lowercase characters using the rules associated with the specified locale.



The schema is SYSIBM.

string-expression
An expression that returns a built-in character string, , Boolean value, or graphic string. If the expression returns a character string, the expression cannot specify FOR BIT DATA (SQLSTATE 42815).

locale-name
A character constant that specifies the locale that defines the rules for conversion to lowercase characters. This value is not case sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming, see "Locale names for SQL and XQuery".

Attention: In the container-only release of Db2 Version 11.5 Mod Pack 1 and later versions, passing UNI_SIMPLE as locale-name will enable use of simple case folding mapping.

code-units
An integer constant that specifies the number of code units in the result. If specified, this must be an integer between:

- 1 and 32672, if the string unit of the result is OCTETS
- 1 and 16336, if the string unit of the result is double bytes or CODEUNITS16
- 1 and 8168, if the string unit of the result is CODEUNITS32

Otherwise, an error is returned (SQLSTATE 42815). The default is the length attribute of *string-expression*.

The value that can be specified for *code-units* depends on which string units are used:

- If the string unit OCTETS is specified for the string expression and the result is graphic data, the value must be even (SQLSTATE 428GC).
- If the string unit OCTETS is specified for the string expression and the string unit of the result is CODEUNIT32, the value must be a multiple of 4 (SQLSTATE 428GC).
- If the string unit CODEUNITS16 is specified for the string expression and the string unit of the result is CODEUNITS32, the value must be a multiple of 2 (SQLSTATE 428GC).

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of *code-units*.

CODEUNITS16 specifies that *code-units* is expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *code-units* is expressed in 32-bit UTF-32 code units. OCTETS specifies that *code-units* is expressed in bytes.

If a string unit is not explicitly specified, the string unit of *string-expression* determines the unit that is used. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

Result

The result of the function is VARCHAR if *string-expression* is CHAR or VARCHAR, and VARGRAPHIC if *string-expression* is GRAPHIC or VARGRAPHIC. The string units of the result is the same as the string units of *string-expression*.

The length attribute of the result is determined by the implicit or explicit value of *code-units*, the implicit or explicit string unit, the result data type, and the result string units, as shown in the following table:

Table 72. Length attribute of the result of LOWER as a function of string unit and result type

Data type and string units of result	**Length attribute for *code-units* in CODEUNITS16**	**Length attribute for *code-units* in CODEUNITS32**	**Length attribute for *code-units* in OCTETS**
VARCHAR in OCTETS	MIN(*code-units* * 3, 32672)	MIN(*code-units* * 4, 32672)	*code-units*
VARCHAR in CODEUNITS32	MIN(*code-units* / 2, 8168)	MIN(*code-units*, 8168)	MIN(*code-units* / 4, 8168)
VARGRAPHIC in CODEUNITS16 or double bytes	*code-units*	MIN(*code-units* * 2, 16336)	MIN(*code-units* / 2, 16336)
VARGRAPHIC in CODEUNITS32	MIN(*code-units* / 2, 8168)	MIN(*code-units*, 8168)	MIN(*code-units* / 4, 8168)

The actual length of the result might be greater than the length of *string-expression*. If the actual length of the result is greater than the length attribute of the result, an error is returned (SQLSTATE 42815). If the number of code units in the result exceeds the value of *code-units*, an error is returned (SQLSTATE 42815).

If *string-expression* is not in UTF-16, this function performs code page conversion of *string-expression* to UTF-16, and of the result from UTF-16 to the code page of *string-expression*. If either code page conversion results in at least one substitution character, the result includes the substitution character, a warning is returned (SQLSTATE 01517), and the warning flag SQLWARN8 in the SQLCA is set to 'W'.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Examples

- *Example 1*: Ensure that the characters in the value of column JOB in the EMPLOYEE table are returned in lowercase characters.

```
SELECT LOWER(JOB, 'en_US')
  FROM EMPLOYEE
  WHERE EMPNO = '000020'
```

  The result is the value 'manager'.

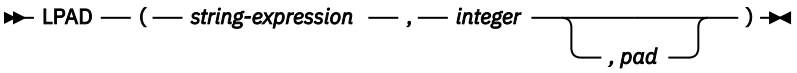
- *Example 2*: Find the lowercase characters for all the 'I' characters in a Turkish string.

```
VALUES LOWER('IİıI', 'tr_TR', CODEUNITS16)
```

  The result is the string 'ıiiı'.

LPAD

The LPAD function pads a string on the left with a specified character string or with blanks.



The schema is SYSIBM.

The LPAD function treats leading or trailing blanks in the string expression as significant. Padding will only occur if the actual length of *string-expression* is less than *integer*, and *pad* is not an empty string.

string-expression
An expression that specifies the source string. The expression must return a built-in character string, graphic string, binary string, CLOB or DBCLOB value, numeric value, Boolean value, or datetime value. If the value is:

- A CLOB, numeric, Boolean, or datetime value, it is implicitly cast to VARCHAR before the function is evaluated
- A DBCLOB value, it is implicitly cast to VARGRAPHIC before the function is evaluated

The data type of the value cannot be a BLOB (SQLSTATE 42815).

integer
An expression that specifies the actual length of the result in the string units of the string expression. The expression must return a built-in numeric value, Boolean value, or character string. In a Unicode database, the expression can also return a graphic string, in which case it is first converted to a character string before the function is evaluated. If the value returned by the expression is not an integer, it is cast to INTEGER before the function is evaluated. The value must be zero or a positive integer that is less than or equal to the maximum length for the result data type in the units of the string expression.

pad
An expression that specifies the string with which to pad. The expression must return a built-in character string, graphic string, binary string, CLOB or DBCLOB value, numeric value, Boolean value, or datetime value. If the value is:

- A CLOB, numeric, Boolean, or datetime value, it is implicitly cast to VARCHAR before the function is evaluated
- A DBCLOB value, it is implicitly cast to VARGRAPHIC before the function is evaluated

The data type of *pad* cannot be a BLOB (SQLSTATE 42815).

The data type of the string expression determines the default pad string:

- The SBCS blank character, if the string expression is a character string.
- The ideographic blank character, if the string expression is a graphic string. For graphic string in an EUC database, X'3000' is used. For graphic string in a Unicode database, X'0020' is used.
- Hexadecimal zero (X'00'), if the string expression is a binary string.

Result

The data type of the result depends on the data type of the string expression:

- VARCHAR if the data type is VARCHAR or CHAR
- VARGRAPHIC if the data type is VARGRAPHIC or GRAPHIC
- VARBINARY if the data type is VARBINARY or BINARY

The result of the function is a varying length string that has the same string unit and code page as the string expression. The values for the string expression and the pad expression must have compatible data types. If the string expression and pad expression have different code pages, then the pad expression is

converted to the code page of the string expression. If either the string expression or the pad expression is FOR BIT DATA or a binary string, no character conversion occurs.

The length attribute of the result depends on whether the value for *integer* is available when the SQL statement containing the function invocation is compiled (for example, if it is specified as a constant or a constant expression) or available only when the function is executed (for example, if it is specified as the result of invoking a function). When the value is available when the SQL statement containing the function invocation is compiled, if *integer* is greater than zero, the length attribute of the result is *integer*. If *integer* is 0, the length attribute of the result is 1. When the value is available only when the function is executed, the length attribute of the result is determined according to the following table:

Table 73. Determining the result length when integer is available only when the function is executed

Data type of *string-expression*	**Result data type length**
CHAR(*n*), VARCHAR(*n*), BINARY(*n*), or VARBINARY(*n*)	Minimum of *n*+100 and 32 672
GRAPHIC(*n*) or VARGRAPHIC(*n*)	Minimum of *n*+100 and 16 336
CHAR(n) or VARCHAR(*n*) or GRAPHIC(*n*) or VARGRAPHIC(*n*) with string units of CODEUNITS32 (Unicode database only)	Minimum of *n*+100 and 8 168

The actual length of the result is determined from *integer*. If *integer* is 0 the actual length is 0, and the result is the empty result string. If *integer* is less than the actual length of *string-expression*, the actual length is *integer* and the result is truncated.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Assume that NAME is a VARCHAR(15) column that contains the values "Chris", "Meg", and "Jeff". The following query will completely pad out a value on the left with periods:

```
SELECT LPAD(NAME,15,'.' ) AS NAME FROM T1;
```

  returns:

```
NAME
---------------
.........Chris
............Meg
...........Jeff
```

- *Example 2*: Assume that NAME is a VARCHAR(15) column that contains the values "Chris", "Meg", and "Jeff". The following query will only pad each value to a length of 5:

```
SELECT LPAD(NAME,5,'.' ) AS NAME FROM T1;
```

  returns:

```
NAME
-----
Chris
..Meg
.Jeff
```

- *Example 3*: Assume that NAME is a CHAR(15) column containing the values "Chris", "Meg", and "Jeff". The LPAD function does not pad because NAME is a fixed length character field and is blank padded already. However, since the length of the result is 5, the columns are truncated:

```
SELECT LPAD(NAME,5,'.' ) AS NAME FROM T1;
```

  returns:

```
NAME
-----
Chris
Meg
Jeff
```

- *Example 4*: Assume that NAME is a VARCHAR(15) column containing the values "Chris", "Meg", and "Jeff". In some cases, a partial instance of the pad specification is returned:

```
SELECT LPAD(NAME,15,'123' ) AS NAME FROM T1;
```

returns:

```
NAME
---------------
1231231231Chris
123123123123Meg
12312312312Jeff
```

- *Example 5*: Assume that NAME is a VARCHAR(15) column containing the values "Chris", "Meg", and "Jeff". Note that "Chris" is truncated, "Meg" is padded, and "Jeff" is unchanged:

```
SELECT LPAD(NAME,4,'.' ) AS NAME FROM T1;
```

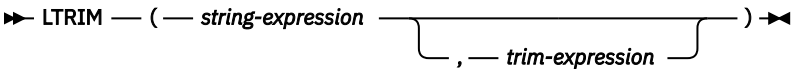
returns:

```
NAME
----
Chri
.Meg
Jeff
```

LTRIM

The LTRIM function removes any of the specified characters from the beginning of a string.

The character search compares the binary representation of each character (consisting of one or more bytes) in the trim expression to the binary representation of each character (consisting of one or more bytes) at the beginning of the string expression. The database collation does not affect the search. If the string expression is defined as FOR BIT DATA or is a binary string, the search compares each byte in the trim expression to the byte at the beginning of the string expression.



The schema is SYSIBM. (The SYSFUN version of this function is also available. That version uses a single parameter, removes leading blanks only, and accepts CLOB arguments.)

string-expression

An expression that specifies the string to be trimmed.

- If only one argument is specified, the expression must return a built-in character string, graphic string, binary string, CLOB or DBCLOB value, numeric value, Boolean value, or datetime value. If the value is:
  - A CLOB, numeric, Boolean, or datetime value, it is implicitly cast to VARCHAR before the function is evaluated
  - A DBCLOB value, it is implicitly cast to VARGRAPHIC before the function is evaluated

  The data type of the value cannot be a BLOB (SQLSTATE 42815).
- If both arguments are specified, the expression must return a value that is a built-in character string, numeric value, Boolean value, or datetime value. If the data type of the value is numeric or datetime, the value is implicitly cast to VARCHAR before the function is evaluated. The actual length of a CLOB value is limited to the maximum size of a VARCHAR data type (SQLSTATE 22001). The

actual length of a BLOB value is limited to the maximum size of a VARBINARY (SQLSTATE 22001). The actual length of a DBCLOB value is limited to the maximum size of a VARGRAPHIC data type (SQLSTATE 22001).

trim-expression

An expression that specifies the characters that are to be removed from the beginning of a string expression. The expression must return a built-in character string, numeric value, Boolean value, or datetime value.

- If the data type of the trim expression is not a string, then the value is implicitly cast to VARCHAR before the function is evaluated.
- If the data type of the trim expression is a CLOB, then the actual length of the value is limited to the maximum size of a VARCHAR (SQLSTATE 22001).
- If the data type of the trim expression is a DBCLOB, then the actual length of the value is limited to the maximum size of a VARGRAPHIC (SQLSTATE 22001).
- If the data type of the trim expression is BLOB, then the actual length of the value is limited to the maximum size of a VARBINARY (SQLSTATE 22001).
- If the string expression is not defined as FOR BIT DATA, then the trim expression cannot be defined as FOR BIT DATA (SQLSTATE 42815).

The data type of the string expression determines the default trim expression:

- A double byte blank, if the string expression is a graphic string in a DBCS or EUC database
- A UCS-2 blank, if the string expression is a graphic string in a Unicode database
- The value X'20', if the string expression is a FOR BIT DATA string
- The value X'00', if the string expression is a binary string
- A single-byte blank in all other cases

The values for the string expression and trim expression must have compatible data types. If one function argument is character FOR BIT DATA, then the other argument cannot be a graphic (SQLSTATE 42846). A combination of character string and graphic string arguments can be used only in a Unicode database (SQLSTATE 42815).

Result

The data type of the result depends on the data type of the string expression:

- VARCHAR if the data type is VARCHAR or CHAR
- CLOB if the data type is CLOB
- VARBINARY if the data type is VARBINARY or BINARY
- BLOB if the data type is BLOB
- VARGRAPHIC if the data type is VARGRAPHIC or GRAPHIC
- DBCLOB if the data type is DBCLOB

The length attribute of the result data type is the same as the length attribute of the data type of the string expression.

The actual length of the result is the length of *string-expression* minus the number of string units removed. If all of the characters are removed, the result is an empty string with a length of zero.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Use the LTRIM function when the host variable HELLO is defined as CHAR(6) and has a value of " Hello".

```
VALUES LTRIM(:HELLO)
```

  The result is 'Hello'. When a *trim-expression* is not specified only blanks are removed.

- *Example 2*: Use the LTRIM function to remove the characters specified in the *trim-expression* from the beginning of the *string-expression*.

```
VALUES LTRIM('...$V..$AR', '$.')
```

  The result is `'V..$AR'`. The function stops when it encounters a character not in the *trim-expression*.

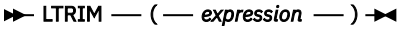
- *Example 3*: Use the LTRIM function to remove the characters specified in the *trim-expression* from the beginning of the *string-expression*

```
VALUES LTRIM('[[ -78]]', '- []')
```

  The result is `'78]]'`. When removing characters and blanks, you must include a blank in the *trim-expression*.

LTRIM (SYSFUN schema)

Returns the characters of the argument with leading blanks removed.



The schema is SYSFUN.

expression
An expression that returns the built-in character string or Boolean value that is to be trimmed. The maximum length is:

- 4000 bytes for a VARCHAR
- 1,048,576 bytes for a CLOB

Result

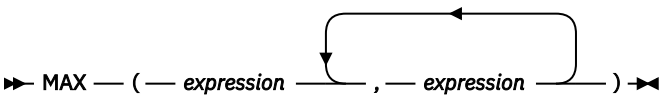
The data type of the result is:

- VARCHAR(4000) if the data type of the expression is CHAR or VARCHAR
- CLOB(1M) if the data type of the expression is CLOB or LONG VARCHAR

The result can be null; if the expression is null, the result is the null value.

MAX

The MAX function returns the maximum value in a set of values.



The schema is SYSIBM.

expression
An expression that returns a value of any built-in data type or user-defined data type that is comparable with data type of the other arguments. The data type cannot be a LOB, distinct type base on a LOB, XML, array, cursor, row, or structured type.

The result of the function is the largest argument value. The result can be null if at least one argument can be null; the result is the null value if any argument is null.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by the data types of all the arguments as explained in "Rules for result data types".

Notes

- The MAX scalar function is a synonym for the GREATEST scalar function.
- The MAX function cannot be used as a source function when creating a user-defined function. Because this function accepts any comparable data types as arguments, it is not necessary to create additional signatures to support user-defined data types.

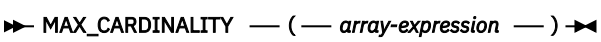
Example

Return the bonus for an employee, the greater of 500 and 5% of the employee's salary.

```
SELECT EMPNO, MAX(SALARY * 0.05, 500)
  FROM EMPLOYEE
```

MAX_CARDINALITY

The MAX_CARDINALITY function returns a value of type BIGINT representing the maximum number of elements that an array can contain. This is the cardinality that was specified in the CREATE TYPE statement for the ordinary array type.



The schema is SYSIBM.

array-expression
An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

Result

The result can be null; if the argument is an associative array, the result is the null value.

Example

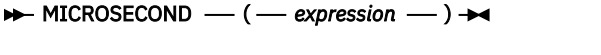
1. Return the maximum cardinality of the RECENT_CALLS array variable of array type PHONENUMBERS:

   ```
   SET LIST_SIZE = MAX_CARDINALITY(RECENT_CALLS)
   ```

   The SQL variable LIST_SIZE is set to 50, which is the maximum cardinality that the array type PHONENUMBERS was defined with.

MICROSECOND

The MICROSECOND function returns the microsecond part of a value.



The schema is SYSIBM.

expression
An expression that returns a value of one of the following built-in data types: DATE, TIMESTAMP, timestamp duration, or a valid character string representation of a date or timestamp that is not a CLOB. If a supplied argument is a DATE, it is first converted to a TIMESTAMP(0) value, assuming a

time of exactly midnight (00.00.00). In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a DATE, TIMESTAMP, or a valid string representation of a date or timestamp:
  - The integer ranges from 0 through 999 999.
  - If the precision of the timestamp exceeds 6, the value is truncated.
- If the argument is a duration:
  - The result reflects the microsecond part of the value which is an integer between -999 999 through 999 999. A nonzero result has the same sign as the argument.

Example

Assume a table TABLEA contains two columns, TS1 and TS2, of type TIMESTAMP. Select all rows in which the microseconds portion of TS1 is not zero and the seconds portion of TS1 and TS2 are identical.

```
SELECT * FROM TABLEA
  WHERE MICROSECOND(TS1) <> 0
    AND
  SECOND(TS1) = SECOND(TS2)
```

MIDNIGHT_SECONDS

Returns an integer value in the range 0 to 86 400, representing the number of seconds between midnight and the time value specified in the argument.

The schema is SYSFUN.

expression
An expression that returns a value of one of the following built-in data types: DATE, TIME, TIMESTAMP, or a valid character string representation of a date, time, or timestamp that is not a CLOB. If the expression is a DATE or a valid string representation of a date, it is first converted to a TIMESTAMP(0) value, assuming a time of exactly midnight (00.00.00). In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

Examples

- *Example 1*: Find the number of seconds between midnight and 00:10:10, and midnight and 13:10:10.

  ```
  VALUES (MIDNIGHT_SECONDS('00:10:10'), MIDNIGHT_SECONDS('13:10:10'))
  ```

  This example returns the following:

  ```
  1           2
  ----------- -----------
          610       47410
  ```

  Since a minute is 60 seconds, there are 610 seconds between midnight and the specified time. The same follows for the second example. There are 3600 seconds in an hour, and 60 seconds in a minute, resulting in 47 410 seconds between the specified time and midnight.

- *Example 2*: Find the number of seconds between midnight and 24:00:00, and midnight and 00:00:00.

```
   VALUES (MIDNIGHT_SECONDS('24:00:00'), MIDNIGHT_SECONDS('00:00:00'))
```

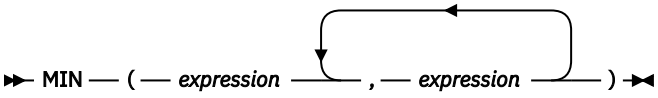
This example returns the following:

```
1           2
----------- -----------
      86400           0
```

Note that these two values represent the same point in time, but return different MIDNIGHT_SECONDS values.

MIN

The MIN function returns the minimum value in a set of values.



The schema is SYSIBM.

expression
An expression that returns a value of any built-in data type or user-defined data type that is comparable with data type of the other arguments. The data type cannot be a LOB, distinct type base on a LOB, XML, array, cursor, row, or structured type.

The result of the function is the smallest argument value. The result can be null if at least one argument can be null; the result is the null value if any argument is null.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by the data types of all the arguments as explained in "Rules for result data types".

Notes

- The MIN scalar function is a synonym for the LEAST scalar function.
- The MIN function cannot be used as a source function when creating a user-defined function. Because this function accepts any comparable data types as arguments, it is not necessary to create additional signatures to support user-defined data types.

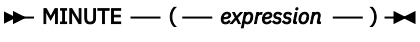
Example

Return the bonus for an employee, the LESSER of 5000 and 5% of the employee's salary.

```
  SELECT EMPNO, MIN(SALARY * 0.05, 5000)
    FROM EMPLOYEE
```

MINUTE

The MINUTE function returns the minute part of a value.



The schema is SYSIBM.

expression
An expression that returns a value of one of the following built-in data types: DATE, TIME, TIMESTAMP, time duration, timestamp duration, or a valid character string representation of a date, time, or timestamp that is not a CLOB. If a supplied argument is a DATE, it is first converted to a TIMESTAMP(0) value, assuming a time of exactly midnight (00.00.00). In a Unicode database, if a

supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a DATE, TIME, TIMESTAMP, or valid string representation of a date, time or timestamp:
  - The result is the minute part of the value, which is an integer between 0 and 59.
- If the argument is a time duration or timestamp duration:
  - The result is the minute part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Examples

Using the CL_SCHED sample table, select all classes with a duration less than 50 minutes.

```
SELECT * FROM CL_SCHED
  WHERE HOUR(ENDING - STARTING) = 0
    AND
  MINUTE(ENDING - STARTING) < 50
```

MINUTES_BETWEEN

The MINUTES_BETWEEN function returns the number of full minutes between the specified arguments.

The schema is SYSIBM.

expression1
An expression that specifies the first datetime value to compute the number of full minutes between two datetime values. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression1* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

expression2
An expression that specifies the second datetime value to compute the number of full minutes between two datetime values. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression2* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

If there is less than a full minute between *expression1* and *expression2*, the result is zero. If *expression1* is later than *expression2*, the result is positive. If *expression1* is earlier than *expression2*, the result is negative. In NPS compatibility mode, this function always returns a positive number. If *expression1* or *expression2* contains time information, this information is also used to determine the number of full minutes. If *expression1* or *expression2* does not contain time information, a time of midnight (00.00.00) is used for the argument that is missing time information.

The result of the function is a BIGINT. If either argument can be null, the result can be null. If either argument is null, the result is the null value.

Examples

1. Set the host variable NUM_MINUTES to the number of full minutes between 2012-03-01-01.00.00 and 2012-02-28-00.00.00.

```
SET :NUM_MINUTES = MINUTES_BETWEEN(TIMESTAMP '2012-03-01-01.07.00',
                                   TIMESTAMP '2012-02-28-00.00.00')
```

   The host variable NUM_MINUTES is set to 2940; 1440 of those minutes are incurred due to February 29, 2012.

2. Set the host variable NUM_MINUTES to the number of full minutes between 2013-09-11-23.59.59 and 2013-09-01-00.00.00.

```
SET :NUM_MINUTES = MINUTES_BETWEEN(TIMESTAMP '2013-09-11-23.59.59',
                                   TIMESTAMP '2013-09-01-00.00.00')
```

   The host variable NUM_MINUTES is set to 15839 because there is 1 second less than a full 15840 minutes between the arguments. It is positive because the first argument is later than the second argument.

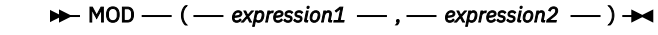
3. Set the host variable NUM_MINUTES to the number of full minutes between 2013-09-01-00.00.00 and 2013-09-11-23.59.59.

```
SET :NUM_MINUTES = MINUTES_BETWEEN(TIMESTAMP '2013-09-01-00.00.00',
                                   TIMESTAMP '2013-09-11-23.59.59')
```

   The host variable NUM_MINUTES is set to -15839 because there is 1 second less than a full 15840 minutes between the arguments. It is negative because the first argument is earlier than the second argument.

MOD

Returns the remainder of the first argument divided by the second argument.



The schema is SYSIBM. (The SYSFUN version of this function continues to be available.)

expression1
An expression that returns a value of any built-in numeric data type.

expression2
An expression that returns a value of any built-in numeric data type. This expression can only be zero when at least one of the function arguments is a decimal floating point.

The formula for calculating the remainder is:

```
MOD(x,y) = x-(x/y)*y
```

where x/y is the truncated integer result of the division.

The result only is negative when the first argument is negative.

The result can be null if either argument can be null or if neither argument is a decimal floating-point number and the dft_sqlmathwarn database configuration parameter is set to YES; the result is the null value when either argument is NULL.

The data type of the result depends on the data types of the arguments.

- Small integer if both arguments are small integers.
- Large integer if one argument is a large integer and the other argument is either a small integer or a large integer.
- Big integer if both arguments are integers and at least one argument is a big integer.

- Decimal if one argument is an integer and the other argument is a decimal. The result has the same precision as the decimal argument.
- Decimal if both arguments are decimals. The precision of the result is `MIN (p-s,p'-s')+ (MAX(s,s')` and the scale is `(MAX(s,s')`. Where p and s represent the precision and scale of the first argument and `p'` and `s'` represent the precision and scale of the second argument.
- Double-precision floating point if one argument is a floating-point number and the other is not a DECFLOAT. The arguments are converted to double-precision floating point numbers before performing the MOD function. For example, if one argument is a floating-point number and the other is an integer or decimal, the function is performed with a temporary copy of the integer or decimal, which has been converted to double-precision floating-point. The result must be in the range of floating-point numbers.
- Double-precision floating point if both arguments are floating-point numbers. The result must be in the range of floating-point numbers.
- DECFLOAT(34) if either argument is a decimal floating-point. If *expression2* evaluates to zero, the result is not a number (NaN) and an invalid operation warning with the associated SQLSTATE is issued.

  If either argument is a special decimal floating-point value, the general arithmetic operation rules for decimal floating-point apply (see Expressions).
- Results when arguments include infinity:
  - `MOD(x, +/-INFINITY)` returns the value x.
  - `MOD(+/-INFINITY, +/-INFINITY)` returns NaN and an invalid operation warning with the associated SQLSTATE.
  - `MOD(+/-INFINITY, x)` returns NaN and an invalid operation warning with the associated SQLSTATE.

Examples

- *Example 1*: Assume the host variable M1 is an INTEGER with a value of 5 and host variable M2 is an INTEGER with a value of 2

```
SELECT MOD(:M1, :M2)
   FROM SYSIBM.SYSDUMMY1
```

  The result is 1 with a data type of INTEGER.
- *Example 2*: Assume the host variable M1 is an INTEGER with a value of 5 and host variable M2 is a DECIMAL(3,2) with a value of 2.20

```
SELECT MOD(:M1, :M2)
   FROM SYSIBM.SYSDUMMY1
```

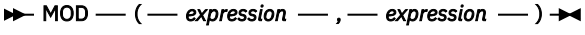
  The result is 0.60 with a data type of DECIMAL(3,2).
- *Example 3*: Assume the host variable M1 is a DECIMAL(4,2) with a value of 5.5 and host variable M2 is a DECIMAL(4,1) with a value of 2.0

```
SELECT MOD(:M1, :M2)
   FROM SYSIBM.SYSDUMMY1
```

  The result is 1.50 with a data type of DECIMAL(4,2).

MOD (SYSFUN schema)

Returns the remainder of the first argument divided by the second argument.



The schema is SYSFUN.

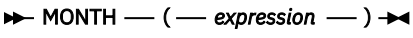
The result is negative only if first argument is negative. The result of the function is:

- SMALLINT if both arguments are SMALLINT
- INTEGER if one argument is INTEGER and the other is INTEGER or SMALLINT
- BIGINT if one argument is BIGINT and the other argument is BIGINT, INTEGER or SMALLINT.

The result can be null; if any argument is null, the result is the null value.

MONTH

The MONTH function returns the month part of a value.



The schema is SYSIBM.

expression
An expression that returns a value of one of the following built-in data types: DATE, TIMESTAMP, date duration, timestamp duration, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a DATE, TIMESTAMP, or a valid string representation of a date or timestamp:
  - The result is the month part of the value, which is an integer between 1 and 12.
- If the argument is a date duration or timestamp duration:
  - The result is the month part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

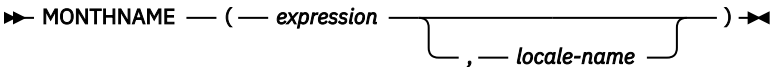
Example

Select all rows from the EMPLOYEE table for people who were born (BIRTHDATE) in DECEMBER.

```
SELECT * FROM EMPLOYEE
  WHERE MONTH(BIRTHDATE) = 12
```

MONTHNAME

The MONTHNAME function returns a character string containing the name of the month (for example, January) for the month portion of the input value.



The schema is SYSIBM. The SYSFUN version of the MONTHNAME function continues to be available.

The character string returned is based on *locale-name* or the value of the special register CURRENT LOCALE LC_TIME.

expression
An expression that returns a value of one of the following built-in data types: a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

locale-name
A character constant that specifies the locale used to determine the language of the result. The value of *locale-name* is not case-sensitive and must be a valid locale (SQLSTATE 42815). For information

about valid locales and their naming, see "Locale names for SQL and XQuery". If *locale-name* is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result is a varying-length character string. The length attribute is 100. If the resulting string exceeds the length attribute of the result, the result will be truncated. If the *expression* argument can be null, the result can be null; if the *expression* argument is null, the result is the null value. The code page of the result is the code page of the section. The string units of the result is determined by the string units of the environment.

Notes

- **Julian and Gregorian calendar**: The transition from the Julian calendar to the Gregorian calendar on 15 October 1582 is taken into account by this function. However, the SYSFUN version of the MONTHNAME function assumes the Gregorian calendar for all calculations.
- **Determinism**: MONTHNAME is a deterministic function. However, when *locale-name* is not explicitly specified, the invocation of the function depends on the value of the special register CURRENT LOCALE LC_TIME. This invocation that depends on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621, 428EC, or 429BX).

Example

Assume that the variable TMSTAMP is defined as TIMESTAMP and has the following value: 2007-03-09-14.07.38.123456. The following examples show several invocations of the function and the resulting string values. The result type in each case is VARCHAR(100).

```
Function invocation                          Result
-------------------------                   ----------
MONTHNAME (TMSTAMP, 'CLDR181_en_US')          March
MONTHNAME (TSMTAMP, 'CLDR181_de_DE')          Marz
MONTHNAME (TMSTAMP, 'CLDR181_fr_FR')          mars
```

MONTHS_BETWEEN

The MONTHS_BETWEEN function returns an estimate of the number of months between *expression1* and *expression2*.

The schema is SYSIBM.

expression1* or *expression2
Expressions that return a value of either a DATE or TIMESTAMP data type.

If *expression1* represents a date that is later than *expression2*, the result is positive. If *expression1* represents a date that is earlier than expression2, the result is negative.

- If *expression1* and *expression2* represent dates or timestamps with the same day of the month, or both arguments represent the last day of their respective months, the result is a the whole number difference based on the year and month values ignoring any time portions of timestamp arguments.
- Otherwise, the whole number part of the result is the difference based on the year and month values. The fractional part of the result is calculated from the remainder based on an assumption that every month has 31 days. If either argument represents a timestamp, the arguments are effectively processed as timestamps with maximum precision, and the time portions of these values are also considered when determining the result.

The result of the function is a DECIMAL(31,15). If either argument can be null, the result can be null. If either argument is null, the result is the null value.

Examples

- *Example 1*: Calculate the number of months that project AD3100 will take. Assume that the start date is 1982-01-01 and the end date is 1983-02-01.

```
SELECT MONTHS_BETWEEN (PRENDATE, PRSDATE)
   FROM PROJECT
   WHERE PROJNO='AD3100'
```

  The result is 13.000000000000000.

- *Example 2*: Here are some additional examples to consider:

Table 74. Additional examples using MONTHS_BETWEEN

Value for argument *e1*	Value for argument *e2*	Value returned by MONTHS_BETWEEN (*e1*,*e2*)	Value returned by ROUND (MONTHS_BETWEEN (*e1*,*e2*)*31,2)	Comment
2005-02-02	2005-01-01	1.032258064516129	32.00	
2007-11-01-09.00.00.00000	2007-12-07-14.30.12.12345	-1.200945386592741	-37.23	
2007-12-13-09.40.30.00000	2007-11-13-08.40.30.00000	1.000000000000000	31.00	See Note 1
2007-03-15	2007-02-20	0.838709677419354	26.00	See Note 2
2008-02-29	2008-02-28-12.00.00	0.016129032258064	0.50	
2008-03-29	2008-02-29	1.000000000000000	31.00	
2008-03-30	2008-02-29	1.032258064516129	32.00	
2008-03-31	2008-02-29	1.000000000000000	31.00	See Note 3

Note:

1. The time difference is ignored because the day of the month is the same for both values.
2. The result is not 23 because, even though February has 28 days, the assumption is that all months have 31 days.
3. The result is not 33 because both dates are the last day of their respective month, and so the result is only based on the year and month portions.

MULTIPLY_ALT

The MULTIPLY_ALT scalar function returns the product of the two arguments.

The schema is SYSIBM.

numeric_expression1
An expression that returns a value of any built-in numeric data type.

numeric_expression2
An expression that returns a value of any built-in numeric data type.

The MULTIPLY_ALT function is provided as an alternative to the multiplication operator, especially when the sum of the decimal precisions of the arguments exceeds 31.

The result of the function is DECIMAL when both arguments are exact numeric data types (DECIMAL, BIGINT, INTEGER, or SMALLINT); otherwise the operation is carried out using decimal floating-point arithmetic and the result of the function is decimal floating-point with a precision determined by the data

type of the arguments in the same way the precision is determined for decimal floating-point arithmetic. A floating-point or string argument is cast to DECFLOAT(34) before evaluating the function.

When the result of the function is DECIMAL, the precision and scale of the result are determined as follows, using the symbols *p* and *s* to denote the precision and scale of the first argument, and the symbols *p*' and *s*' to denote the precision and scale of the second argument.

- The precision is MIN(31, *p* + *p*')
- The scale is:
  - 0 if the scale of both arguments is 0
  - MIN(31, *s* + *s*') if *p* + *p*' is less than or equal to 31
  - MAX(MIN(3, *s* + *s*'), 31 - (*p* - *s* + *p*' - *s*')) if *p* + *p*' is greater than 31.

The result can be null if at least one argument can be null, or if the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if one of the arguments is null.

The MULTIPLY_ALT function is a preferable choice to the multiplication operator when performing decimal arithmetic where a scale of at least 3 is required and the sum of the precisions exceeds 31. In these cases, the internal computation is performed so that overflows are avoided. The final result is then assigned to the result data type, using truncation where necessary to match the scale. Note that overflow of the final result is still possible when the scale is 3.

The following table is a sample comparing the result types using MULTIPLY_ALT and the multiplication operator.

Type of argument 1	Type of argument 2	Result using MULTIPLY_ALT	Result using multiplication operator
DECIMAL(31,3)	DECIMAL(15,8)	DECIMAL(31,3)	DECIMAL(31,11)
DECIMAL(26,23)	DECIMAL(10,1)	DECIMAL(31,19)	DECIMAL(31,24)
DECIMAL(18,17)	DECIMAL(20,19)	DECIMAL(31,29)	DECIMAL(31,31)
DECIMAL(16,3)	DECIMAL(17,8)	DECIMAL(31,9)	DECIMAL(31,11)
DECIMAL(26,5)	DECIMAL(11,0)	DECIMAL(31,3)	DECIMAL(31,5)
DECIMAL(21,1)	DECIMAL(15,1)	DECIMAL(31,2)	DECIMAL(31,2)

Example

Multiply two values where the data type of the first argument is DECIMAL(26,3) and the data type of the second argument is DECIMAL(9,8). The data type of the result is DECIMAL(31,7).

```
values multiply_alt(98765432109876543210987.654,5.43210987)
1
--------------------------------
536504678578875294857887.5277415
```

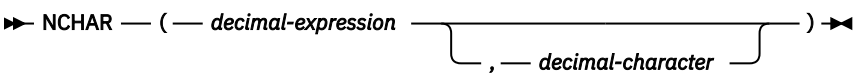
Note that the complete product of these two numbers is 536504678578875294857887.52774154498, but the last 4 digits are truncated to match the scale of the result data type. Using the multiplication operator with the same values will cause an arithmetic overflow, since the result data type is DECIMAL(31,11) and the result value has 24 digits left of the decimal, but the result data type only supports 20 digits.

NCHAR

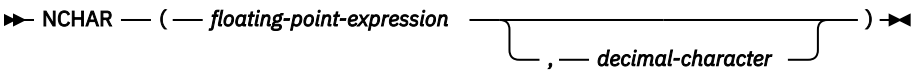
The NCHAR function returns a fixed-length national character string representation of a variety of data types.

Integer to nchar

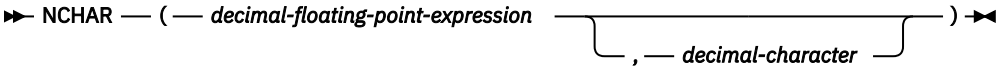
Decimal to nchar



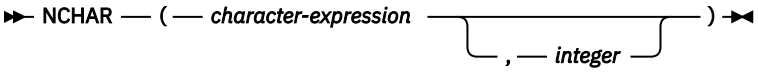
Floating-point to nchar



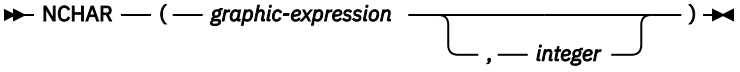
Decimal floating-point to nchar



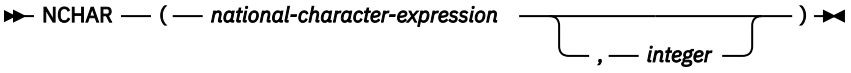
Character to nchar



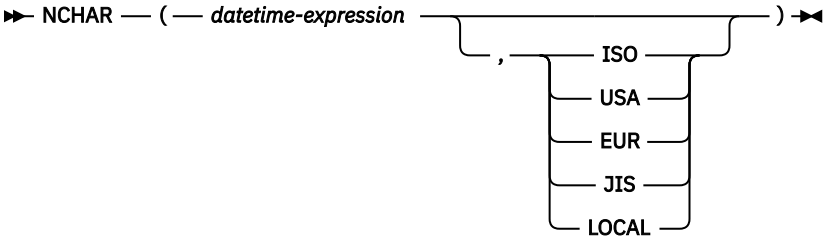
Graphic to nchar



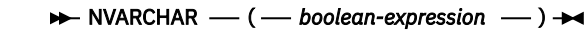
Nchar to nchar



Datetime to nchar



Boolean to nvarchar



The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature.

The NCHAR function can be specified only in a Unikode database (SQLSTATE 560AA).

Result

The NCHAR function returns a fixed-length national character string representation of:

- An integer number, if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number, if the first argument is a decimal number
- A double-precision floating-point number, if the first argument is a DOUBLE or REAL
- A decimal floating-point number, if the argument is a decimal floating-point number (DECFLOAT)
- A character string, if the first argument is any type of character string
- A graphic string, if the first argument is any type of graphic string
- A national character string, if the first argument is any type of national character string
- A datetime value, if the first argument is a DATE, TIME, or TIMESTAMP
- A Boolean value (TRUE or FALSE)

The NCHAR scalar function is a synonym for a scalar cast function with result string units as specified in the following table.

Table 75. NCHAR scalar function synonyms

NCHAR_MAPPING value	Synonym function	Result string units
CHAR_CU32	CHAR	CODEUNITS32
GRAPHIC_CU32	GRAPHIC	CODEUNITS32
GRAPHIC_CU16	GRAPHIC[1]	CODEUNITS16

1. When the first argument has string units of CODEUNITS32 and the second argument is not specified, the length attribute of the result is different from the GRAPHIC function because the string units of the result is CODEUNITS16. In this case, the length attribute of the result is 2 times the length attribute of the result that is determined by the GRAPHIC function.

NCHR

The NCHR function returns a Unicode character with the specified UTF-32 Unicode code point.

The schema is SYSIBM.

Note: The NCHR function can be specified only in a Unicode database (SQLSTATE 560AA).

integer-expression

An expression that specifies the UTF-32 Unicode code point.

The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type.

If the value is not of type INTEGER, it is implicitly cast to INTEGER before the function is evaluated.

The integer-expression must be greater than or equal to 0 (SQLSTATE 42815).

The integer-expression must represent a valid UTF-32 Unicode character (SQLSTATE 22021).

Result

The result of the function is a Unicode character with the CHAR data type and a length of 1 CODEUNITS32.

If the argument can be null, the result can be null. If the argument is null, the result is the null value.

Notes

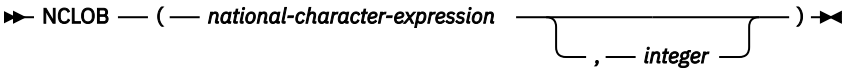
As a syntax alternative, you can specify UNICHR as a synonym for NCHR.

In the following example, the variable GCLEF is assigned the value '𝄞' (UTF-32 Unicode value U&'\+01d11e'= 119070):SET GCLEF=NCHR(119070):

```
SET GCLEF=NCHR(119070);
```

NCLOB

The NCLOB function returns a NCLOB representation of any type of national character string.



The schema is SYSIBM.

The NCLOB function can be specified only in a Unicode database (SQLSTATE 560AA).

The NCLOB scalar function is a synonym for a scalar cast function with result string units as specified in the following table.

Table 76. NCLOB scalar function synonyms

NCLOB_MAPPING value	Synonym function	Result string units
CHAR_CU32	CLOB	CODEUNITS32
GRAPHIC_CU32	DCLOB	CODEUNITS32
GRAPHIC_CU16	DCLOB[1]	CODEUNITS16

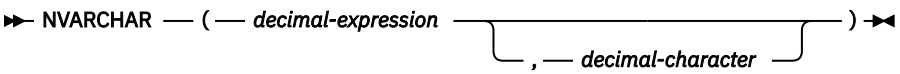
1. When the first argument has string units CODEUNITS32 and the second argument is not specified, the length attribute of the result is different from the DCLOB function because the string units of the result is CODEUNITS16. In this case, the length attribute of the result is 2 times the length attribute of the result that is otherwise determined by the DCLOB function.

NVARCHAR

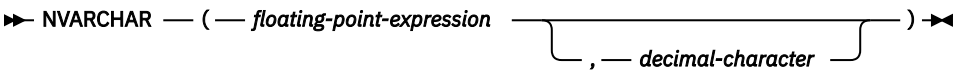
The NVARCHAR function returns a varying-length national character string representation of a variety of data types.

Integer to nvarchar

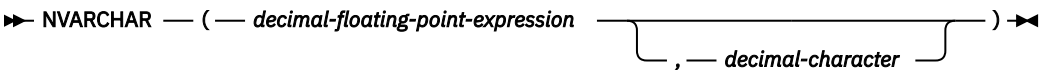
Decimal to nvarchar



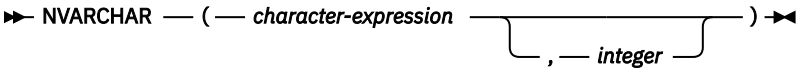
Floating-point to nvarchar



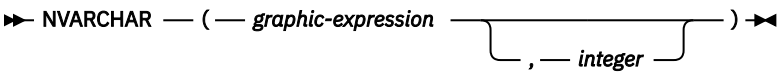
Decimal floating-point to nvarchar



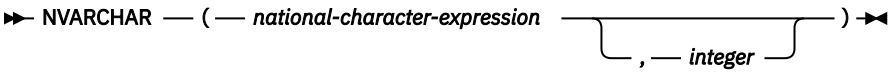
Character to nvarchar



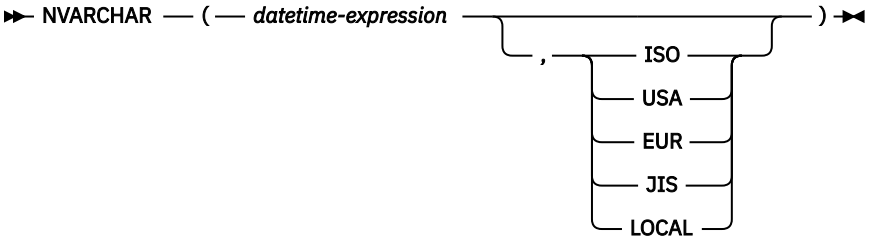
Graphic to nvarchar



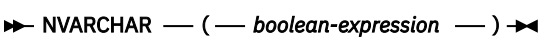
Nchar to nvarchar



Datetime to nvarchar



Boolean to nvarchar



The schema is SYSIBM.

The function name cannot be specified as a qualified name when keywords are used in the function signature.

NVARCHAR can be specified only in a Unicode database (SQLSTATE 560AA).

Result

The NVARCHAR function returns a varying-length national character string representation of:

- An integer number, if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number, if the first argument is a decimal number
- A double-precision floating-point number, if the first argument is a DOUBLE or REAL

- A decimal floating-point number, if the first argument is a decimal floating-point number (DECFLOAT)
- A character string, if the first argument is any type of character string
- A graphic string, if the first argument is any type of graphic string
- An national character string, if the first argument is any type of national character string
- A datetime value, if the first argument is a DATE, TIME, or TIMESTAMP
- A Boolean value (TRUE or FALSE)

The NVARCHAR scalar function is a synonym for a scalar cast function with result string units as specified in the following table.

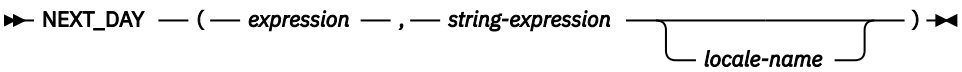
Table 77. NVARCHAR scalar function synonyms

NCHAR_MAPPING	Synonym function	Result string units
CHAR_CU32	VARCHAR	CODEUNITS32
GRAPHIC_CU32	VARGRAPHIC	CODEUNITS32
GRAPHIC_CU16	VARGRAPHIC[1]	CODEUNITS16

1. When the first argument has string units CODEUNITS32 and the second argument is not specified, the length attribute of the result is different from the VARGRAPHIC function because the string units of the result is CODEUNITS16. In this case, the length attribute of the result is 2 times the length attribute of the result that is otherwise determined by the VARGRAPHIC function.

NEXT_DAY

The NEXT_DAY scalar function returns a datetime value that represents the first weekday, named by *string-expression*, that is later than the date in *expression*.



The schema is SYSIBM.

expression
An expression that returns a value of one of the following built-in data types: a DATE or a TIMESTAMP.

string-expression
An expression that returns a built-in character data type. The value must be a valid day of the week for the *locale-name*. The value can be specified either as the full name of the day or the associated abbreviation. For example, if the locale is 'en_US' then the following values are valid:

Table 78. Valid day names and abbreviations for the 'en_US' locale

Day of week	Abbreviation
MONDAY	MON
TUESDAY	TUE
WEDNESDAY	WED
THURSDAY	THU
FRIDAY	FRI
SATURDAY	SAT
SUNDAY	SUN

The minimum length of the input value is the length of the abbreviation. The characters can be specified in lower or upper case and any characters immediately following a valid abbreviation are ignored.

locale-name
A character constant that specifies the locale used to determine the language of the *string-expression* value. The value of *locale-name* is not case sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming, see "Locale names for SQL and XQuery". If *locale-name* is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result of the function has the same data type as *expression*, unless *expression* is a string, in which case the result data type is TIMESTAMP(6) . The result can be null; if any argument is null, the result is the null value.

Any hours, minutes, seconds or fractional seconds information included in *expression* is not changed by the function. If *expression* is a string representing a date, the time information in the resulting TIMESTAMP value is all set to zero.

Notes

- **Determinism:** NEXT_DAY is a deterministic function. However, when locale-name is not explicitly specified, the invocation of the function depends on the value of the special register CURRENT LOCALE LC_TIME. Invocations of the function that depend on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621, 428EC, or 429BX).

Examples

- *Example 1*: Set the variable NEXTDAY with the date of the Tuesday following April 24, 2007.

```
SET NEXTDAY = NEXT_DAY(DATE '2007-04-24', 'TUESDAY')
```

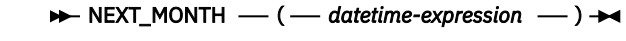
  The variable NEXTDAY is set with the value of '2007-05-01', since April 24, 2007 is itself a Tuesday.
- *Example 2*: Set the variable vNEXTDAY with the timestamp of the first Monday in May, 2007. Assume the variable vDAYOFWEEK = 'MON'.

```
SET vNEXTDAY = NEXT_DAY(LAST_DAY(CURRENT_TIMESTAMP),vDAYOFWEEK)
```

  The variable vNEXTDAY is set with the value of '2007-05-07-12.01.01.123456', assuming that the value of the CURRENT_TIMESTAMP special register is '2007-04-24-12.01.01.123456'.

NEXT_MONTH

The NEXT_MONTH function returns the first day of the next month after the specified date.



The schema is SYSIBM.

datetime-expression
An expression that specifies a date after which the first day of the next month is to be returned. The expression must return a value that is a DATE, a TIMESTAMP, a CHAR, or a VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported using implicit casting. If *datetime-expression* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string accepted by the TIMESTAMP scalar function.

The result of the function is DATE. If the argument can be null, the result can be null. If the argument is null, the result is the null value.

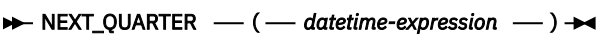
Example

The following example returns the date value of the first day of the next month after the date specified by the input:

```
values next_month('2007-02-18')
      Result: 2007-03-01
```

NEXT_QUARTER

The NEXT_QUARTER function returns the first day of the next quarter after the date specified by the input.



The schema is SYSIBM.

datetime-expression
An expression that specifies a date after which the first day of the next quarter is to be returned. The expression must return a value that is a DATE, a TIMESTAMP, a CHAR, or a VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported using implicit casting. If *datetime-expression* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string accepted by the TIMESTAMP scalar function.

The result of the function is DATE. If the argument can be null, the result can be null. If the argument is null, the result is the null value.

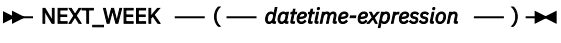
Example

The following example returns the date value of the first day of the next quarter after the date specified by the input:

```
values next_quarter('2007-02-18')
      Result: 2007-04-01
```

NEXT_WEEK

The NEXT_WEEK function returns the first day of the next week after the specified date. Sunday is considered the first day of that new week.



The schema is SYSIBM.

datetime-expression
An expression that specifies a date after which the first day of the next week is to be returned. The expression must return a value that is a DATE, a TIMESTAMP, a CHAR, or a VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported using implicit casting. If *datetime-expression* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string accepted by the TIMESTAMP scalar function.

The result of the function is DATE. If the argument can be null, the result can be null. If the argument is null, the result is the null value.

Example

The following example returns the date value of the first day of the next week after the date specified by the input:

```
values next_week('2007-02-18')
      Result: 2007-02-25
```

NEXT_YEAR

The NEXT_YEAR function returns the first day of the year follows the year containing the date specified by the input.

The schema is SYSIBM.

datetime-expression
An expression that specifies a date after which first day of the next year is to be returned. The expression must return a value that is a DATE, a TIMESTAMP, a CHAR, or a VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported using implicit casting. If *datetime-expression* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string accepted by the TIMESTAMP scalar function.

The result of the function is DATE. If the argument can be null, the result can be null. If the argument is null, the result is the null value.

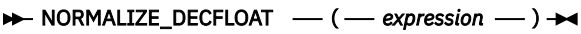
Example

Returns the date value of the first day of the year that follows the year containing the date specified by the input.

```
values next_year('2007-02-18')
      Result: 2008-01-01
```

NORMALIZE_ DECFLOAT

The NORMALIZE_DECFLOAT function returns a decimal floating-point value equal to the input argument set to its simplest form; that is, a nonzero number with trailing zeros in the coefficient has those zeros removed.



The schema is SYSIBM.

Returning a decimal floating-point value equal to the input argument set to its simplest form may require representing the number in normalized form by dividing the coefficient by the appropriate power of ten and adjusting the exponent accordingly. A zero value has its exponent set to 0.

expression
An expression that returns a value of any built-in numeric data type. Arguments of type SMALLINT, INTEGER, REAL, DOUBLE, or DECIMAL(*p*,*s*), where *p* <= 16, are converted to DECFLOAT(16) for processing. Arguments of type BIGINT or DECIMAL(*p*,*s*), where *p* > 16, are converted to DECFLOAT(34) for processing.

The result of the function is a DECFLOAT(16) value if the data type of expression after conversion to decimal floating-point is DECFLOAT(16). Otherwise, the result of the function is a DECFLOAT(34) value. If the argument is a special decimal floating-point value, the result is the same special decimal floating-point value. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

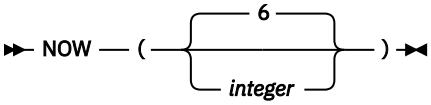
Examples

The following examples show the values that are returned by the NORMALIZE_DECFLOAT function, given a variety of input decimal floating-point values:

```
NORMALIZE_DECFLOAT(DECFLOAT(2.1)) = 2.1
NORMALIZE_DECFLOAT(DECFLOAT(-2.0)) = -2
NORMALIZE_DECFLOAT(DECFLOAT(1.200)) = 1.2
NORMALIZE_DECFLOAT(DECFLOAT(-120)) = -1.2E+2
NORMALIZE_DECFLOAT(DECFLOAT(120.00)) = 1.2E+2
NORMALIZE_DECFLOAT(DECFLOAT(0.00)) = 0
NORMALIZE_DECFLOAT(-NAN) = -NaN
NORMALIZE_DECFLOAT(-INFINITY) = -Infinity
```

NOW

The NOW function returns a timestamp based on a reading of the time-of-day clock when the SQL statement is executed at the current server.



The schema is SYSIBM.

integer
Specifies the precision of the timestamp. Valid values are 0 to 12, inclusive. If you do not specify an integer, the default precision of 6 is used.

The value returned by the NOW function is the same as the value returned by the CURRENT TIMESTAMP special register. If this function is used more than once within a single SQL statement or used with CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP special registers within a single statement, all values are based on a single clock reading.

The precision of the clock reading varies by platform and the resulting value is padded with zeros if the precision of the retrieved clock reading is less than the precision of the request.

The data type of the result is a timestamp. The result cannot be null.

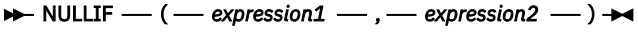
Example

Return the timestamp for the current time.

```
values now().
           Result: 2015-04-01-03.17.04.579645.
```

NULLIF

The NULLIF function returns a null value if the arguments are equal, otherwise it returns the value of the first argument.



The schema is SYSIBM.

expression1
An expression that returns a value of any built-in or user-defined data type.

expression2
An expression that returns a value of any built-in or user-defined data type that is comparable with the data type of the other argument according to the rules for equality comparison.

The result of using NULLIF(e1,e2) is the same as using the following expression:

```
CASE WHEN e1=e2 THEN NULL ELSE e1 END
```

When e1=e2 evaluates to unknown (because one or both arguments is NULL), CASE expressions consider this not true. Therefore, in this situation, NULLIF returns the value of the first argument.

Notes

- The NULLIF function cannot be used as a source function when creating a user-defined function. Because this function accepts any comparable data types as arguments, it is not necessary to create additional signatures to support user-defined data types.

Example

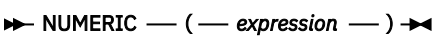
Assume host variables PROFIT, CASH, and LOSSES have DECIMAL data types with the values 4500.00, 500.00, and 5000.00 respectively:

```
NULLIF (:PROFIT + :CASH , :LOSSES )
```

Returns a null value.

NUMERIC

The NUMERIC function returns a decimal representation of a number, a string representation of a number, or a datetime value.

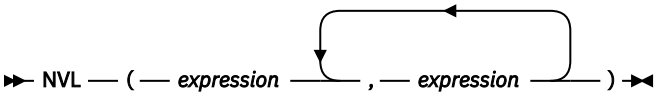


The schema is SYSIBM.

The NUMERIC scalar function is a synonym for the DECIMAL scalar function.

NVL

The NVL function returns the first argument that is not null.

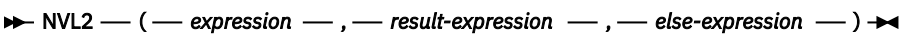


The schema is SYSIBM.

NVL is a synonym for COALESCE.

NVL2

The NVL2 function returns the second argument when the first argument is not NULL. If the first argument is NULL, the third argument is returned.



The schema is SYSIBM.

The NVL2 function is a synonym for the following statement:

```
CASE WHEN expression IS NOT NULL
     THEN result-expression
     ELSE else-expression
END
```

OCTET_LENGTH

The OCTET_LENGTH function returns the length of *expression* in octets (bytes).

The schema is SYSIBM.

expression
An expression that returns a value that is a built-in string data type.

The result of the function is INTEGER. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The length of character or graphic strings includes trailing blanks. The length of binary strings includes binary zeros. The length of varying-length strings is the actual length and not the maximum length.

For greater portability, code your application to be able to accept a result of data type DECIMAL(31).

Examples

- *Example 1*: Assume that table T1 has a GRAPHIC(10) column named C1.

```
SELECT OCTET_LENGTH(C1) FROM T1
```

  returns the value 20.

- *Example 2*: The following example works with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown in different Unicode encoding forms in the following example:

	'&'	'N'	'~'	'A'	'B'
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'

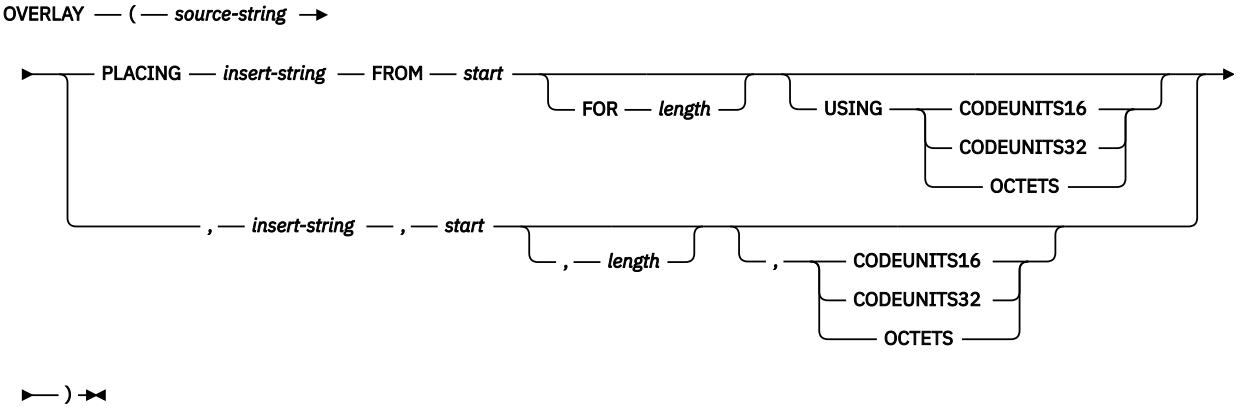
  Assume that the variables UTF8_VAR and UTF16_VAR contain the UTF-8 and the UTF-16BE representations of the string, respectively.

```
SELECT OCTET_LENGTH(UTF8_VAR),
  OCTET_LENGTH(UTF16_VAR)
FROM SYSIBM.SYSDUMMY1
```

  returns the values 9 and 12, respectively.

OVERLAY

The OVERLAY function returns a string in which, beginning at *start* in *source-string*, *length* of the specified code units have been deleted and *insert-string* has been inserted.



The schema is SYSIBM.

source-string
An expression that specifies the source string. The expression must return a value that is a built-in string, numeric, boolean, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function.

insert-string
An expression that specifies the string to be inserted into *source-string*, starting at the position identified by *start*. The expression must return a value that is a built-in string, numeric, boolean, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. If the code page of the *insert-string* differs from that of the *source-string*, *insert-string* is converted to the code page of the *source-string*.

start
An expression that returns an integer value. The integer value specifies the starting point within the source string where the deletion and the insertion of another string is to begin. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The integer value is the starting point in code units using the specified string units. The integer value must be between 1 and the actual length of *source-string* in the specified string units plus one (SQLSTATE 42815). If OCTETS is specified and the result is graphic data, the value must be an odd number between 1 and the actual octet length of *source-string* plus one (SQLSTATE 428GC or 22011).

length
An expression that specifies the number of code units (in the specified string units) that are to be deleted from the source string, starting at the position identified by *start*. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value must be a positive integer or zero (SQLSTATE 22011). If OCTETS is specified and the result is graphic data, the value must be an even number or zero (SQLSTATE 428GC).

Not specifying *length* is equivalent to specifying a value of 1, except when OCTETS is specified and the result is graphic data, in which case, not specifying *length* is equivalent to specifying a value of 2.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of *start* and *length*.

CODEUNITS16 specifies that *start* and *length* are expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *start* and *length* are expressed in 32-bit UTF-32 code units. OCTETS specifies that *start* and *length* are expressed in bytes.

If the string unit is specified as CODEUNITS16 or CODEUNITS32, and the result is a binary string or a FOR BIT DATA string, an error is returned (SQLSTATE 428GC). If the string unit is specified as

CODEUNITS16 or OCTETS, and the string units of *source-string* is CODEUNITS32, an error is returned (SQLSTATE 428GC). If the string unit is specified as OCTETS, the operation is performed in the code page of the *source-string*.

If a string unit argument is not specified and both *source-string* and *insert-string* are either a character string that is not FOR BIT DATA or is a graphic string, the default is CODEUNITS32. Otherwise, the default is OCTETS.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

The data type of the result depends on the data types of *source-string* and *insert-string*, as shown in the following tables of supported type combinations. The string unit of the result is the string unit of *source-string*. If either *source-string* or *insert-string* is defined as FOR BIT DATA the other argument cannot be defined with string units of CODEUNITS32. The second table applies to Unicode databases only.

Table 79. Data type of the result as a function of the data types of source-string and insert-string

source-string	***insert-string***	**Result**
CHAR or VARCHAR	CHAR or VARCHAR	VARCHAR
GRAPHIC or VARGRAPHIC	GRAPHIC or VARGRAPHIC	VARGRAPHIC
CLOB	CHAR, VARCHAR, or CLOB	CLOB
CHAR or VARCHAR	CLOB	CLOB
DBCLOB	GRAPHIC, VARGRAPHIC, or DBCLOB	DBCLOB
GRAPHIC or VARGRAPHIC	DBCLOB	DBCLOB
CHAR or VARCHAR	CHAR FOR BIT DATA or VARCHAR FOR BIT DATA	VARCHAR FOR BIT DATA
CHAR FOR BIT DATA or VARCHAR FOR BIT DATA	CHAR, VARCHAR, CHAR FOR BIT DATA, or VARCHAR FOR BIT DATA	VARCHAR FOR BIT DATA
BINARY or VARBINARY	BINARY or VARBINARY	VARBINARY
BLOB	BLOB	BLOB
BINARY or VARBINARY	BLOB	BLOB

Note: If the data types for *source-string* and *insert-string* are a combination of binary and FOR BIT DATA string, the argument that is not a binary data type is handled as if it was cast to the corresponding binary data type.

Table 80. Data type of the result as a function of the data types of source-string and insert-string (Unicode databases only)

source-string	***insert-string***	**Result**
CHAR or VARCHAR	GRAPHIC or VARGRAPHIC	VARCHAR
GRAPHIC or VARGRAPHIC	CHAR or VARCHAR	VARGRAPHIC
CLOB	GRAPHIC, VARGRAPHIC, or DBCLOB	CLOB
DBCLOB	CHAR, VARCHAR, or CLOB	DBCLOB

A *source-string* can have a length of 0; in this case, *start* must be 1 (as implied by the bounds provided in the description for *start*), and the result of the function is a copy of the *insert-string*.

An *insert-string* can also have a length of 0. This has the effect of deleting the code units identified by *start* and *length* from the *source-string*.

The length attribute of the result is the length attribute of *source-string* plus the length attribute of *insert-string* when the string units of the *source-string* and *insert-string* are the same or the result string units is CODEUNITS32. Special cases are listed in the following table.

Table 81. Data type of the result as a function of the data types of source-string and insert-string (special cases, Unicode databases only)

source-string		***insert-string***		**Result**	
Data type	**String units**	**Data type**	**String units**	**Length attribute**	**String units**
Character string with length attribute A	OCTETS	Graphic string with length attribute B	CODEUNITS16	A+3*B	OCTETS
			CODEUNITS32	A+4*B	
		Character with length attribute B	CODEUNITS32	A+4*B	
Graphic string with length attribute A	CODEUNITS16	Character with length attribute B	OCTETS	A+B	CODEUNITS16
			CODEUNITS32	A+2*B	
		Graphic string with length attribute B	CODEUNITS32	A+2*B	

The actual length of the result depends on the actual length of *source-string*, the actual length of the deleted string, the actual length of the *insert-string*, and string units used for the *start* and *length* arguments. For example, if the string arguments are character strings in OCTETS and the OCTETS is used as the fourth argument, the actual length of the result is A1 - MIN((A1 - V2 + 1), V3) + A4, where:

- A1 is the actual length of *source-string*
- V2 is the value of *start*
- V3 is the value of *length*
- A4 is the actual length of *insert-string*

If the actual length of the result string exceeds the maximum for the return data type, an error is returned (SQLSTATE 54006).

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Create the strings 'INSISTING', 'INSISERTING', and 'INSTING' from the string 'INSERTING' by inserting text into the middle of the existing text.

```
SELECT OVERLAY('INSERTING','IS',4,2,OCTETS),
  OVERLAY('INSERTING','IS',4,0,OCTETS),
  OVERLAY('INSERTING','',4,2,OCTETS)
  FROM SYSIBM.SYSDUMMY1
```

- *Example 2*: Create the strings 'XXINSERTING', 'XXNSERTING', 'XXSERTING', and 'XXERTING' from the string 'INSERTING' by inserting text before the existing text, using 1 as the starting point.

```
SELECT OVERLAY('INSERTING','XX',1,0,CODEUNITS16)),
  OVERLAY('INSERTING','XX',1,1,CODEUNITS16)),
  OVERLAY('INSERTING','XX',1,2,CODEUNITS16)),
  OVERLAY('INSERTING','XX',1,3,CODEUNITS16))
  FROM SYSIBM.SYSDUMMY1
```

- *Example 3*: Create the string 'ABCABCXX' from the string 'ABCABC' by inserting text after the existing text. Because the source string is 6 characters long, set the starting position to 7 (one plus the length of the source string).

```
SELECT OVERLAY('ABCABC','XX',7,0,CODEUNITS16))
  FROM SYSIBM.SYSDUMMY1
```

- *Example 4*: Change the string 'Hegelstraße' to 'Hegelstrasse'.

```
SELECT OVERLAY('Hegelstraße','ss',10,1,CODEUNITS16))
  FROM SYSIBM.SYSDUMMY1
```

- *Example 5*: The following example works with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown in different Unicode encoding forms in the following example:

	'&'	'N'	'~'	'A'	'B'
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'

Assume that the variables UTF8_VAR and UTF16_VAR contain the UTF-8 and the UTF-16BE representations of the string, respectively. Use the OVERLAY function to insert a 'C' into the Unicode string '&N~AB'.

```
SELECT OVERLAY(UTF8_VAR, 'C', 1, CODEUNITS16),
  OVERLAY(UTF8_VAR, 'C', 1, CODEUNITS32),
  OVERLAY(UTF8_VAR, 'C', 1, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 'C?N~AB', 'CN~AB', and 'CbbbN~AB', respectively, where '?' represents X'EDB49E', which corresponds to the X'DD1E' in the intermediate UTF-16 form, and 'bbb' replaces the UTF-8 incomplete characters X'9D849E'.

```
SELECT OVERLAY(UTF8_VAR, 'C', 5, CODEUNITS16),
  OVERLAY(UTF8_VAR, 'C', 5, CODEUNITS32),
  OVERLAY(UTF8_VAR, 'C', 5, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~CB', '&N~AC', and '&N~AB', respectively.

```
SELECT OVERLAY(UTF16_VAR, 'C', 1, CODEUNITS16),
  OVERLAY(UTF16_VAR, 'C', 1, CODEUNITS32)
FROM SYSIBM.SYSDUMMY1
```

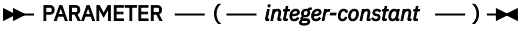
returns the values 'C?N~AB' and 'CN~AB', respectively, where '?' represents the unmatched low surrogate U+DD1E.

```
SELECT OVERLAY(UTF16_VAR, 'C', 5, CODEUNITS16),
  OVERLAY(UTF16_VAR, 'C', 5, CODEUNITS32)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~CB' and '&N~AC', respectively.

PARAMETER

The PARAMETER function represents a position in an SQL statement where the value is provided dynamically by XQuery as part of the invocation of the db2-fn:sqlquery function.



The schema is SYSIBM.

integer-constant
An integer constant that specifies a position index of a value in the arguments of db2-fn:sqlquery. The value must be between 1 and the total number of the arguments specified in the db2-fn:sqlquery SQL statement (SQLSTATE 42815).

The PARAMETER function represents a position in an SQL statement where the value is provided dynamically by XQuery as part of the invocation of the db2-fn:sqlquery function. The argument of the PARAMETER function determines which value is substituted for the PARAMETER function when the db2-fn:sqlquery function is executed. The value supplied by the PARAMETER function can be referenced multiple times within the same SQL statement.

This function can only be used in a fullselect contained in the string literal argument of the db2-fn:sqlquery function in an XQuery expression (SQLSTATE 42887).

Example

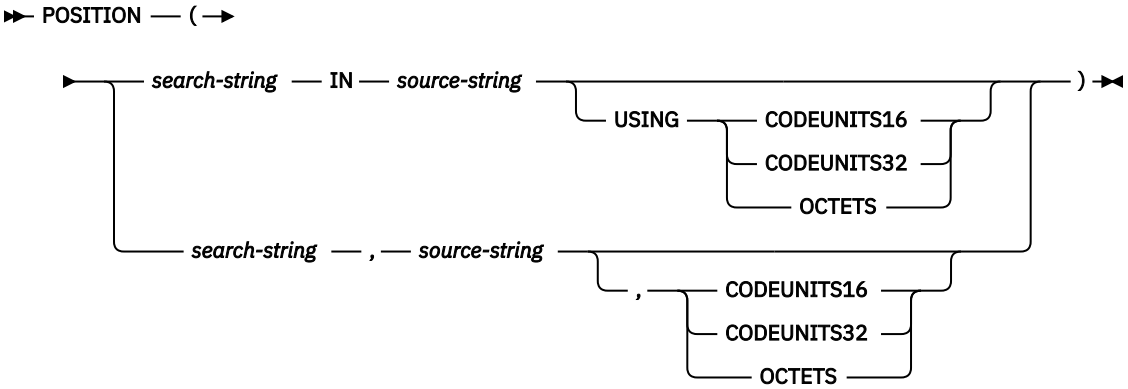
In the following example, the db2-fn:sqlquery function call uses one PARAMETER function call and the XQuery expression `$po/@OrderDate`, the order date attribute. The PARAMETER function call returns the value of order date attribute:

```
xquery
declare default element namespace "http://posample.org";
for $po in db2-fn:xmlcolumn('PURCHASEORDER.PORDER')/PurchaseOrder,
  $item in $po/item/partid
for $p in db2-fn:sqlquery(
  "select description from product where promostart < PARAMETER(1)",
    $po/@OrderDate )
where $p//@pid = $item
return
<RESULT>
  <PoNum>{data($po/@PoNum)}</PoNum>
  <PartID>{data($item)} </PartID>
  <PoDate>{data($po/@OrderDate)}</PoDate>
</RESULT>
```

The example returns the purchase ID, part ID, and the purchase date for all the parts sold after the promotional start date.

POSITION

The POSITION function returns the starting position of the first occurrence of one string within another string.



The schema is SYSIBM.

The string the POSITION function searches for is called the *search-string*. The string it searches in is called the *source-string*. The POSITION function returns the starting position of the first occurrence of one string (called the *search-string*) within another string (called the *source-string*). If *search-string* is not found and neither argument is null, the result is zero. If the *search-string* is found, the result is a number from 1 to the actual length of *source-string*, expressed in the string unit that is explicitly specified. The

search is done using the collation of the database, unless *search-string* or *source-string* is defined as a binary string or as FOR BIT DATA, in which case the search is done using a binary comparison.

If *source-string* has an actual length of 0, the result of the function is 0. If *search-string* has an actual length of 0 and *source-string* is not null, the result of the function is 1.

search-string
An expression that specifies the string that is to be searched for. This expression must return a value that is a built-in character string, graphic string, binary string, numeric value, Boolean value, or datetime value. If the value is not a character string, graphic string, or binary string, it is implicitly cast to VARCHAR before the function is evaluated. The expression cannot be a LOB file reference variable.

source-string
An expression that specifies the string to be searched through. This expression must return a value that is a built-in character string, numeric value, Boolean value, or datetime value. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function.

CODEUNITS16 or CODEUNITS32 or OCTETS
Specifies the string unit of the result. CODEUNITS16 specifies that the result is to be expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that the result is to be expressed in 32-bit UTF-32 code units. OCTETS specifies that the result is to be expressed in bytes.

If a string unit is specified as CODEUNITS16 or CODEUNITS32, and *search-string* or *source-string* is a binary string or a FOR BIT DATA string, an error is returned (SQLSTATE 428GC).

If a string unit argument is not specified and both *search-string* and *source-string* are either a character string that is not FOR BIT DATA or is a graphic string, the default is CODEUNITS32. Otherwise, the default is OCTETS.

If a locale-sensitive UCA-based collation is used for this function, then the CODEUNITS16 option offers the best performance characteristics.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

The first and second arguments must have compatible string types. For more information about compatibility, see "Rules for string conversions". In a Unicode database, if one string argument is character (not FOR BIT DATA) and the other string argument is graphic, then the *search-string* is converted to the data type of the *source-string* for processing. If one argument is character FOR BIT DATA, the other argument must not be graphic (SQLSTATE 42846).

Result

The result of the function is a large integer. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Select the RECEIVED column, the SUBJECT column, and the starting position of the string 'GOOD BEER' within the NOTE_TEXT column for all rows in the IN_TRAY table that contain that string.

```
SELECT RECEIVED, SUBJECT, POSITION('GOOD BEER', NOTE_TEXT, OCTETS)
FROM IN_TRAY
WHERE POSITION('GOOD BEER', NOTE_TEXT, OCTETS) <> 0
```

- *Example 2*: Find the position of the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable LOCATION with the position, as measured in CODEUNITS32 units, within the string.

```
SET :LOCATION = POSITION(
  'ß', 'Jürgen lives on Hegelstraße', CODEUNITS32
)
```

The value of host variable LOCATION is set to 26.

- *Example 3*: Find the position of the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable LOCATION with the position, as measured in OCTETS, within the string.

```
SET :LOCATION = POSITION(
  'ß', 'Jürgen lives on Hegelstraße', OCTETS
)
```

  The value of host variable LOCATION is set to 27.

- *Example 4*: The following examples work with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the non-spacing combining tilde character. This string is shown in different Unicode encoding forms in the following example:

	'&'	'N'	'~'	'A'	'B'
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'

  Assume that the variable UTF8_VAR contains the UTF-8 representation of the string.

```
SELECT POSITION('N', UTF8_VAR, CODEUNITS16),
  POSITION('N', UTF8_VAR, CODEUNITS32),
  POSITION('N', UTF8_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

  returns the values 3, 2, and 5, respectively.

  Assume that the variable UTF16_VAR contains the UTF-16BE representation of the string.

```
SELECT POSITION('B', UTF16_VAR, CODEUNITS16),
   POSITION('B', UTF16_VAR, CODEUNITS32),
   POSITION('B', UTF16_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

  returns the values 6, 5, and 11, respectively.

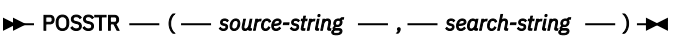
- *Example 5*: In a Unicode database created with the case insensitive collation CLDR181_LEN_S1, find the position of the word 'Brown' in the phrase 'The quick brown fox'.

```
SET :LOCATION = POSITION('Brown', 'The quick brown fox', CODEUNITS16)
```

  The value of the host variable LOCATION is set to 11.

POSSTR

The POSSTR function returns the starting position of the first occurrence of one string (called the *search-string*) within another string (called the *source-string*).



The schema is SYSIBM.

Numbers for the *search-string* position start at 1 (not 0).

The result of the function is a large integer. If either of the arguments can be null, the result can be null; if either of the arguments is null, the result is the null value.

source-string
An expression that specifies the string to be searched through. This expression must return a built-in character string, numeric value, Boolean value, or datetime value. If the value is not a character string, it is implicitly cast to VARCHAR before the function is evaluated.

search-string
An expression that specifies the string that is to be searched for. This expression must return a value that is a built-in string, signed numeric, Boolean, or datetime value. If the value is not a string, it is

implicitly cast to VARCHAR before the function is evaluated. The actual length must not exceed the maximum length of a VARCHAR.

The expression cannot include any of the following elements (SQLSTATE 42824):

- A LOB file reference variable
- A parameter of an inlined SQL user-defined function
- A transition variable in an inlined trigger
- A local variable in a compound SQL (inlined) statement
- A user-defined function
- A non-deterministic built-in function
- A scalar fullselect

In a Unicode database, if one argument is character (not FOR BIT DATA) and the other argument is graphic, then the *search-string* is converted to the data type of the *source-string* for processing. If one argument is character FOR BIT DATA, the other argument must not be graphic (SQLSTATE 42846).

Both *search-string* and *source-string* have zero or more contiguous positions. If the strings are character or binary strings, a position is a byte. If the strings are graphic strings, a position is a double byte. POSSTR operates on a strict byte-count basis, without awareness of either the database collation or changes between single and multi-byte characters. The POSITION, LOCATE, or LOCATE_IN_STRING functions can be used to operate with awareness of the database collation and the string units.

The following rules apply:

- The data types of *source-string* and *search-string* must be compatible, otherwise an error is raised (SQLSTATE 42884).
  - If *source-string* is a character string, then *search-string* must be a character string, but not a CLOB, with an actual length of 32672 bytes or less.
  - If *source-string* is a graphic string, then *search-string* must be a graphic string, but not a DBCLOB, with an actual length of 16336 double-byte characters or less.
  - If *source-string* is a binary string, then *search-string* must be a binary string with an actual length of 32672 bytes or less.
- If *search-string* has a length of zero, the result returned by the function is 1.
- Otherwise:
  - If *source-string* has a length of zero, the result returned by the function is zero.
  - Otherwise:
    - If the value of *search-string* is equal to an identical length substring of contiguous positions from the value of *source-string*, then the result returned by the function is the starting position of the first such substring within the *source-string* value.
    - Otherwise, the result returned by the function is 0.

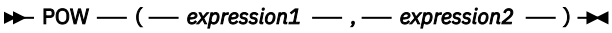
Example

Select RECEIVED and SUBJECT columns as well as the starting position of the words 'GOOD BEER' within the NOTE_TEXT column for all entries in the IN_TRAY table that contain these words.

```
SELECT RECEIVED, SUBJECT, POSSTR(NOTE_TEXT, 'GOOD BEER')
  FROM IN_TRAY
  WHERE POSSTR(NOTE_TEXT, 'GOOD BEER') <> 0
```

POW

The POW function returns the result of raising the first argument to the power of the second argument.

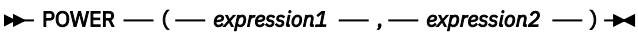


The schema is SYSIBM.

POW is a synonym for POWER.

POWER

The POWER function returns the result of raising the first argument to the power of the second argument.



The schema is SYSIBM. (The SYSFUN version of the POWER function continues to be available.)

expression1
An expression that returns a value of any built-in numeric data type.

expression2
An expression that returns a value of any built-in numeric data type.

If the value of expression1 is equal to zero, then expression2 must be greater than or equal to zero. If both arguments are 0, the result is 1. If the value of expression1 is less than zero, then expression2 must be an integer value.

The result of the function is:

- INTEGER if both arguments are INTEGER or SMALLINT
- BIGINT if one argument is BIGINT and the other argument is BIGINT, INTEGER or SMALLINT
- DECFLOAT(34) if one of the arguments is decimal floating-point. If either argument is a DECFLOAT and one of the following statements is true, the result is NAN and an invalid operation condition:
  - Both arguments are zero
  - The second argument has a nonzero fractional part
  - The second argument has more than 9 digits
  - The second argument is INFINITY
- DOUBLE otherwise

If the argument is a special decimal floating-point value, the rules for general arithmetic operations for decimal floating-point apply. See "General arithmetic operation rules for decimal floating-point" on page 142 in "Expressions" on page 132.

The result can be null; if any argument is null, the result is the null value.

Example

Assume that the host variable HPOWER is an integer with a value of 3.

```
VALUES POWER(2,:HPOWER)
```

Returns the value 8.

QUANTIZE

The QUANTIZE function returns a decimal floating-point value that is equal in value (except for any rounding) and sign to *numeric-expression* and that has an exponent equal to the exponent of *exp-expression*.

The number of digits (16 or 34) is the same as the number of digits in *numeric-expression*.

The schema is SYSIBM.

numeric-expression
An expression that returns a value of any built-in numeric data type. If the argument is not a decimal floating-point value, it is converted to DECFLOAT(34) for processing.

exp-expression
An expression that returns a value of any built-in numeric data type. If the argument is not a decimal floating-point value, it is converted to DECFLOAT(34) for processing. The *exp-expression* is used as an example pattern for rescaling *numeric-expression*. The sign and coefficient of *exp-expression* are ignored.

The coefficient of the result is derived from that of *numeric-expression*. It is rounded, if necessary (if the exponent is being increased), multiplied by a power of ten (if the exponent is being decreased), or remains unchanged (if the exponent is already equal to that of *exp-expression*).

The CURRENT DECFLOAT ROUNDING MODE special register determines the rounding mode.

Unlike other arithmetic operations on the decimal floating-point data type, if the length of the coefficient after the quantize operation is greater than the precision specified by *exp-expression*, the result is NaN and a warning is returned (SQLSTATE 0168D). This ensures that, unless there is a warning condition, the exponent of the result of QUANTIZE is always equal to that of *exp-expression*.

- if either argument is NaN, NaN is returned
- if either argument is sNaN, NaN is returned and a warning is returned (SQLSTATE 01565)
- if both arguments are infinity (positive or negative), infinity with the same sign as the first argument is returned
- if one argument is infinity (positive or negative) and the other argument is not infinity, NaN is returned and a warning is returned (SQLSTATE 0168D)

The result of the function is a DECFLOAT(16) value if both arguments are DECFLOAT(16). Otherwise, the result of the function is a DECFLOAT(34) value. The result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: The following examples show the values that are returned by the QUANTIZE function given a variety of input decimal floating-point values and assuming a rounding mode of ROUND_HALF_UP:

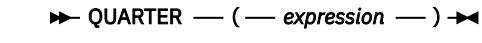
```
QUANTIZE(2.17, DECFLOAT(0.001)) = 2.170
QUANTIZE(2.17, DECFLOAT(0.01)) = 2.17
QUANTIZE(2.17, DECFLOAT(0.1)) = 2.2
QUANTIZE(2.17, DECFLOAT('1E+0')) = 2
QUANTIZE(2.17, DECFLOAT('1E+1')) = 0E+1
QUANTIZE(2, DECFLOAT(INFINITY)) = NaN    -- warning
QUANTIZE(0, DECFLOAT('1E+5')) = 0E+5
QUANTIZE(217, DECFLOAT('1E-1')) = 217.0
QUANTIZE(217, DECFLOAT('1E+0')) = 217
QUANTIZE(217, DECFLOAT('1E+1')) = 2.2E+2
QUANTIZE(217, DECFLOAT('1E+2')) = 2E+2
```

- *Example 2*: In the following example the value -0 is returned for the QUANTIZE function. The CHAR function is used to avoid the potential removal of the minus sign by a client program:

```
CHAR(QUANTIZE(-0.1, DECFLOAT(1))) = -0
```

QUARTER

Returns an integer value in the range 1 to 4, representing the quarter of the year for the date specified in the argument.



The schema is SYSFUN.

expression
An expression that returns a value of one of the following built-in data types: DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before the function is executed.

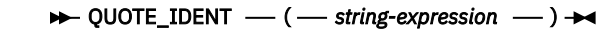
The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

QUOTE_IDENT

The QUOTE_IDENT function returns a string that can be used as an identifier in an SQL statement.

The schema is SYSIBM.

Syntax



string-expression

An expression that specifies the input string. The expression must return a character string, signed numeric value, or datetime value. If the data type of the input string is not VARCHAR, it is implicitly cast to VARCHAR before the function is evaluated. If the data type of the input string is CHAR or VARCHAR, the expression must not be FOR BIT DATA (SQLSTATE 42815).

Result

The data type of the result is VARCHAR, and the result has the same codepage and string units as the input string. The length attribute of the result depends on the length attribute of the input string:

Length attribute of the input string	Length attribute of the result
L OCTETS	`min(32672,(L*2)+2)` OCTETS
L CODEUNITS32	`min(8168,(L*2)+2)` CODEUNITS32

If the argument can be null, the result can be null. If the argument is null, the result is the null value.

Examples

Statement	Result
`quote_ident('HELLO WORLD')`	`"HELLO WORLD"`
`quote_ident('HELLOWORLD')`	`HELLOWORLD`
`quote_ident('HELLO_WORLD')`	`HELLO_WORLD`
`quote_ident('hello world')`	`"hello world"`
`quote_ident('hello"world')`	`"hello""world"`
`quote_ident('hello''world')`	`"hello'world"`

Statement	Result
`quote_ident('')`	`""`

Related reference

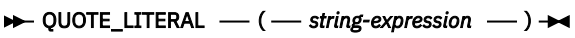
“Identifiers” on page 5
An *identifier* is a token that is used to form a name. An identifier in an SQL statement is either an SQL identifier or a host identifier.

QUOTE_LITERAL

The QUOTE_LITERAL function returns a string that can be used as a string constant in an SQL statement.

The schema is SYSIBM.

Syntax



string-expression

An expression that specifies the input string. The expression must return a character string, signed numeric value, or datetime value. If the data type of the input string is not VARCHAR, it is implicitly cast to VARCHAR before the function is evaluated. If the data type of the input string is CHAR or VARCHAR, the expression must not be FOR BIT DATA (SQLSTATE 42815).

Result

The data type of the result is VARCHAR, and the result has the same codepage and string units as the input string. The length attribute of the result depends on the length attribute of the input string:

Length attribute of the input string	Length attribute of the result
L OCTETS	`min(32672,(L*2)+2)` OCTETS
L CODEUNITS32	`min(8168,(L*2)+2)` CODEUNITS32

If the argument can be null, the result can be null. If the argument is null, the result is the null value.

Examples

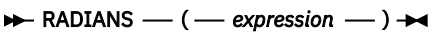
Statement	Result
`quote_literal(42.5)`	`'42.5'`
`quote_literal('You''re here!')`	`'You''re here!'`

Related reference

“Identifiers” on page 5
An *identifier* is a token that is used to form a name. An identifier in an SQL statement is either an SQL identifier or a host identifier.

RADIANS

The RADIANS function returns the number of radians for an argument that is expressed in degrees.



The schema is SYSIBM. (The SYSFUN version of the RADIANS function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type. If the argument is decimal floating-point, the operation is performed in decimal floating-point; otherwise, the argument is converted to double-precision floating-point for processing by the function.

If the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*); otherwise, the result is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example

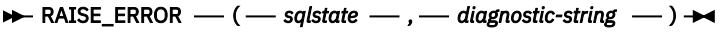
Assume that host variable HDEG is an INTEGER with a value of 180. The following statement:

```
VALUES RADIANS(:HDEG)
```

Returns the value +3.14159265358979E+000.

RAISE_ERROR

The RAISE_ERROR function causes the statement that includes the function to return an error with the specified SQLSTATE, SQLCODE -438, and *diagnostic-string*.



The schema is SYSIBM.

sqlstate
A character string containing exactly 5 bytes. It must be of type CHAR defined with a length of 5 or type VARCHAR defined with a length of 5 or greater. In a Unicode database, the expression can also return a graphic string. If the returned value is not a character string, it is cast to a character string before the function is evaluated.

The *sqlstate* value must obey the following rules for application-defined SQLSTATEs:

- Each character must be from the set of digits ("0" through "9") or non-accented upper case letters ("A" through "Z")
- The SQLSTATE class (first two characters) cannot be "00", "01", or "02" because these are not error classes.
- If the SQLSTATE class (first two characters) starts with the character "0" through "6" or "A" through "H", then the subclass (last three characters) must start with a letter in the range "I" through "Z".
- If the SQLSTATE class (first two characters) starts with the character "7", "8", "9" or "I" though "Z", then the subclass (last three characters) can be any of "0" through "9" or "A" through "Z".

If the SQLSTATE does not conform to these rules, an error occurs (SQLSTATE 428B3).

diagnostic-string
An expression that returns a character string that describes the error condition, or a Boolean value. In a Unicode database, the expression can also return a graphic string. If the returned value is not a character string, it is cast to a character string before the function is evaluated. If the string exceeds 70 bytes, it is truncated.

Result

The RAISE_ERROR function always returns the null value with an undefined data type. To use this function in a context where the data type cannot be determined, a cast specification must be used to give the null returned value a data type. A CASE expression is where the RAISE_ERROR function will be most useful.

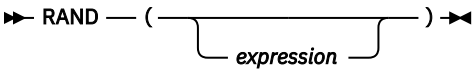
Example

List employee numbers and education levels as Post Graduate, Graduate and Diploma. If an education level is greater than 20, raise an error.

```
SELECT EMPNO,
  CASE WHEN EDUCLVL < 16 THEN 'Diploma'
       WHEN EDUCLVL < 18 THEN 'Graduate'
       WHEN EDUCLVL < 21 THEN 'Post Graduate'
       ELSE RAISE_ERROR('70001',
         'EDUCLVL has a value greater than 20')
  END
FROM EMPLOYEE
```

RAND (SYSFUN schema)

The RAND function returns a floating point value between 0 and 1. RAND is a non-deterministic function.



The schema is SYSFUN.

expression
An expression that returns a value of data type SMALLINT, INTEGER, or BOOLEAN. The value must be between 0 and 2,147,483,647. The returned value is used as a seed value.

A specific seed value produces the same sequence of random numbers for a specific instance of a RAND function in a query each time the query is executed. The seed value is used only for the first invocation of an instance of the RAND function within a statement. If a seed value is not specified, a different sequence of random numbers is produced each time the query is executed within the same session. To produce a set of random numbers that varies from session to session, specify a random seed; for example, one that is based on the current time.

The RAND scalar function does not guarantee the uniqueness of the random numbers. Use the GENERATE_UNIQUE scalar function to generate a series unique numbers.

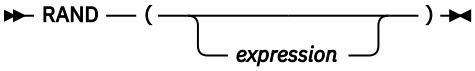
The RAND function relies on the random number facilities of the host operating system. The random number facility of each host might vary in factors such as the number of potential distinct values and the quality of the randomness. For these reasons, the output of this function is not suitable as a source of randomness in a cryptographic system.

Result

The data type of the result is double-precision floating point. If the argument is null, the result is the null value. If either argument can be null, the result can be null. If either argument is null, the result is the null value.

RAND (SYSIBM schema)

The RAND function returns a floating point value between 0 and 1. RAND is a non-deterministic function.



The schema is SYSIBM.

expression
The expression represents the value of a seed which is used to initialize the random number generator. The expression must return a built-in character string, Boolean value, or numeric value. In a Unicode database, the expression can also return a graphic string, in which case it is first converted to

a character string before the function is evaluated. If the returned value is not a INTEGER, it is cast to INTEGER before evaluating the function. The value must be between 0 and 2,147,483,647.

A specific seed value produces the same sequence of random numbers for a specific instance of a RAND function in a query each time the query is executed. The seed value is used only for the first invocation of an instance of the RAND function within a statement. If a seed value is not specified, a different sequence of random numbers is produced each time the query is executed within the same session. To produce a set of random numbers that varies from session to session, specify a random seed; for example, one that is based on the current time.

The RAND scalar function does not guarantee the uniqueness of the random numbers. Use the GENERATE_UNIQUE scalar function to generate a series unique numbers.

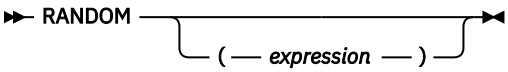
The RAND function relies on the random number facilities of the host operating system. The random number facility of each host might vary in factors such as the number of potential distinct values and the quality of the randomness. For these reasons, the output of this function is not suitable as a source of randomness in a cryptographic system.

Result

The data type of the result is double-precision floating point. If the argument is null, the result is the null value. If either argument can be null, the result can be null. If either argument is null, the result is the null value.

RANDOM

The RANDOM function returns a floating point value between 0 and 1. RANDOM is a non-deterministic function.

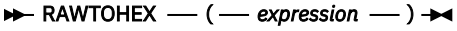


The schema is SYSIBM.

The RANDOM scalar function is a synonym for the “RAND (SYSIBM schema) ” on page 453.

RAWTOHEX

The RAWTOHEX function returns a hexadecimal representation of a value as a character string.



The schema is SYSIBM.

expression
An expression that specifies the string for which the hexadecimal value is to be returned. The expression must return a built-in character string, graphic string, binary string, numeric value, Boolean value, or datetime value. If the value is not a character, graphic, or, binary string, it is implicitly cast to VARCHAR before the function is evaluated.

Result

The result of the function is VARCHAR. If the argument can be null, the result can be null. If the argument is null, the result is the null value. The length of the result is computed based on the following table:

Table 82. Data type of the result as a function of the data types of the argument data type and the length attribute

Argument data type[1]	**Length attribute**[2]	**Result data type**
CHAR(A) or BINARY(A)	A<128	CHAR(A*2)

Table 82. Data type of the result as a function of the data types of the argument data type and the length attribute (continued)

Argument data type[1]	Length attribute[2]	Result data type
CHAR(A) or BINARY(A)	A>127	VARCHAR(A*2)
VARCHAR(A), VARBINARY(A), CLOB(A), or BLOB(A)	A<16337	VARCHAR(A*2)
GRAPHIC(A)	A<64	CHAR(A*2*2)
GRAPHIC(A)	A>63	VARCHAR(A*2*2)
VARGRAPHIC(A) or DBCLOB(A)	A<8169	VARCHAR(A*2*2)
CHAR(A CODEUNITS32)	A<64	VARCHAR(A*4*2)
VARCHAR(A CODEUNITS32) or CLOB(A CODEUNITS32)	A<4085	VARCHAR(A*4*2)
GRAPHIC(A CODEUNITS32)	A<64	VARCHAR(A*2*2*2)
VARGRAPHIC(A CODEUNITS32) or DBCLOB(A CODEUNITS32)	A<4085	VARCHAR(A*2*2*2)

1. If string units are not specified, then the string units for the data type are not CODEUNITS32.
2. The maximum length attributes reflect a data type limit or the limit of 16336 bytes for the input argument.

Example

```
values rawtohex('hello')
    returns 68656C6C6F
```

REAL

The REAL function returns a single-precision floating-point representation of either a number or a string representation of a number.

Numeric to REAL

String to REAL

The schema is SYSIBM.

Numeric to REAL

numeric-expression
An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a single-precision floating-point column or variable. If the numeric value of the argument is not within the range of single-precision floating-point, an error is returned (SQLSTATE 22003).

String to REAL

string-expression
An expression that returns a value that is character-string or Unicode graphic-string representation of a number. The data type of string-expression must not be a CLOB or a DBCLOB (SQLSTATE 42884).

The result is the same number that would result from CAST(string-expression AS REAL). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming a valid numeric constant (SQLSTATE 22018). If the numeric value of the argument is not within the range of single-precision floating-point, an error is returned (SQLSTATE 22003).

The result of the function is a single-precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Notes

- The CAST specification should be used to increase the portability of applications.

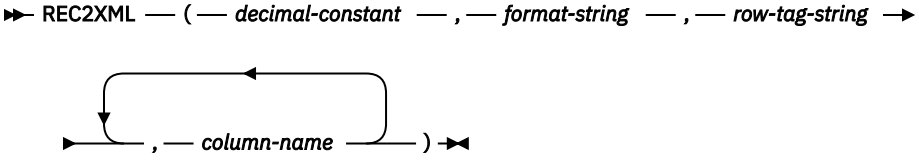
Example

Using the EMPLOYEE table, find the ratio of salary to commission for employees whose commission is not zero. The columns involved (SALARY and COMM) have DECIMAL data types. The required result is in single-precision floating point. Therefore, REAL is applied to SALARY so that the division is carried out in floating point (actually double-precision) and then REAL is applied to the complete expression to return the result in single-precision floating point.

```
SELECT EMPNO, REAL(REAL(SALARY)/COMM)
  FROM EMPLOYEE
  WHERE COMM > 0
```

REC2XML

The REC2XML function returns a string formatted with XML tags, containing column names and column data.



The schema is SYSIBM.

In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

decimal-constant
The expansion factor for replacing column data characters. The decimal value must be greater than 0.0 and less than or equal to 6.0. (SQLSTATE 42820).

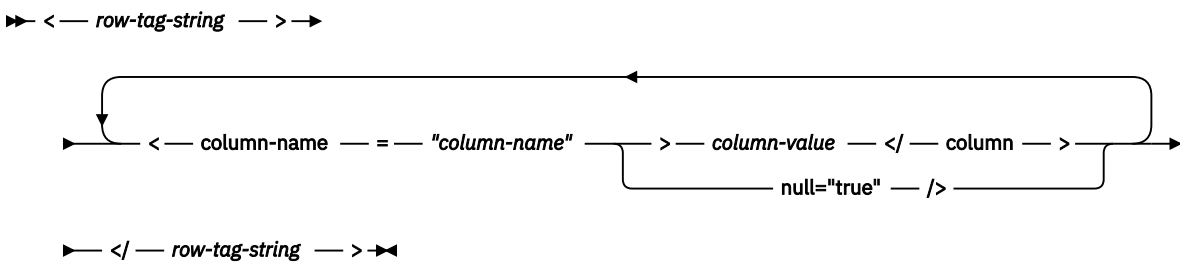
The *decimal-constant* value is used to calculate the result length of the function. For every column with a character data type, the length attribute of the column is multiplied by this expansion factor before it is added in to the result length.

To specify no expansion, use a value of 1.0. Specifying a value less than 1.0 reduces the calculated result length. If the actual length of the result string is greater than the calculated result length of the function, then an error is raised (SQLSTATE 22001).

format-string
The string constant that specifies which format the function is to use during execution.

The *format-string* is case-sensitive, so the following values must be specified in uppercase to be recognized.

COLATTVAL or COLATTVAL_XML
These formats return a string with columns as attribute values.



Column names may or may not be valid XML attribute values. For column names which are not valid XML attribute values, character replacement is performed on the column name before it is included in the result string.

Column values may or may not be valid XML element names. If the *format-string* COLATTVAL is specified, then for the column names which are not valid XML element values, character replacement is performed on the column value before it is included in the result string. If the *format-string* COLATTVAL_XML is specified, then character replacement is not performed on column values (although character replacement is still performed on column names).

row-tag-string
A string constant that specifies the tag used for each row. If an empty string is specified, then a value of "row" is assumed.

If a string of one or more blank characters is specified, then no beginning *row-tag-string* or ending *row-tag-string* (including the angle bracket delimiters) will appear in the result string.

column-name
A qualified or unqualified name of a table column. The column must have one of the following data types (SQLSTATE 42815):

- numeric (SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, DOUBLE)
- character string (CHAR, VARCHAR; a character string with a subtype of BIT DATA is not allowed)
- datetime (DATE, TIME, TIMESTAMP)
- a user-defined type based on one of the previously listed data types

The same column name cannot be specified more than once (SQLSTATE 42734).

The result of the function is VARCHAR in OCTETS, regardless of the string units of the environment or the specified columns. The maximum length is 32 672 bytes (SQLSTATE 54006).

Consider the following invocation:

```
REC2XML (dc, fs, rt, c1, c2, ..., cn)
```

If the value of "fs" is either "COLATTVAL" or "COLATTVAL_XML", then the result is the same as this expression:

```
'<' CONCAT rt CONCAT '>' CONCAT y1 CONCAT y2
CONCAT ... CONCAT yn CONCAT '</' CONCAT rt CONCAT '>'
```

where y_n is equivalent to:

```
'<column name="' CONCAT xvcn CONCAT vn
```

and vn is equivalent to:

```
'">' CONCAT rn CONCAT '</column>'
```

if the column is not null, and

```
'" null="true"/>'
```

if the column value is null.

xvc_n is equivalent to a string representation of the column name of c_n, where any characters appearing in Table 84 on page 458 are replaced with the corresponding representation. This ensures that the resulting string is a valid XML attribute or element value token.

The r_n is equivalent to a string representation as indicated in Table 83 on page 458

Table 83. Column Values String Result	
Data type of c_n	**r_n**
CHAR, VARCHAR	The value is a string. If the *format-string* does not end in the characters "_XML", then each character in c_n is replaced with the corresponding replacement representation from Table 84 on page 458, as indicated. The length attribute is: dc * the length attribute of c_n.
SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC, REAL, DOUBLE	The value is LTRIM(RTRIM(CHAR(c_n))). The length attribute is the result length of CHAR(c_n). The decimal character is always the period (".") character.
DATE	The value is CHAR(c_n,ISO). The length attribute is the result length of CHAR(c_n,ISO).
TIME	The value is CHAR(c_n,JIS). The length attribute is the result length of CHAR(c_n,JIS)
TIMESTAMP	The value is CHAR(c_n). The length attribute is the result length of CHAR(c_n).

Character replacement:

Depending on the value specified for the *format-string*, certain characters in column names and column values will be replaced to ensure that the column names form valid XML attribute values and the column values form valid XML element values.

Table 84. Character Replacements for XML Attribute Values and Element Values	
Character	**Replacement**
<	&lt;
>	&gt;
"	&quot;
&	&amp;
'	&apos;

Examples

Note: REC2XML does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- *Example 1*: Using the DEPARTMENT table in the sample database, format the department table row, except the DEPTNAME and LOCATION columns, for department 'D01' into an XML string. Since the data

does not contain any of the characters which require replacement, the expansion factor will be 1.0 (no expansion). Also note that the MGRNO value is null for this row.

```
SELECT REC2XML (1.0, 'COLATTVAL', '', DEPTNO, MGRNO, ADMRDEPT)
  FROM DEPARTMENT
  WHERE DEPTNO = 'D01'
```

This example returns the following VARCHAR(117) string:

```
<row>
<column name="DEPTNO">D01</column>
<column name="MGRNO" null="true"/>
<column name="ADMRDEPT">A00</column>
</row>
```

- *Example 2*: A 5-day university schedule introduces a class named "&43<FIE" to a table called CL_SCHED, with a new format for the CLASS_CODE column. Using the REC2XML function, this example formats an XML string with this new class data, except for the class end time.

  The length attribute for the REC2XML call with an expansion factor of 1.0 would be 128 (11 for the "<row>" and "</row>" overhead, 21 for the column names, 75 for the "<column name=", ">", "</column>" and double quotation marks, 7 for the CLASS_CODE data, 6 for the DAY data, and 8 for the STARTING data). Since the "&" and "<" characters will be replaced, an expansion factor of 1.0 will not be sufficient. The length attribute of the function will need to support an increase from 7 to 14 bytes for the new format CLASS_CODE data.

  However, since it is known that the DAY value will never be more than 1 digit long, an unused extra 5 units of length are added to the total. Therefore, the expansion only needs to handle an increase of 2. Since CLASS_CODE is the only character string column in the argument list, this is the only column data to which the expansion factor applies. To get an increase of 2 for the length, an expansion factor of 9/7 (approximately 1.2857) would be needed. An expansion factor of 1.3 will be used.

  ```
  SELECT REC2XML (1.3, 'COLATTVAL', 'record', CLASS_CODE, DAY, STARTING)
    FROM CL_SCHED
    WHERE CLASS_CODE = '&43<FIE'
  ```

  This example returns the following VARCHAR(167) string:

  ```
  <record>
  <column name="CLASS_CODE">&amp;43&lt;FIE</column>
  <column name="DAY">5</column>
  <column name="STARTING">06:45:00</column>
  </record>
  ```

- *Example 3*: Assume that new rows have been added to the EMP_RESUME table in the sample database. The new rows store the resumes as strings of valid XML. The COLATTVAL_XML *format-string* is used so character replacement will not be carried out. None of the resumes are more than 3500 bytes in length. The following query is used to select the XML version of the resumes from the EMP_RESUME table and format it into an XML document fragment.

  ```
  SELECT REC2XML (1.0, 'COLATTVAL_XML', 'row', EMPNO, RESUME_XML)
    FROM (SELECT EMPNO, CAST(RESUME AS VARCHAR(3500)) AS RESUME_XML
      FROM EMP_RESUME
      WHERE RESUME_FORMAT = 'XML')
    AS EMP_RESUME_XML
  ```

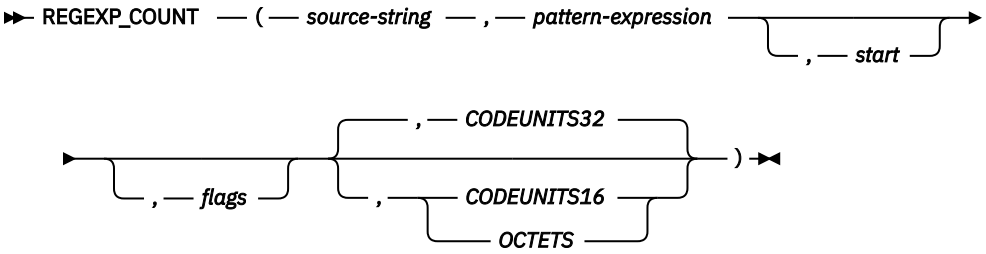
  This example returns a row for each employee who has a resume in XML format. Each returned row will be a string with the following format:

  ```
  <row>
  <column name="EMPNO">{employee number}</column>
  <column name="RESUME_XML">{resume in XML}</column>
  </row>
  ```

  Where "{employee number}" is the actual EMPNO value for the column and "{resume in XML}" is the actual XML fragment string value that is the resume.

REGEXP_COUNT

The REGEXP_COUNT scalar function returns a count of the number of times that a regular expression pattern is matched in a string.



The schema is SYSIBM.

source-string
An expression that specifies the string in which the search is to take place. This expression must return a built-in character string, graphic string, numeric value, Boolean value, or datetime value. A numeric, Boolean, or datetime value is implicitly cast to VARCHAR before the function is evaluated. A character string cannot specify the FOR BIT DATA attribute (SQLSTATE 42815).

pattern-expression
An expression that specifies the regular expression string that is the pattern for the search. This expression must return a built-in character string, graphic string, numeric value, Boolean value, or datetime value. A numeric, Boolean, or datetime value is implicitly cast to VARCHAR before the function is evaluated. The length of a CLOB or DBCLOB expression must not be greater than the maximum length of a VARCHAR or VARGRAPHIC data type. A character string cannot specify the FOR BIT DATA attribute (SQLSTATE 42815).

start
An expression that specifies the position within *source-string* at which the search is to start. The expression must return a built-in character string, graphic string, Boolean, or numeric value. If the value is not of type INTEGER, it is implicitly cast to INTEGER before the function is evaluated. The value of the integer must be greater than or equal to 1. If OCTETS is specified and the source string is graphic data, the value of the integer must be odd (SQLSTATE 428GC). The default start value is 1. See parameter description for CODEUNITS16, CODEUNITS32, or OCTETS for the string unit that applies to the start position.

flags
An expression that specifies flags that controls aspects of the pattern matching. The expression must return a built-in character string that does not specify the FOR BIT DATA attribute (SQLSTATE 42815). The string can include one or more valid flag values and the combination of flag values must be valid (SQLSTATE 2201T). An empty string is the same as the value 'c'. The default flag value is 'c'.

Table 85. Supported flag values

Flag value	Description
c	Specifies that matching is case-sensitive. This flag is the default value if 'c' or 'i' is not specified. This value must not be specified with a value of 'i'.
i	Specifies that matching is case insensitive. This value must not be specified with a value of 'c'.
m	Specifies that the input data can contain more than one line. By default, the '^' in a pattern matches only the start of the input string; the '$' in a pattern matches only the end of the input string. If this flag is set, "^" and "$" also matches at the start and end of each line within the input string.

Table 85. Supported flag values (continued)	
Flag value	**Description**
n	Specifies that the '.' character in a pattern matches a line terminator in the input string. By default, the '.' character in a pattern does not match a line terminator. A carriage-return and line-feed pair in the input string behaves as a single-line terminator, and matches a single "." in a pattern.
s	Specifies that the '.' character in a pattern matches a line terminator in the input string. This value is a synonym for the 'n' value.
x	Specifies that white space characters in a pattern are ignored, unless escaped.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of the start value:

- CODEUNITS16 specifies that the start value is expressed in 16-bit UTF-16 code units.
- CODEUNITS32 specifies that the start value is expressed in 32-bit UTF-32 code units. This is the default.
- OCTETS specifies that the start value is expressed in bytes.

If the string unit is specified as CODEUNITS16 or OCTETS, and if the string unit of the source string is CODEUNITS32, an error is returned (SQLSTATE 428GC).

For more information, see "String units in built-in functions" in “Character strings” on page 31.

Result

The result of the function is an INTEGER that represents the number of occurrences of the pattern expression within the source string. If the pattern expression is not found and no argument is null, the result is 0.

If any argument of the REGEXP_COUNT function can be null, the result can be null. If any argument is null, the result is the null value.

Notes

- The regular expression processing is done by using the International Components for Unicode (ICU) regular expression interface.
- **Considerations for non-Unicode databases**:
  - A regular expression pattern supports only half-width control characters; use a character string data type for the pattern expression argument. A character string data type can be used for the pattern expression argument even when a graphic string data type is used for the source string argument.
  - The source string argument must be a graphic string data type if the pattern expression argument is a graphic string data type.

Example

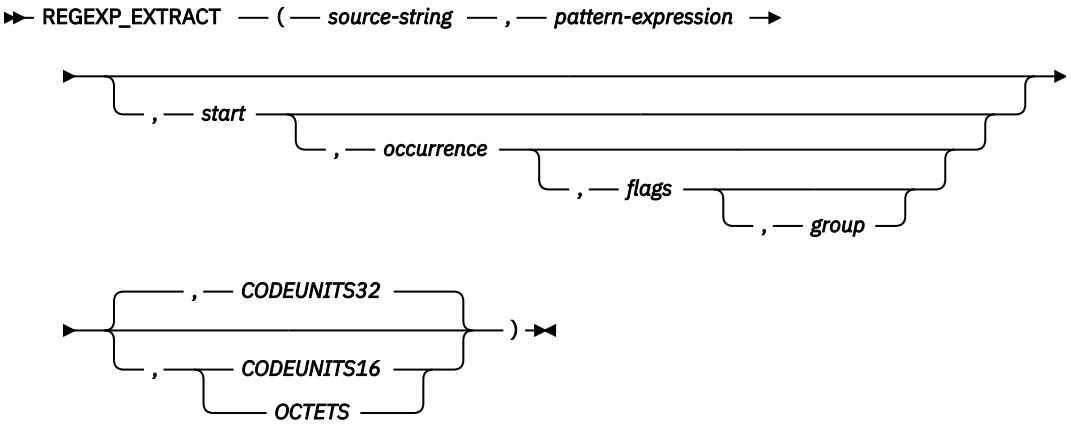
Count the number of times "Steven" or "Stephen" occurs in the string "Steven Jones and Stephen Smith are the best players".

```
SELECT REGEXP_COUNT(
   'Steven Jones and Stephen Smith are the best players', 'Ste(v|ph)en')
     FROM sysibm.sysdummy1
```

The result is 2.

REGEXP_EXTRACT

The REGEXP_EXTRACT scalar function returns one occurrence of a substring of a string that matches the regular expression pattern.



The schema is SYSIBM.

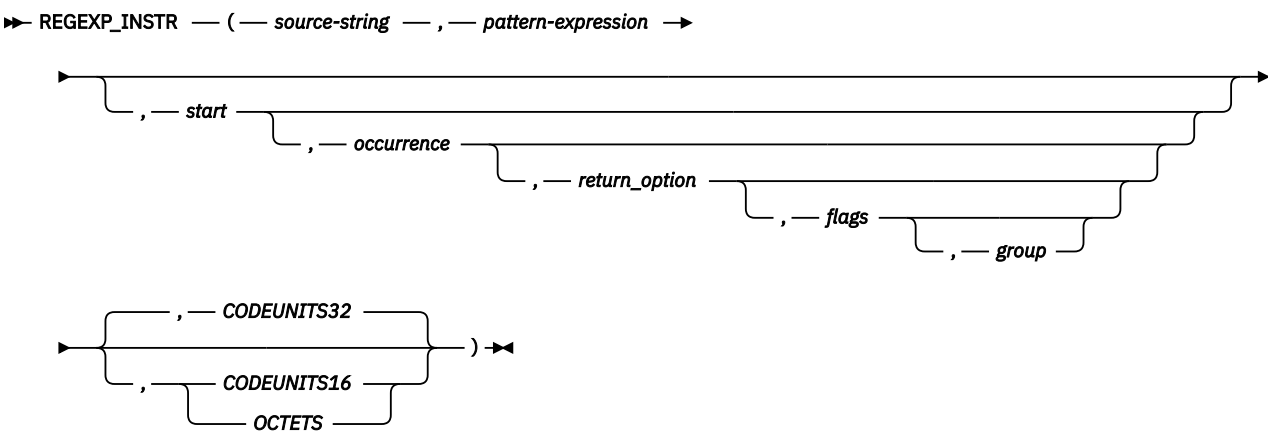
The REGEXP_EXTRACT scalar function is a synonym for the REGEXP_SUBSTR scalar function.

Notes

- The regular expression processing is done by using the International Components for Unicode (ICU) regular expression interface.
- **Considerations for non-Unicode databases**:
  - A regular expression pattern supports only half-width control characters; use a character string data type for the pattern expression argument. A character string data type can be used for the pattern expression argument even when a graphic string data type is used for the source string argument.
  - The source string argument must be a graphic string data type if the pattern expression argument is a graphic string data type.
  - The source string and replacement string arguments must both be character string data types or both be graphic string data types.

REGEXP_INSTR

The REGEXP_INSTR scalar function returns the starting or ending position of the matched substring, depending on the value of the **`return_option`** argument.



The schema is SYSIBM.

source-string
An expression that specifies the string in which the search is to take place. This expression must return a built-in character string, graphic string, numeric value, Boolean value, or datetime value. A numeric, Boolean, or datetime value is implicitly cast to VARCHAR before the function is evaluated. A character string cannot specify the FOR BIT DATA attribute (SQLSTATE 42815).

pattern-expression
An expression that specifies the regular expression string that is the pattern for the search. This expression must return a built-in character string, graphic string, numeric value, Boolean value, or datetime value. A numeric, Boolean, or datetime value is implicitly cast to VARCHAR before the function is evaluated. The length of a CLOB or DBCLOB expression must not be greater than the maximum length of a VARCHAR or VARGRAPHIC data type. A character string cannot specify the FOR BIT DATA attribute (SQLSTATE 42815).

start
An expression that specifies the position within *source-string* at which the search is to start. The expression must return a built-in character string, graphic string, Boolean, or numeric value. If the value is not of type INTEGER, it is implicitly cast to INTEGER before the function is evaluated. The value of the integer must be greater than or equal to 1. If OCTETS is specified and the source string is graphic data, the value of the integer must be odd (SQLSTATE 428GC). The default start value is 1. See parameter description for CODEUNITS16, CODEUNITS32, or OCTETS for the string unit that applies to the start position.

occurrence
An expression that specifies which occurrence of the pattern expression within the source string to search for. The expression must return a built-in character string, graphic string, Boolean, or numeric value. If the value is not of type INTEGER, it is implicitly cast to INTEGER before the function is evaluated. This value must be greater than or equal 1. The default occurrence value is 1, which indicates that only the first occurrence of the pattern expression is considered.

return-option
An expression that specifies what is returned relative to the occurrence. The expression must return a built-in character string, graphic string, Boolean, or numeric value. If the value is not of type INTEGER, it is implicitly cast to INTEGER before the function is evaluated. This value must be 0 or 1 (SQLSTATE 22546):

- A value of 0 returns the position of the first string unit of the occurrence.
- A value of 1 returns the position of the string unit that follows the occurrence.

The default return option value is 0.

flags
An expression that specifies flags that controls aspects of the pattern matching. The expression must return a built-in character string that does not specify the FOR BIT DATA attribute (SQLSTATE 42815). The string can include one or more valid flag values and the combination of flag values must be valid (SQLSTATE 2201T). An empty string is the same as the value 'c'. The default flag value is 'c'.

Table 86. Supported flag values

Flag value	Description
c	Specifies that matching is case-sensitive. This flag is the default value if 'c' or 'i' is not specified. This value must not be specified with a value of 'i'.
i	Specifies that matching is case insensitive. This value must not be specified with a value of 'c'.
m	Specifies that the input data can contain more than one line. By default, the '^' in a pattern matches only the start of the input string; the '$' in a pattern matches only the end of the input string. If this flag is set, "^" and "$" also matches at the start and end of each line within the input string.

Table 86. Supported flag values (continued)	
Flag value	**Description**
n	Specifies that the '.' character in a pattern matches a line terminator in the input string. By default, the '.' character in a pattern does not match a line terminator. A carriage-return and line-feed pair in the input string behaves as a single-line terminator, and matches a single "." in a pattern.
s	Specifies that the '.' character in a pattern matches a line terminator in the input string. This value is a synonym for the 'n' value.
x	Specifies that white space characters in a pattern are ignored, unless escaped.

group
An expression that specifies which capture group of the pattern expression is used to determine the position within source string to return. The expression must return a built-in character string, graphic string, Boolean, or numeric value. If the value is not of type INTEGER, it is implicitly cast to INTEGER before the function is evaluated. This value must be greater than or equal to 0 and must not be greater than the number of capture groups in the pattern expression (SQLSTATE 22546). The default group value is 0, which indicates that the position is based on the string that matches the entire pattern.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of both the start value and the result:

- CODEUNITS16 specifies that the start value and result are expressed in 16-bit UTF-16 code units.
- CODEUNITS32 specifies that the start value and result are expressed in 32-bit UTF-32 code units. This is the default.
- OCTETS specifies that the start value and result are expressed in bytes.

If the string unit is specified as CODEUNITS16 or OCTETS, and the string unit of the source string is CODEUNITS32, an error is returned (SQLSTATE 428GC).

For more information, see "String units in built-in functions" in “Character strings” on page 31.

Result

The result of the function is a large integer. If the pattern expression is found, the result is a number from 1 to n, where n is the actual length of the source string plus 1. The result value represents the position expressed in the string units used to process the function. If the pattern expression is not found and no argument is null, the result is 0.

If any argument of the REGEXP_INSTR function can be null, the result can be null. If any argument is null, the result is the null value.

Notes

- The regular expression processing is done by using the International Components for Unicode (ICU) regular expression interface.
- **Considerations for non-Unicode databases**:
  - A regular expression pattern supports only half-width control characters; use a character string data type for the pattern expression argument. A character string data type can be used for the pattern expression argument even when a graphic string data type is used for the source string argument.
  - The source string argument must be a graphic string data type if the pattern expression argument is a graphic string data type.

Examples

1. Find the first occurrence of a 'o' that has a character that is preceding it.

```
SELECT REGEXP_INSTR('hello to you', '.o',1,1)
   FROM sysibm.sysdummy1
```

The result is 4, which is the position of the second 'l' character.

2. Find the second occurrence of a 'o' that has a character that is preceding it.

```
SELECT REGEXP_INSTR('hello to you', '.o',1,2)
   FROM sysibm.sysdummy1
```

The result is 7, which is the position of the character 't'.

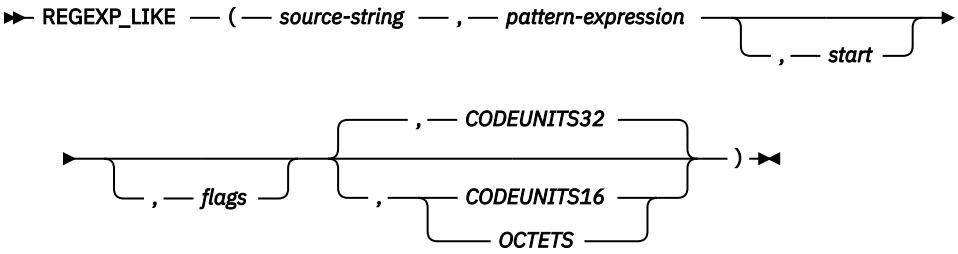
3. Find the position after the third occurrence of the first capture group of the regular expression '(.o).' using case insensitive matching.

```
SELECT REGEXP_INSTR('hello TO you', '(.o).', 1,3,1,'i',1)
   FROM sysibm.sysdummy1
```

The result is 12, which is the position of the character 'u' at the end of the string.

REGEXP_LIKE

The REGEXP_LIKE scalar function returns a boolean value indicating if the regular expression pattern is found in a string. The function can be used only where a predicate is supported.



The schema is SYSIBM.

source-string
An expression that specifies the string in which the search is to take place. This expression must return a built-in character string, graphic string, numeric value, Boolean value, or datetime value. A numeric, Boolean, or datetime value is implicitly cast to VARCHAR before the function is evaluated. A character string cannot specify the FOR BIT DATA attribute (SQLSTATE 42815).

pattern-expression
An expression that specifies the regular expression string that is the pattern for the search. This expression must return a built-in character string, graphic string, numeric value, Boolean value, or datetime value. A numeric, Boolean, or datetime value is implicitly cast to VARCHAR before the function is evaluated. The length of a CLOB or DBCLOB expression must not be greater than the maximum length of a VARCHAR or VARGRAPHIC data type. A character string cannot specify the FOR BIT DATA attribute (SQLSTATE 42815).

start
An expression that specifies the position within *source-string* at which the search is to start. The expression must return a built-in character string, graphic string, Boolean, or numeric value. If the value is not of type INTEGER, it is implicitly cast to INTEGER before the function is evaluated. The value of the integer must be greater than or equal to 1. If OCTETS is specified and the source string is graphic data, the value of the integer must be odd (SQLSTATE 428GC). The default start value is 1. See parameter description for CODEUNITS16, CODEUNITS32, or OCTETS for the string unit that applies to the start position.

flags
An expression that specifies flags that controls aspects of the pattern matching. The expression must return a built-in character string that does not specify the FOR BIT DATA attribute (SQLSTATE 42815).

The string can include one or more valid flag values and the combination of flag values must be valid (SQLSTATE 2201T). An empty string is the same as the value 'c'. The default flag value is 'c'.

Table 87. Supported flag values

Flag value	Description
c	Specifies that matching is case-sensitive. This flag is the default value if 'c' or 'i' is not specified. This value must not be specified with a value of 'i'.
i	Specifies that matching is case insensitive. This value must not be specified with a value of 'c'.
m	Specifies that the input data can contain more than one line. By default, the '^' in a pattern matches only the start of the input string; the '$' in a pattern matches only the end of the input string. If this flag is set, "^" and "$" also matches at the start and end of each line within the input string.
n	Specifies that the '.' character in a pattern matches a line terminator in the input string. By default, the '.' character in a pattern does not match a line terminator. A carriage-return and line-feed pair in the input string behaves as a single-line terminator, and matches a single "." in a pattern.
s	Specifies that the '.' character in a pattern matches a line terminator in the input string. This value is a synonym for the 'n' value.
x	Specifies that white space characters in a pattern are ignored, unless escaped.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of the start value:

- CODEUNITS16 specifies that the start value is expressed in 16-bit UTF-16 code units.
- CODEUNITS32 specifies that the start value is expressed in 32-bit UTF-32 code units. This is the default.
- OCTETS specifies that the start value is expressed in bytes.

If the string unit is specified as CODEUNITS16 or OCTETS, and if the string unit of the source string is CODEUNITS32, an error is returned (SQLSTATE 428GC).

For more information, see "String units in built-in functions" in “Character strings” on page 31.

Result

The result of the function is a BOOLEAN value. If the pattern expression is found, the result is true. If the pattern expression is not found, the result is false. If the value of any of the arguments is null, the result is unknown.

Notes

- The regular expression processing is done by using the International Components for Unicode (ICU) regular expression interface.
- **Considerations for non-Unicode databases**:
  - A regular expression pattern supports only half-width control characters; use a character string data type for the pattern expression argument. A character string data type can be used for the pattern expression argument even when a graphic string data type is used for the source string argument.
  - The source string argument must be a graphic string data type if the pattern expression argument is a graphic string data type.
  - The source string and replacement string arguments must both be character string data types or both be graphic string data types.

Examples

1. Select the employee number where the last name is spelled LUCCHESSI, LUCHESSI, or LUCHESI from the EMPLOYEE table without considering upper or lower case letters.

```
SELECT EMPNO FROM EMPLOYEE
   WHERE REGEXP_LIKE(LASTNAME,'luc+?hes+?i','i')
```

   The result is 1 row with EMPNO value '000110'.

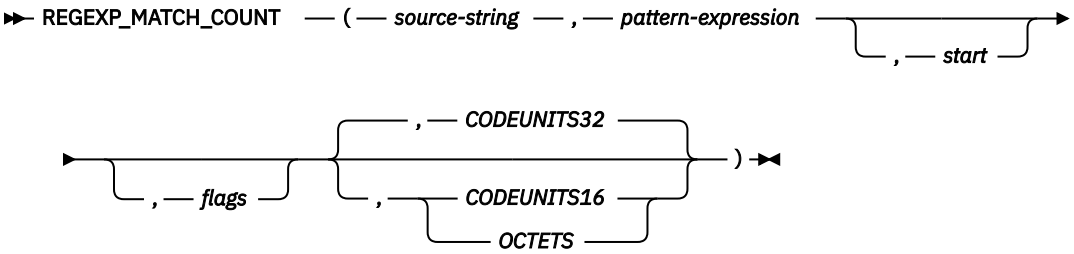
2. Select any invalid product identifier values from the PRODUCT table. The expected format is 'nnn-nnn-nn' where 'n' is a digit 0 - 9.

```
SELECT PID FROM PRODUCT
   WHERE NOT REGEXP_LIKE(pid,'[0-9]{3}-[0-9]{3}-[0-9]{2}')
```

   The result is 0 rows because all the product identifiers match the pattern.

REGEXP_MATCH_COUNT

The REGEXP_MATCH_COUNT scalar function returns a count of the number of times that a regular expression pattern is matched in a string.



The schema is SYSIBM.

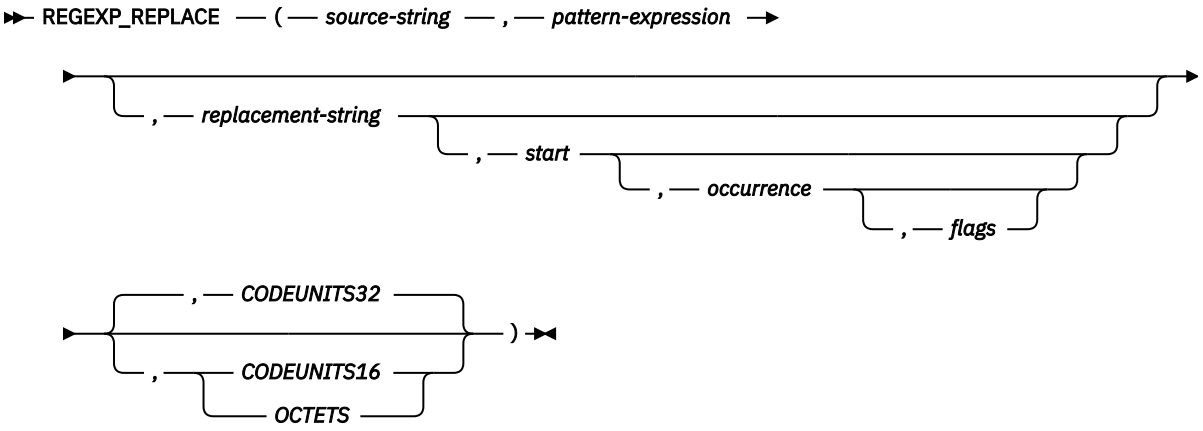
The REGEXP_MATCH_COUNT scalar function is a synonym for the REGEXP_COUNT scalar function.

Notes

- The regular expression processing is done by using the International Components for Unicode (ICU) regular expression interface.
- **Considerations for non-Unicode databases**:
  - A regular expression pattern supports only half-width control characters; use a character string data type for the pattern expression argument. A character string data type can be used for the pattern expression argument even when a graphic string data type is used for the source string argument.
  - The source string argument must be a graphic string data type if the pattern expression argument is a graphic string data type.
  - The source string and replacement string arguments must both be character string data types or both be graphic string data types.

REGEXP_REPLACE

The REGEXP_REPLACE scalar function returns a modified version of the source string where occurrences of the regular expression pattern found in the source string are replaced with the specified replacement string.



The schema is SYSIBM.

source-string
An expression that specifies the string in which the search is to take place. This expression must return a built-in character string, graphic string, numeric value, Boolean value, or datetime value. A numeric, Boolean, or datetime value is implicitly cast to VARCHAR before the function is evaluated. A character string cannot specify the FOR BIT DATA attribute (SQLSTATE 42815).

pattern-expression
An expression that specifies the regular expression string that is the pattern for the search. This expression must return a built-in character string, graphic string, numeric value, Boolean value, or datetime value. A numeric, Boolean, or datetime value is implicitly cast to VARCHAR before the function is evaluated. The length of a CLOB or DBCLOB expression must not be greater than the maximum length of a VARCHAR or VARGRAPHIC data type. A character string cannot specify the FOR BIT DATA attribute (SQLSTATE 42815).

replacement-string
An expression that specifies the replacement string for matching substrings. The expression must return a value that is a built-in character string, graphic string, numeric value, Boolean value, or datetime value. A numeric, Boolean, or datetime value is implicitly cast to VARCHAR before the function is evaluated. A character string cannot specify the FOR BIT DATA attribute (SQLSTATE 42815). The default replacement string is the empty string.

The content of the replacement string can include references to capture group text from the search to use in the replacement text. These references are of the form '$n' or '\n', where n is the number of the capture group and 0 represents the entire string that matches the pattern. The value for n must be in the range 0-9 and not greater than the number of capture groups in the pattern (SQLSTATE 2201V). For example, either '$2' or '\2' can be used to refer to the content found in the source string for the second capture group that is specified in the pattern expression. If the pattern expression must include a literal reference to a '$' or '\' character, that character must be preceded with an '/' character as an escape character ('\$' or '\\').

start
An expression that specifies the position within *source-string* at which the search is to start. The expression must return a built-in character string, graphic string, Boolean, or numeric value. If the value is not of type INTEGER, it is implicitly cast to INTEGER before the function is evaluated. The value of the integer must be greater than or equal to 1. If OCTETS is specified and the source string is graphic data, the value of the integer must be odd (SQLSTATE 428GC). The default start value is 1. See parameter description for CODEUNITS16, CODEUNITS32, or OCTETS for the string unit that applies to the start position.

occurrence
An expression that specifies which occurrence of the pattern expression within the source string is to be searched for and replaced. The expression must return a built-in character string, graphic string, Boolean, or numeric value. If the value is not of type INTEGER, it is implicitly cast to INTEGER before the function is evaluated. The occurrence value must be greater than or equal to 0. The default occurrence value is 0, which indicates that all occurrences of the pattern expression in the source string are replaced.

flags
An expression that specifies flags that controls aspects of the pattern matching. The expression must return a built-in character string that does not specify the FOR BIT DATA attribute (SQLSTATE 42815). The string can include one or more valid flag values and the combination of flag values must be valid (SQLSTATE 2201T). An empty string is the same as the value 'c'. The default flag value is 'c'.

Table 88. Supported flag values

Flag value	Description
c	Specifies that matching is case-sensitive. This flag is the default value if 'c' or 'i' is not specified. This value must not be specified with a value of 'i'.
i	Specifies that matching is case insensitive. This value must not be specified with a value of 'c'.
m	Specifies that the input data can contain more than one line. By default, the '^' in a pattern matches only the start of the input string; the '$' in a pattern matches only the end of the input string. If this flag is set, "^" and "$" also matches at the start and end of each line within the input string.
n	Specifies that the '.' character in a pattern matches a line terminator in the input string. By default, the '.' character in a pattern does not match a line terminator. A carriage-return and line-feed pair in the input string behaves as a single-line terminator, and matches a single "." in a pattern.
s	Specifies that the '.' character in a pattern matches a line terminator in the input string. This value is a synonym for the 'n' value.
x	Specifies that white space characters in a pattern are ignored, unless escaped.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of the start value:

- CODEUNITS16 specifies that the start value is expressed in 16-bit UTF-16 code units.
- CODEUNITS32 specifies that the start value is expressed in 32-bit UTF-32 code units. This is the default.
- OCTETS specifies that the start value is expressed in bytes.

If the string unit is specified as CODEUNITS16 or OCTETS, and if the string unit of the source string is CODEUNITS32, an error is returned (SQLSTATE 428GC).

For more information, see "String units in built-in functions" in “Character strings” on page 31.

Result

The result of the function is a string. If there are no occurrences of the pattern to be replaced and no argument is null, the original string is returned. The data type of the string is the same data type as the source string, except for CHAR, which becomes VARCHAR; and VARGRAPHIC, which becomes GRAPHIC.

The length attribute of the result data type is determined based on the length attributes of the source string and the replacement string by using the following calculation:

```
MIN(MaxTypeLen, LAS+(LAS+1)*LAR)
```

where *MaxTypeLen* represents the maximum length attribute for the data type of the result, *LAS* represents the length attribute for the data type of the source-string, and *LAR* represents the length attribute for the data type of the replacement string. If the replacement string is not specified, the value for *LAR* is 0. If the actual length of the result string exceeds the maximum for the return data type, an error is returned (SQLSTATE 54006).

If any argument of the REGEXP_REPLACE function can be null, the result can be null. If any argument is null, the result is the null value.

Notes

- The regular expression processing is done by using the International Components for Unicode (ICU) regular expression interface.
- **Considerations for non-Unicode databases**:
  - A regular expression pattern supports only half-width control characters; use a character string data type for the pattern expression argument. A character string data type can be used for the pattern expression argument even when a graphic string data type is used for the source string argument.
  - The source string argument must be a graphic string data type if the pattern expression argument is a graphic string data type.
  - The source string and replacement string arguments must both be character string data types or both be graphic string data types.

Example

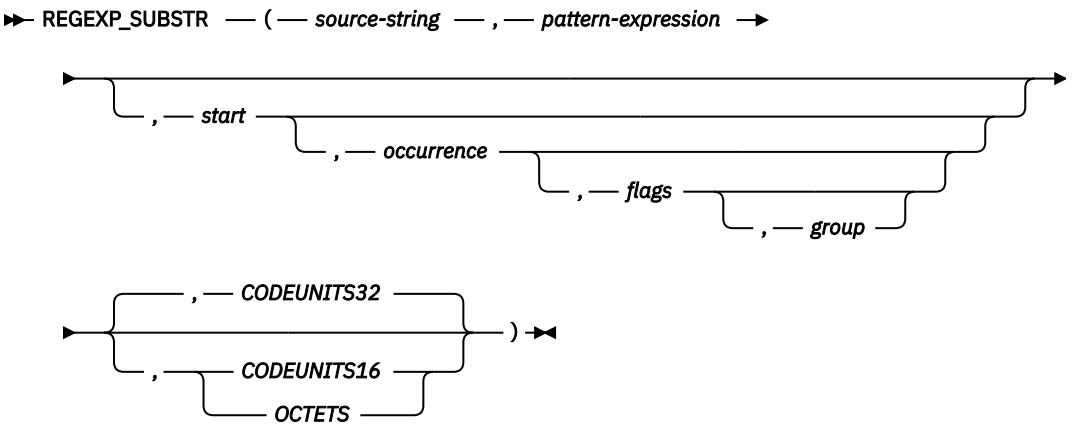
Replace the second occurrence of the pattern 'R.d' with 'Orange' using a case sensitive search.

```
SELECT REGEXP_REPLACE(
   'Red Yellow RED Blue Red Green Blue', 'R.d','Orange',1,2,'c')
      FROM sysibm.sysdummy1
```

The result is 'Red Yellow RED Blue Orange Green Blue'.

REGEXP_SUBSTR

The REGEXP_SUBSTR scalar function returns one occurrence of a substring of a string that matches the regular expression pattern.



The schema is SYSIBM.

source-string
An expression that specifies the string in which the search is to take place. This expression must return a built-in character string, graphic string, numeric value, Boolean value, or datetime value. A numeric, Boolean, or datetime value is implicitly cast to VARCHAR before the function is evaluated. A character string cannot specify the FOR BIT DATA attribute (SQLSTATE 42815).

pattern-expression
An expression that specifies the regular expression string that is the pattern for the search. This expression must return a built-in character string, graphic string, numeric value, Boolean value, or datetime value. A numeric, Boolean, or datetime value is implicitly cast to VARCHAR before the function is evaluated. The length of a CLOB or DBCLOB expression must not be greater than the maximum length of a VARCHAR or VARGRAPHIC data type. A character string cannot specify the FOR BIT DATA attribute (SQLSTATE 42815).

start
An expression that specifies the position within *source-string* at which the search is to start. The expression must return a built-in character string, graphic string, Boolean, or numeric value. If the value is not of type INTEGER, it is implicitly cast to INTEGER before the function is evaluated. The value of the integer must be greater than or equal to 1. If OCTETS is specified and the source string is graphic data, the value of the integer must be odd (SQLSTATE 428GC). The default start value is 1. See parameter description for CODEUNITS16, CODEUNITS32, or OCTETS for the string unit that applies to the start position.

occurrence
An expression that specifies which occurrence of the pattern expression within *source-string* to search for. The expression must return a built-in character string, graphic string, Boolean, or numeric value. If the value is not of type INTEGER, it is implicitly cast to INTEGER before the function is evaluated. The occurrence value must be greater than or equal 1. The default occurrence value is 1, which indicates that only the first occurrence of the pattern expression is considered.

flags
An expression that specifies flags that controls aspects of the pattern matching. The expression must return a built-in character string that does not specify the FOR BIT DATA attribute (SQLSTATE 42815). The string can include one or more valid flag values and the combination of flag values must be valid (SQLSTATE 2201T). An empty string is the same as the value 'c'. The default flag value is 'c'.

Table 89. Supported flag values

Flag value	Description
c	Specifies that matching is case-sensitive. This flag is the default value if 'c' or 'i' is not specified. This value must not be specified with a value of 'i'.
i	Specifies that matching is case insensitive. This value must not be specified with a value of 'c'.
m	Specifies that the input data can contain more than one line. By default, the '^' in a pattern matches only the start of the input string; the '$' in a pattern matches only the end of the input string. If this flag is set, "^" and "$" also matches at the start and end of each line within the input string.
n	Specifies that the '.' character in a pattern matches a line terminator in the input string. By default, the '.' character in a pattern does not match a line terminator. A carriage-return and line-feed pair in the input string behaves as a single-line terminator, and matches a single "." in a pattern.
s	Specifies that the '.' character in a pattern matches a line terminator in the input string. This value is a synonym for the 'n' value.
x	Specifies that white space characters in a pattern are ignored, unless escaped.

group
An expression that specifies which capture group of the pattern expression within source string to return. The expression must return a built-in character, binary, or graphic string, or a Boolean value. If the value is not of type INTEGER, it is implicitly cast to INTEGER before the function is evaluated. The group value must be greater than or equal to 0 and must not be greater than the number of capture groups in the pattern expression (SQLSTATE 22546). The default group value is 0, which indicates that the string that matches the entire pattern is to be returned.

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of the start value:

- CODEUNITS16 specifies that the start value is expressed in 16-bit UTF-16 code units.
- CODEUNITS32 specifies that the start value is expressed in 32-bit UTF-32 code units. This is the default.
- OCTETS specifies that the start value is expressed in bytes.

If the string unit is specified as CODEUNITS16 or OCTETS, and if the string unit of the source string is CODEUNITS32, an error is returned (SQLSTATE 428GC).

For more information, see "String units in built-in functions" in “Character strings” on page 31.

Result

The result of the function is a string. The data type of the string is the same data type as the source string, except for CHAR, which becomes VARCHAR; and GRAPHIC, which becomes and VARGRAPHIC. The length attribute of the result data type is same as the length attribute of the source string. The actual length of the result is the length of the occurrence in the string that matches the pattern expression. If the pattern expression is not found, the result is the null value.

The result of the REGEXP_SUBSTR function can be null. If any argument is null, the result is the null value.

Notes

- The regular expression processing is done by using the International Components for Unicode (ICU) regular expression interface.
- **Considerations for non-Unicode databases**:
  - A regular expression pattern supports only half-width control characters; use a character string data type for the pattern expression argument. A character string data type can be used for the pattern expression argument even when a graphic string data type is used for the source string argument.
  - The source string argument must be a graphic string data type if the pattern expression argument is a graphic string data type.

Examples

1. Return the string which matches any character preceding a 'o'.

```
SELECT REGEXP_SUBSTR('hello to you', '.o',1,1)
   FROM sysibm.sysdummy1
```

   The result is 'lo'.
2. Return the second string occurrence which matches any character preceding a 'o'.

```
SELECT REGEXP_SUBSTR('hello to you', '.o',1,2)
   FROM sysibm.sysdummy1
```

   The result is 'to'.
3. Return the third string occurrence which matches any character preceding a 'o'.

```
SELECT REGEXP_SUBSTR('hello to you', '.o',1,3)
   FROM sysibm.sysdummy1
```

   The result is 'yo'.

REPEAT

The REPEAT function returns a character string that is composed of the first argument repeated the number of times that are specified by the second argument.

The schema is SYSIBM. The SYSFUN version of the REPEAT function continues to be available.

expression1
An expression that specifies the string to be repeated. The expression must return a built-in character string, numeric value, Boolean value, or datetime value. If the value is not a character string, it is implicitly cast to VARCHAR before the function is evaluated.

expression2
An expression that is a positive integer value or zero that specifies the number of times to repeat the string. The expression must return a built-in character string, graphic string, or numeric value. If the value is not an integer, it is implicitly cast to INTEGER before the function is evaluated.

Result

The result of the function is one of the following data types:

- VARBINARY if *expression1* is a BINARY or VARBINARY string
- VARCHAR if *expression1* is a CHAR or VARCHAR string
- VARGRAPHIC if *expression1* is GRAPHIC or VARGRAPHIC string
- CLOB if *expression1* is CLOB
- BLOB if *expression1* is BLOB
- DBCLOB if *expression1* is a DBCLOB

If *expression2* is a constant, the length attribute of the result is minimum of the length attribute of *expression1* times *expression2* and the maximum length of the result data type. Otherwise, the length attribute depends on the data type of the result:

- 4000 for VARBINARY and VARCHAR
- 2000 for VARGRAPHIC
- 1 MB for CLOB, DBCLOB, and BLOB

The actual length of the result is the actual length of *expression1* times *expression2*. If the actual length of the result string exceeds the length attribute for the return type, an error is returned (SQLSTATE 54006).

If the result data type is a character string or graphic string, the string units of the result are the string units of *expression1*.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

1. Repeat 'abc' two times to create 'abcabc'.

```
SELECT REPEAT('abc',2)
  FROM SYSIBM.SYSDUMMY1
```

2. List the phrase 'REPEAT THIS' five times. Use the CHAR function to limit the output to 60 bytes.

```
SELECT CHAR(REPEAT('REPEAT THIS',5), 60)
  FROM SYSIBM.SYSDUMMY1
```

This example outputs the following string:

```
'REPEAT THISREPEAT THISREPEAT THISREPEAT THISREPEAT THIS     '
```

3. For the following query, the LENGTH function returns a value of 0 because the result of repeating a string zero times is an empty string, which is a zero-length string.

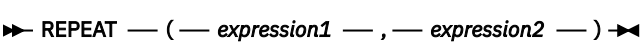
```
SELECT LENGTH(REPEAT('REPEAT THIS',0))
  FROM SYSIBM.SYSDUMMY1
```

4. For the following query, the LENGTH function returns a value of 0. A value of 0 because the result of repeating an empty string any number of times is an empty string, which is a zero-length string.

```
SELECT LENGTH(REPEAT('', 5))
  FROM SYSIBM.SYSDUMMY1
```

REPEAT (SYSFUN schema)

Returns a character string composed of the first argument repeated the number of times specified by the second argument.



The schema is SYSFUN.

expression1
An expression that returns a value of built-in character string, graphic string, or binary string data type. In a Unicode database, the expression can also return a graphic string, in which case it is first converted to a character string before the function is evaluated. The maximum length is:

- 4000 bytes for a VARCHAR
- 1,048,576 bytes for a CLOB or binary string

expression2
An expression that returns a value of data type SMALLINT or INTEGER. The returned value specifies the number of times to repeat the string returned by *expression1*.

Result

The data type of the result is:

- VARCHAR(4000 OCTETS) if the first expression returns a CHAR or VARCHAR value
- CLOB(1M OCTETS) if the first expression returns a CLOB value
- BLOB(1M) if the first expression returns a BLOB value

The result can be null; if any argument is null, the result is the null value.

Examples

List the phrase 'REPEAT THIS' five times.

```
VALUES REPEAT('REPEAT THIS', 5)
```

This example returns the following VARCHAR(4000 OCTETS) output:

```
1
------------------------------------------------------------
REPEAT THISREPEAT THISREPEAT THISREPEAT THISREPEAT THIS
```

List the phrase 'REPEAT THIS' five times and limit the output to 60 bytes:

```
VALUES CHAR(REPEAT('REPEAT THIS', 5), 60)
```

This example returns the following CHAR(60) output:

```
1
-------------------------------------------------------------
REPEAT THISREPEAT THISREPEAT THISREPEAT THISREPEAT THIS
```

REPLACE

Replaces all occurrences of *search-string* in *source-string* with *replace-string*.

The schema is SYSIBM. The SYSFUN version of the REPLACE function continues to be available but it is not sensitive to the database collation.

If the search string is not found in the source string, the search string is returned unchanged. If the Unicode database is defined with a locale-sensitive UCA-based collation and none of the *source-string*, *search-string*, or *replace-string* arguments are defined as FOR BIT DATA or as a binary string, a linguistically correct search is done. Otherwise, the search is done using a binary comparison with no special consideration for multi-byte characters.

source-string
An expression that specifies the source string. The expression must return a value that is a built-in character string, numeric value, DBCLOB value, Boolean value, or datetime value. If the value is:

- A numeric, datetime, or CLOB value, it is implicitly cast to VARCHAR before the function is evaluated
- A DBCLOB value, it is implicitly cast to VARGRAPHIC before the function is evaluated

The actual length of binary strings cannot exceed 1,048,576 bytes (SQLSTATE 42815).

search-string
An expression that specifies the string to be removed from the source string. The expression must return a value that is a built-in character string, numeric value, DBCLOB value, Boolean value, or datetime value. If the value is:

- A numeric, datetime, or CLOB value, it is implicitly cast to VARCHAR before the function is evaluated
- A DBCLOB value, it is implicitly cast to VARGRAPHIC before the function is evaluated

The actual length of binary strings cannot exceed 1,048,576 bytes (SQLSTATE 42815).

replace-string
An expression that specifies the replacement string. The expression must return a value that is a built-in character string, numeric value, DBCLOB value, Boolean value, or datetime value. If the value is:

- A numeric, datetime, or CLOB value, it is implicitly cast to VARCHAR before the function is evaluated
- A DBCLOB value, it is implicitly cast to VARGRAPHIC before the function is evaluated

The actual length of binary strings cannot exceed 1,048,576 bytes (SQLSTATE 42815). If the expression is an empty string or is not specified, nothing replaces the string that is removed from the source string.

All three arguments must have compatible data types.

Result

The data type of the result depends on the data type of the arguments:

- If any argument is a BLOB, the result is a BLOB.
- If the arguments are binary strings and not BLOB, the result is a VARBINARY.
- If the arguments are character strings, the result is a VARCHAR. If any argument is defined as FOR BIT DATA, the result is defined as FOR BIT DATA.

- If the arguments are graphic strings, the result is a VARGRAPHIC.

In a Unicode database, if both character strings and graphic strings are used as arguments, then the result data type is based on the string type of the *source-string* argument. If the *source-string* argument is a character string type, then the result data type is VARCHAR. If the *source-string* argument is a graphic string type, then the result data type is VARGRAPHIC.

The string unit of the result is the string unit of *source-string*. If any argument is defined as FOR BIT DATA, the other arguments cannot be defined with string units of CODEUNITS32.

The string units of the result is the same as the string units of *source-string*. The length attribute of the result depends on the arguments:

- The length attribute of the result is the length attribute of the source-string (with the same string units) in the following cases:
  - The *replace-string* argument is not specified or is specified as an empty string constant.
  - The search-string is a constant and the number of bytes in the search-string constant is greater than or equal to:
    - The number of bytes of a constant *replace-string*.
    - The length attribute of a non-constant character string type *replace-string* in OCTETS.
    - The length attribute times 2 of a non-constant graphic string type *replace-string* in double bytes or CODEUNITS16.
    - The length attribute times 4 of a non-constant *replace-string* in CODEUNITS32.
    - The length attribute of a non-constant binary string type *replace-string*.
- Otherwise, the length attribute of the result is determined by the following calculation depending on the data type of the result (that allows for the smallest possible *search-string* to be replaced by the largest possible *replace-string*):
  - For VARCHAR with string units of OCTETS:
    - If L1 < = 4000, the length attribute of the result is `MIN(4000, (L3*(L1/L2)) + MOD(L1,L2))`
    - Otherwise, the length attribute of the result is `MIN(32672, (L3*(L1/L2)) + MOD(L1,L2))`
  - For VARCHAR with string units of CODEUNITS32, the length attribute of the result is `MIN(8168, (L3*(L1/L2)) + MOD(L1,L2))`.
  - For VARGRAPHIC with string units of double bytes or CODEUNITS16:
    - If L1 < = 2000, the length attribute of the result is `MIN(2000, (L3*(L1/L2)) + MOD(L1,L2))`
    - Otherwise, the length attribute of the result is `MIN(16336, (L3*(L1/L2)) + MOD(L1,L2))`
  - For VARGRAPHIC with string units of CODEUNITS32, the length attribute of the result is `MIN(8168, (L3*(L1/L2)) + MOD(L1,L2))`.
  - For VARBINARY, the length attribute of the result is `MIN(32672, (L3*(L1/L2)) + MOD(L1,L2))`
  - For BLOB, the length attribute of the result is `MIN(2G, (L3*(L1/L2)) + MOD(L1,L2))`

where the length values for L1, L2, and L3 are given in the following table based on the string units of the result:

Table 90. Length values of L1, L2, and L3

String units of the result	L1	L2	L3
OCTETS (a character string)	Length attribute of *source-string*	One (1) if not a constant, otherwise the actual number of bytes in the constant expressed as a character string type.	• If *replace-string* is a constant, the actual number of bytes in the constant expressed as a character string type. • If *replace-string* is a character string in OCTETS then the length attribute of *replace-string*. • If *replace-string* is a character string or a graphic string in CODEUNITS32, then 4 times the length attribute of *replace-string*. • If *replace-string* is a graphic string in double bytes or CODEUNITS16, then 3 times the length attribute of *replace-string*.
CODEUNITS16 or double bytes (a graphic string)	Length attribute of *source-string*	One (1) if not a constant, otherwise the actual number of double bytes or CODEUNITS16 string units in the constant expressed as a graphic string type.	• If *replace-string* is a constant, the actual number of double bytes or CODEUNITS16 string units in the constant expressed as a graphic string type. • If *replace-string* is a character string in OCTETS or a graphic string in double bytes or CODEUNITS16 then the length attribute of *replace-string*. • If *replace-string* is a character string or graphic string in CODEUNITS32 then the 2 times the length attribute of *replace-string*.

Table 90. Length values of L1, L2, and L3 (continued)			
String units of the result	**L1**	**L2**	**L3**
CODEUNITS32 (a character string or a graphic string)	Length attribute of *source-string*	One (1) if not a constant, otherwise length attribute of *search-string*.	Length attribute of *replace-string*
Not applicable (a binary string)	Length attribute of *source-string*	One (1) if not a constant, otherwise the actual number of bytes in the constant expressed as a binary string type.	• If *replace-string* is a constant, the actual number of bytes in the constant expressed as a binary string type. • If *replace-string* is a binary string or character string that is FOR BIT DATA, then the length attribute of *replace-string*.

If the result is a character string, the length attribute of the result must not exceed the maximum length of the VARCHAR data type in the string units of the result. If the result is a graphic string, the length attribute of the result must not exceed the maximum length of the VARGRAPHIC data type in the string units of the result.

The actual length of the result is the actual length of *source-string* plus the number of occurrences of *search-string* that exist in *source-string* multiplied by the actual length of *replace-string* minus the actual length of *search-string*. If the actual length of the result would exceed the length attribute of the result or would exceed 1,048,576 bytes, an error is returned (SQLSTATE 22001).

If the actual length of the *replace-string* exceeds the maximum for the return data type, an error is returned. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Replace all occurrences of the letter 'N' in the word 'DINING' with 'VID'.

```
VALUES CHAR (REPLACE ('DINING', 'N', 'VID'), 10)
```

  The result is the string 'DIVIDIVIDG'.

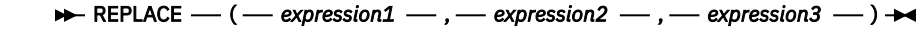
- *Example 2*: In a Unicode database with case-insensitive collation CLDR181_LEN_S1, replace the word 'QUICK' with the word 'LARGE'.

```
VALUES REPLACE ('The quick brown fox', 'QUICK', 'LARGE')
```

  The result is the string 'The LARGE brown fox'.

REPLACE (SYSFUN schema)

Replaces all occurrences of *expression2* in *expression1* with *expression3*.



The schema is SYSFUN.

The search is done using a binary comparison with no special consideration for multi-byte characters.

expression1* or *expression2* or *expression3
The data type for the arguments can be of any built-in character string or binary string type.

The expression must return a built-in character string, Boolean value, or binary string. In a Unicode database, the expression can also return a graphic string, in which case it is first converted to a character string before the function is evaluated. The maximum length is:

- 4000 bytes for a VARCHAR
- 1,048,576 bytes for a CLOB or binary string

A CHAR value is converted to VARCHAR and a LONG VARCHAR value is converted to CLOB(1M).

Result

The data type of the result is VARCHAR(4000).

The result can be null; if any argument is null, the result is the null value.

Examples

Replace all occurrences of the letter 'N' in the word 'DINING' with 'VID':

```
VALUES REPLACE ('DINING', 'N', 'VID')
```

This example returns the following VARCHAR(4000) output:

```
1
----------
DIVIDIVIDG
```

Replace all occurrences of the letter 'N' in the word 'DINING' with 'VID', and limit the output to 9 bytes:

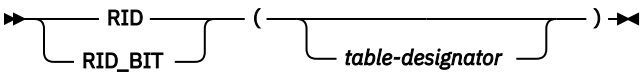
```
VALUES CHAR (REPLACE ('DINING', 'N', 'VID'), 9)
```

This example returns the following CHAR(9) output:

```
1
----------
DIVIDIVID
```

RID and RID_BIT

The RID and RID_BIT functions are used to uniquely identify a row. Each returns the row identifier (RID) of a row. The result of the RID_BIT function, unlike the result of the RID function, contains table information to help you avoid inadvertently using it with the wrong table. Both functions are non-deterministic.



The schema is SYSIBM. The function name cannot be specified as a qualified name.

RID
Indicates that the RID function is to be run. Starting from Db2 version 11.5.3, the RID function can be used in partitioned database environment. The value provided by the RID function does not include database partition information. In a partitioned database environment, use **DBPARTITIONNUM** to make the value unique.

RID_BIT
Indicates that the RID_BIT function is to be run.

table-designator
Uniquely identifies a base table, view, or nested table expression (SQLSTATE 42703). If the table designator specifies a view or nested table expression, the RID_BIT and RID functions return the RID of the base table of the view or nested table expression. The specified view or nested table expression must contain only one base table in its outer subselect (SQLSTATE 42703). The table designator must be deletable (SQLSTATE 42703). For information about deletable views, see the "Notes" section of "CREATE VIEW".

If a table designator is not specified, the FROM clause must contain only one element which can be derived to be the table designator (SQL STATE 42703).

Result

The data type of the result is either BIGINT (for RID) or VARCHAR (16) FOR BIT DATA (for RID_BIT). The result can be null.

The RID or RID_BIT function might return different values when invoked several times for a single row. For example, if RID or RID_BIT is run both before and after the reorg utility is run against the specified table, the function might return a different value each time.

Notes

- To implement optimistic locking in an application, use the values returned by the ROW CHANGE TOKEN expression as arguments to the RID_BIT scalar function.
- Starting from Db2 version 11.5.3, the RID scalar functions can be used with column-organized tables.
- ***Syntax alternatives***: The following alternatives are non-standard. They are supported for compatibility with earlier product versions or with other database products.
  - The pseudocolumn "ROWID" can be used to refer to the RID. An unqualified ROWID reference is equivalent to RID_BIT() and a qualified ROWID such as EMPLOYEE.ROWID is equivalent to RID_BIT(EMPLOYEE).

Examples

- *Example 1*: Return the RID and last name of employees in department 20 from the EMPLOYEE table.

```
SELECT RID_BIT (EMPLOYEE), ROW CHANGE TOKEN FOR EMPLOYEE, LASTNAME
  FROM EMPLOYEE
  WHERE DEPTNO = '20'
```

- *Example 2*: Given table EMP1, which is defined as follows:

```
CREATE TABLE EMP1 (
  EMPNO   CHAR(6),
  NAME    CHAR(30),
  SALARY  DECIMAL(9,2),
  PICTURE BLOB(250K),
  RESUME  CLOB(32K)
)
```

Set host variable HV_EMP_RID to the value of the RID_BIT built-in scalar function, and HV_EMP_RCT to the value of the ROW CHANGE TOKEN expression for the row corresponding to employee number 3500.

```
SELECT RID_BIT(EMP1), ROW CHANGE TOKEN FOR EMP1
  INTO :HV_EMP_RID, :HV_EMP_RCT FROM EMP1
  WHERE EMPNO = '3500'
```

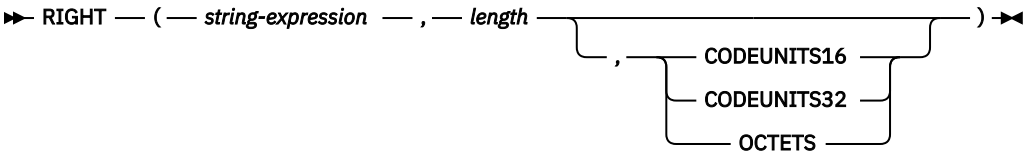
Using that RID value to identify the employee, and user-defined function UPDATE_RESUME, increase the employee's salary by $1000 and update the employee's resume.

```
UPDATE EMP1 SET
  SALARY = SALARY + 1000,
  RESUME = UPDATE_RESUME(:HV_RESUME)
```

```
WHERE RID_BIT(EMP1) = :HV_EMP_RID
AND ROW CHANGE TOKEN FOR EMP1 = :HV_EMP_RCT
```

RIGHT

The RIGHT function returns the rightmost string of *string-expression* of length *length*, expressed in the specified string unit.



The schema is SYSIBM. The SYSFUN version of the RIGHT function continues to be available.

If *string-expression* is a character string, the result is a character string. If *string-expression* is a graphic string, the result is a graphic string

string-expression
An expression that specifies the string from which the result is derived. The expression must return a value that is a built-in string, numeric, Boolean, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. A substring of *string-expression* is zero or more contiguous code points of *string-expression*.

length

An expression that specifies the length of the result. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. *length* must be greater than or equal to 0 (SQLSTATE 22011). If OCTETS is specified and the result is graphic data, the value must be an even number (SQLSTATE 428GC).

If *length* is not a constant and a string unit is not specified, then *length* must be less than or equal to the length attribute of *string-expression* (SQLSTATE 22011).

If *length* is not a constant and a string unit is specified, then *length* must be less than or equal to the value from table 1 (SQLSTATE 22011):

Table 91. Maximum value of length when a string unit is specified

String unit of *string-expression*	**Specified string unit**	**Maximum value of *length*** **L = length attribute of *string-expression***
String unit of *string-expression*	Specified string unit	Maximum value of *length* L = length attribute of *string-expression*
OCTETS	OCTETS	L
OCTETS	CODEUNITS16	L/2
OCTETS	CODEUNITS32	L/4
CODEUNITS16	OCTETS	L*2
CODEUNITS16	CODEUNITS16	L
CODEUNITS16	CODEUNITS32	L/2
CODEUNITS32	OCTETS	L*4
CODEUNITS32	CODEUNITS16	L*2

Table 91. Maximum value of length when a string unit is specified (continued)		
String unit of *string-expression*	**Specified string unit**	**Maximum value of *length*** **L = length attribute of *string-expression***
CODEUNITS32	CODEUNITS32	L

If *length* is a constant:

- if *string-expression*is CHAR, VARCHAR, GRAPHIC or VARGRAPHIC, *length* must be less than or equal to 32 672 OCTETS, 16 336 CODEUNITS16 or 8 168 CODEUNITS32 (SQLSTATE 22011)
- if *string-expression* is CLOB or DBCLOB, *length* must be less than or equal to 2147483647 OCTETS, 1 073 741 823 CODEUNITS16 or 536870911 CODEUNITS32 (SQLSTATE 22011)
- if *string-expression* is BLOB, *length* must be less than or equal to 2147483647 OCTETS (SQLSTATE 22011)

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of *length*.

CODEUNITS16 specifies that *length* is expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *length* is expressed in 32-bit UTF-32 code units. OCTETS specifies that *length* is expressed in bytes.

If the string unit is specified as CODEUNITS16 or CODEUNITS32, and *string-expression* is a binary string or bit data, an error is returned (SQLSTATE 428GC). If the string unit is specified as OCTETS and *string-expression* is a graphic string, *length* must be an even number; otherwise, an error is returned (SQLSTATE 428GC). If a string unit is not explicitly specified, the string unit of *string-expression* determines the unit that is used. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

The *string-expression* is padded on the right with the necessary number of padding characters so that the specified substring of *string-expression* always exists. The character used for padding is the same character that is used to pad the string in contexts where padding would occur. For more information about padding, see "String assignments" in "Assignments and comparisons".

The result of the function is a varying-length string that depends on the data type of *string-expression*:

- VARCHAR if *string-expression* is CHAR or VARCHAR
- CLOB if *string-expression* is CLOB
- VARGRAPHIC if *string-expression* is GRAPHIC or VARGRAPHIC
- DBCLOB if *string-expression* is DBCLOB
- VARBINARY if *string-expression* is BINARY or VARBINARY
- BLOB if *string-expression* is BLOB

The string unit of the result is the string unit of *string-expression*.The length attribute of the result depends on how *length* and string unit are specified:

- If *length* is not a constant, then the length attribute of the result is the same as the length attribute of *string-expression*.
- If *length* is a constant and a string unit is not specified, then the length attribute of the result is the maximum of length and the length attribute of *string-expression*.
- If *length* is a constant and a string unit is specified, then the length attribute of the result is shown in Table 2:

Table 92. Length attribute of result when length is a constant and a string unit is specified		
String unit of *string-expression*	**Specified string unit**	**Maximum value of *length*** **L = length attribute of *string-expression***
OCTETS	OCTETS	max(L, *length*)
OCTETS	CODEUNITS16	max(L, *length* * 2)
OCTETS	CODEUNITS32	max(L, *length* * 4)
CODEUNITS16	OCTETS	max(L, *length* / 2)
CODEUNITS16	CODEUNITS16	max(L, *length*)
CODEUNITS16	CODEUNITS32	max(L, *length* * 2)
CODEUNITS32	OCTETS	max(L, *length* / 4)
CODEUNITS32	CODEUNITS16	max(L, *length* / 2)
CODEUNITS32	CODEUNITS32	max(L, *length*)

The actual length of the result (in string units) is *length*.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Assume that variable ALPHA has a value of "ABCDEF". The following statement:

```
SELECT RIGHT(ALPHA,3)
  FROM SYSIBM.SYSDUMMY1
```

  returns "DEF", which are the three rightmost characters in ALPHA.

- *Example 2*: Assume that variable NAME, which is defined as VARCHAR(50), has a value of "KATIE AUSTIN", and that the integer variable LASTNAME_LEN has a value of 6. The following statement:

```
SELECT RIGHT(NAME,LASTNAME_LEN)
  FROM SYSIBM.SYSDUMMY1
```

  returns the value "AUSTIN".

- *Example 3*: The following statement returns a zero-length string.

```
SELECT RIGHT('ABCABC',0)
  FROM SYSIBM.SYSDUMMY1
```

- *Example 4*: The FIRSTNME column in the EMPLOYEE table is defined as VARCHAR(12). Find the first name of an employee whose last name is "BROWN" and return the first name in a 10-byte string.

```
SELECT RIGHT(FIRSTNME, 10)
  FROM EMPLOYEE
  WHERE LASTNAME = 'BROWN'
```

  returns a VARCHAR(12) string that has the value "DAVID" followed by five blank characters.

- *Example 5*: In a Unicode database, FIRSTNAME is a VARCHAR(12) column. One of its values is the 6-character string "Jürgen". When FIRSTNAME has this value:

```
Function...                      Returns...

RIGHT(FIRSTNAME,5,CODEUNITS32)  'ürgen' -- x'C3BC7267656E'
RIGHT(FIRSTNAME,5,CODEUNITS16)  'ürgen' -- x'C3BC7267656E'
RIGHT(FIRSTNAME,5,OCTETS)       'rgen'  -- x'207267656E', a truncated string
```

- *Example 6*: The following example works with the Unicode string "&N~AB," where "&" is the musical symbol G clef character, and "~" is the combining tilde character. This string is shown in different Unicode encoding forms in the following example:

	"&"	**"N"**	**"~"**	**"A"**	**"B"**
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'

Assume that the variable UTF8_VAR, with a length attribute of 20 bytes, contains the UTF-8 representation of the string.

```
SELECT RIGHT(UTF8_VAR, 2, CODEUNITS16),
  RIGHT(UTF8_VAR, 2, CODEUNITS32),
  RIGHT(UTF8_VAR, 2, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values "AB", "AB", and "AB"', respectively.

```
SELECT RIGHT(UTF8_VAR, 5, CODEUNITS16),
  RIGHT(UTF8_VAR, 5, CODEUNITS32),
  RIGHT(UTF8_VAR, 5, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values "?N~AB", "&N~AB", and "N~AB", respectively, where ?'is X'EDB49E'.

```
SELECT RIGHT(UTF8_VAR, 10, CODEUNITS16),
  RIGHT(UTF8_VAR, 10, CODEUNITS32),
  RIGHT(UTF8_VAR, 10, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values "&N~ABbbbb", "&N~ABbbbbb", and "&N~ABb", respectively, where "b" represents the blank character.

Assume that the variable UTF16_VAR, with a length attribute of 20 code units, contains the UTF-16BE representation of the string.

```
SELECT RIGHT(UTF16_VAR, 2, CODEUNITS16),
  RIGHT(UTF16_VAR, 2, CODEUNITS32),
  RIGHT(UTF16_VAR, 2, OCTETS))
FROM SYSIBM.SYSDUMMY1
```

returns the values "AB", "AB", and "B", respectively.

```
SELECT RIGHT(UTF16_VAR, 5, CODEUNITS16),
  RIGHT(UTF16_VAR, 5, CODEUNITS32),
  RIGHT(UTF16_VAR, 6, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values "?N~AB", "&N~AB", and "~AB", respectively, where ? is the stand-alone low surrogate X'DD1E'.

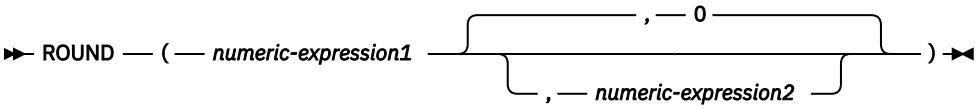
```
SELECT RIGHT(UTF16_VAR, 10, CODEUNITS16),
  RIGHT(UTF16_VAR, 10, CODEUNITS32),
  RIGHT(UTF16_VAR, 10, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values "&N~ABbbbb", "&N~ABbbbbb", and "?N~AB", respectively, where "b" represents the blank character and ? is X'DD1E'.

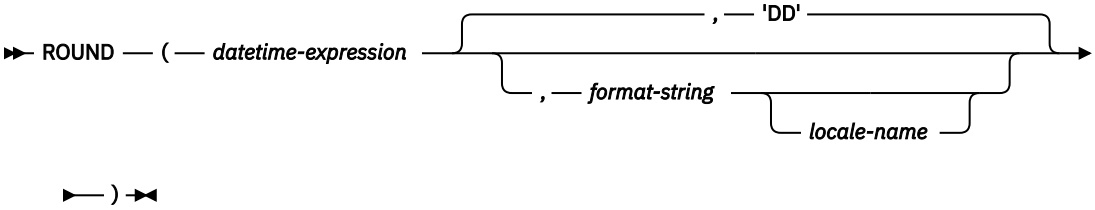
ROUND

The ROUND function returns a rounded value of a number or a datetime value.

ROUND numeric



ROUND datetime



The schema is SYSIBM. The SYSFUN version of the ROUND numeric function continues to be available.

The return value depends on the first argument:

- If the result of the first argument is a numeric value, the ROUND functions returns a number, rounded to the specified number of places to the right or left of the decimal point.
- If the first argument is a DATE, TIME, or TIMESTAMP the ROUND functions returns a datetime value, rounded to the unit specified by *format-string*.

ROUND numeric

If *numeric-expression1* is positive, a digit value of 5 or greater is an indication to round to the next higher positive number. For example, ROUND(3.5,0) = 4. If *numeric-expression1* is negative, a digit value of 5 or greater is an indication to round to the next lower negative number. For example, ROUND(-3.5,0) = -4.

numeric-expression1
An expression that must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, or numeric data type. If the value is not a numeric data type, it is implicitly cast to DECFLOAT(34) before evaluating the function.

If the expression is a decimal floating-point data type, the DECFLOAT rounding mode will not be used. The rounding behavior of ROUND corresponds to a value of ROUND_HALF_UP. If a different rounding behavior is wanted, use the QUANTIZE function.

numeric-expression2
An expression that returns a value that is a built-in numeric data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

If *numeric-expression2* is not negative, *numeric-expression1* is rounded to the absolute value of *numeric-expression2* number of places to the right of the decimal point.

If *numeric-expression2* is negative, *numeric-expression1* is rounded to the absolute value of *numeric-expression2*+1 number of places to the left of the decimal point.

If the absolute value of a negative *numeric-expression2* is larger than the number of digits to the left of the decimal point, the result is 0. For example, ROUND(748.58,-4) = 0. If *numeric-expression1* is positive, a digit value of 5 is rounded to the next higher positive number. If *numeric-expression1* is negative, a digit value of 5 is rounded to the next lower negative number.

The data type and length attribute of the result are the same as the data type and length attribute of the first argument, except that the precision is increased by one if the *numeric-expression1*

is DECIMAL and the precision is less than 31. For example, an argument with a data type of DECIMAL(5,2) results in DECIMAL(6,2). An argument with a data type of DECIMAL(31,2) results in DECIMAL(31,2). The scale is the same as the scale of the first argument.

If either argument can be null or if the argument is not a decimal floating-point number and the database is configured with **dft_sqlmathwarn** set to YES, the result can be null. If either argument is null, the result is the null value.

This function is not affected by the setting of the CURRENT DECFLOAT ROUNDING MODE special register, even for decimal floating-point arguments. The rounding behavior of ROUND corresponds to a value of ROUND_HALF_UP. If you want behavior for a decimal floating-point value that conforms to the rounding mode specified by the CURRENT DECFLOAT ROUNDING MODE special register, use the QUANTIZE function instead.

ROUND datetime

If *datetime-expression* has a datetime data type, the ROUND function returns *datetime-expression* rounded to the unit specified by the *format-string*. If *format-string* is not specified, *datetime-expression* is rounded to the nearest day, as if 'DD' is specified for *format-string*.

datetime-expression

An expression that must return a value that is a date, a time, or a timestamp. String representations of these data types are not supported and must be explicitly cast to a DATE, TIME, or TIMESTAMP for use with this function; alternatively, you can use the ROUND_TIMESTAMP function for a string representation of a date or timestamp.

format-string

An expression that returns a built-in character string data type with an actual length that is not greater than 255 bytes. The format element in *format-string* specifies how *datetime-expression* should be rounded. For example, if *format-string* is 'DD', a timestamp that is represented by *datetime-expression* is rounded to the nearest day. Leading and trailing blanks are removed from the string, and the resulting substring must be a valid format element for the type of *datetime-expression* (SQLSTATE 22007). The default is 'DD', which cannot be used if the data type of *datetime-expression* is TIME.

Allowable values for *format-string* are listed in the table of format elements listed in the Table 1.

locale-name

A character constant that specifies the locale used to determine the first day of the week when using format element values DAY, DY, or D. The value of *locale-name* is not case sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming, see "Locale names for SQL and XQuery". If *locale-name* is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result of the function has the same DATE type as *datetime-expression*. The result can be null; if any argument is null, the result is the null value.

The following format elements are used to identify the rounding or truncation unit of the datetime value in the ROUND, ROUND_TIMESTAMP, TRUNCATE and TRUNC_TIMESTAMP functions.

Table 93. Format elements for ROUND, ROUND_TIMESTAMP, TRUNCATE, and TRUNC_TIMESTAMP

Format element	Rounding or truncating unit	ROUND example	TRUNCATE example
CC SCC	Century Rounds up to the start of the next century after the 50th year of the century (for example on 1951-01-01-00.00.00). Not valid for TIME argument.	Input Value: 1897-12-04-12.22.22.000000 Result: 1901-01-01-00.00.00.000000	Input Value: 1897-12-04-12.22.22.000000 Result: 1801-01-01-00.00.00.000000

Table 93. Format elements for ROUND, ROUND_TIMESTAMP, TRUNCATE, and TRUNC_TIMESTAMP (continued)

Format element	Rounding or truncating unit	ROUND example	TRUNCATE example
SYYYY YYYY YEAR SYEAR YYY YY Y	Year Rounds up on July 1st to January 1st of the next year. Not valid for TIME argument.	Input Value: 1897-12-04-12.22.22.000000 Result: 1898-01-01-00.00.00.000000	Input Value: 1897-12-04-12.22.22.000000 Result: 1897-01-01-00.00.00.000000
IYYY IYY IY I	ISO Year Rounds up on July 1st to the first day of the next ISO year. The first day of the ISO year is defined as the Monday of the first ISO week. Not valid for TIME argument.	Input Value: 1897-12-04-12.22.22.000000 Result: 1898-01-03-00.00.00.000000	Input Value: 1897-12-04-12.22.22.000000 Result: 1897-01-04-00.00.00.000000
Q	Quarter Rounds up on the 16th day of the second month of the quarter. Not valid for TIME argument.	Input Value: 1999-06-04-12.12.30.000000 Result: 1999-07-01-00.00.00.000000	Input Value: 1999-06-04-12.12.30.000000 Result: 1999-04-01-00.00.00.000000
MONTH MON MM RM	Month Rounds up on the 16th day of the month. Not valid for TIME argument.	Input Value: 1999-06-18-12.12.30.000000 Result: 1999-07-01-00.00.00.000000	Input Value: 1999-06-18-12.12.30.000000 Result: 1999-06-01-00.00.00.000000
WW	Same day of the week as the first day of the year. Rounds up on the 12th hour of the 4th day of the week, with respect to the first day of the year. Not valid for TIME argument.	Input Value: 2000-05-05-12.12.30.000000 Result: 2000-05-06-00.00.00.000000	Input Value: 2000-05-05-12.12.30.000000 Result: 2000-04-29-00.00.00.000000

Table 93. Format elements for ROUND, ROUND_TIMESTAMP, TRUNCATE, and TRUNC_TIMESTAMP (continued)

Format element	Rounding or truncating unit	ROUND example	TRUNCATE example
IW	Same day of the week as the first day of the ISO year. See "WEEK_ISO scalar function" for details. Rounds up on the 12th hour of the 4th day of the week, with respect to the first day of the ISO year. Not valid for TIME argument.	Input Value: 2000-05-05-12.12.30.000000 Result: 2000-05-08-00.00.00.000000	Input Value: 2000-05-05-12.12.30.000000 Result: 2000-05-01-00.00.00.000000
W	Same day of the week as the first day of the month. Rounds up on the 12th hour of the 4th day of the week, with respect to the first day of the month. Not valid for TIME argument.	Input Value: 2000-06-21-12.12.30.000000 Result: 2000-06-22-00.00.00.000000	Input Value: 2000-06-21-12.12.30.000000 Result: 2000-06-15-00.00.00.000000
DDD DD J	Day Rounds up on the 12th hour of the day. Not valid for TIME argument.	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-18-00.00.00.000000	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-17-00.00.00.000000
DAY DY D	Starting day of the week. Rounds up with respect to the 12th hour of the 4th day of the week. The first day of the week is based on the locale (see *locale-name*). Not valid for TIME argument.	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-21-00.00.00.000000	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-14-00.00.00.000000

Table 93. Format elements for ROUND, ROUND_TIMESTAMP, TRUNCATE, and TRUNC_TIMESTAMP (continued)

Format element	Rounding or truncating unit	ROUND example	TRUNCATE example
HH HH12 HH24	Hour Rounds up at 30 minutes.	Input Value: 2000-05-17-23.59.59.000000 Result: 2000-05-18-00.00.00.000000	Input Value: 2000-05-17-23.59.59.000000 Result: 2000-05-17-23.00.00.000000
MI	Minute Rounds up at 30 seconds.	Input Value: 2000-05-17-23.58.45.000000 Result: 2000-05-17-23.59.00.000000	Input Value: 2000-05-17-23.58.45.000000 Result: 2000-05-17-23.58.00.000000
SS	Second Rounds up at half a second.	Input Value: 2000-05-17-23.58.45.500000 Result: 2000-05-17-23.58.46.000000	Input Value: 2000-05-17-23.58.45.500000 Result: 2000-05-17-23.58.45.000000

Note: The format elements in Table 93 on page 486 must be specified in uppercase.

If a format element that applies to a time part of a value is specified for a date argument, the date argument is returned unchanged. If a format element that is not valid for a time argument is specified for a time argument, an error is returned (SQLSTATE 22007).

Notes

- **Determinism:** ROUND is a deterministic function. However, the following invocations of the function depend on the value of the special register CURRENT LOCALE LC_TIME.
  - Round of a datetime value when *locale-name* is not explicitly specified and one of the following is true:
    - *format-string* is not a constant
    - *format-string* is a constant and includes format elements that are locale sensitive

    Invocations of the function that depend on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621, 428EC, or 429BX).

Examples

- *Example 1*: Calculate the value of 873.726, rounded to 2, 1, 0, -1, -2, -3, and -4 decimal places, respectively.

```
VALUES (
  ROUND(873.726, 2),
  ROUND(873.726, 1),
  ROUND(873.726, 0),
  ROUND(873.726,-1),
  ROUND(873.726,-2),
  ROUND(873.726,-3),
  ROUND(873.726,-4) )
```

This example returns:

```
1         2         3         4         5         6         7
--------- --------- --------- --------- --------- --------- ---------
  873.730   873.700   874.000   870.000   900.000  1000.000     0.000
```

- *Example 2*: Calculate using both positive and negative numbers.

```
   VALUES (
     ROUND(3.5, 0),
     ROUND(3.1, 0),
     ROUND(-3.1, 0),
     ROUND(-3.5,0) )
```

This example returns:

```
1    2    3    4
---- ---- ---- ----
 4.0  3.0 -3.0 -4.0
```

- *Example 3*: Calculate the decimal floating-point number 3.12350 rounded to three decimal places.

```
   VALUES (
     ROUND(DECFLOAT('3.12350'), 3))
```

This example returns:

```
1
-----
3.12400
```

- *Example 4*: Set the host variable RND_DT with the input date rounded to the nearest month value.

```
   SET :RND_DATE = ROUND(DATE('2000-08-16'), 'MONTH');
```

The value set is 2000-09-01.

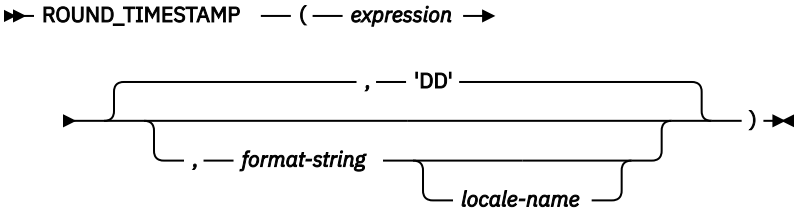
- *Example 5*: Set the host variable RND_TMSTMP with the input timestamp rounded to the nearest year value.

```
   SET :RND_TMSTMP = ROUND(TIMESTAMP('2000-08-14-17.30.00'),
                           'YEAR');
```

The value set is 2001-01-01-00.00.00.000000.

ROUND_TIMESTAMP

The ROUND_TIMESTAMP scalar function returns a TIMESTAMP based on a provided argument (*expression*), rounded to the unit specified in another argument (*format-string*).



The schema is SYSIBM.

If *format-string* is not specified, *expression* is rounded to the nearest day, as if `'DD'` is specified for *format-string*.

expression
An expression that returns a value of one of the following built-in data types: a DATE or a TIMESTAMP.

format-string
An expression that returns a built-in character string data type with an actual length that is not greater than 255 bytes. The format element in *format-string* specifies how *expression* should be rounded. For example, if format-string is `'DD'`, the timestamp that is represented by *expression* is rounded to the nearest day. Leading and trailing blanks are removed from the string, and the resulting substring must be a valid format element for a timestamp (SQLSTATE 22007). The default is `'DD'`.

Allowable values for *format-string* are listed in the table of format elements found in the description of the ROUND function.

locale-name
A character constant that specifies the locale used to determine the first day of the week when using format model values DAY, DY, or D. The value of *locale-name* is not case sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming, see "Locale names for SQL and XQuery". If *locale-name* is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result of the function is a TIMESTAMP with a timestamp precision of:

- *p* when the data type of *expression* is TIMESTAMP(*p*)
- 0 when the data type of *expression* is DATE
- 6 otherwise.

The result can be null; if any argument is null, the result is the null value.

Notes

- **Determinism:** ROUND_TIMESTAMP is a deterministic function. However, the following invocations of the function depend on the value of the special register CURRENT LOCALE LC_TIME.
  - Round of a date or timestamp value when *locale-name* is not explicitly specified and one of the following is true:
    - *format-string* is not a constant
    - *format-string* is a constant and includes format elements that are locale sensitive

    Invocations of the function that depend on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621, 428EC, or 429BX).

Example

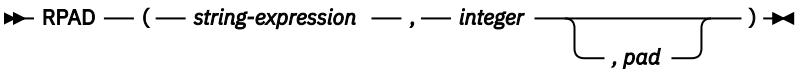
Set the host variable *RND_TMSTMP* with the input timestamp rounded to the nearest year value.

```
SET :RND_TMSTMP = ROUND_TIMESTAMP('2000-08-14-17.30.00', 'YEAR');
```

The value set is 2001-01-01-00.00.00.000000.

RPAD

The RPAD function returns a string composed of *string-expression* padded on the right, with *pad* or blanks.



The schema is SYSIBM.

The RPAD function treats leading or trailing blanks in *string-expression* as significant. Padding will only occur if the actual length of *string-expression* is less than *integer*, and *pad* is not an empty string.

string-expression
An expression that specifies the source string. The expression must return a value that is a built-in character string, graphic string, BINARY, VARBINARY, numeric, or datetime data type. CLOB and DBCLOB are supported through implicit casting. If the value is a CLOB, it is implicitly cast to VARCHAR before the function is evaluated. If the value is a DBCLOB, it is implicitly cast to VARGRAPHIC before the function is evaluated. If the data type of the *string-expression* value is numeric or datetime, the value is implicitly cast to VARCHAR before the function is evaluated. The data type of *string-expression* cannot be a BLOB (SQLSTATE 42815).

integer
An expression that specifies the actual length of the result in the string units of *string-expression*. The expression must return a value that is a built-in numeric, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported through implicit casting. If the expression is not of type INTEGER, it is cast to INTEGER before the function is evaluated. The value must be zero or a positive integer that is less than or equal to the maximum length for the result data type in the string units of *string-expression*.

pad
An expression that specifies the string with which to pad. The expression must return a value that is a built-in character string, graphic string, BINARY, VARBINARY, numeric, or datetime data type. CLOB and DBCLOB are supported through implicit casting. If the value is a CLOB, it is implicitly cast to VARCHAR before the function is evaluated. If the value is a DBCLOB, it is implicitly cast to VARGRAPHIC before the function is evaluated. If the data type of the *pad* value is numeric or datetime, the value is implicitly cast to VARCHAR before the function is evaluated. The data type of *pad* cannot be a BLOB (SQLSTATE 42815).

If *pad* is not specified, the pad character is determined as follows:

- SBCS blank character if *string-expression* is a character string.
- Ideographic blank character, if *string-expression* is a graphic string. For graphic string in an EUC database, X'3000' is used. For graphic string in a Unicode database, X'0020' is used.
- Hexadecimal zero (X'00'), if *string-expression* is a binary string.

The data type of the result depends on the data type of the *string-expression*:

- VARCHAR if the data type is VARCHAR or CHAR
- VARGRAPHIC if the data type is VARGRAPHIC or GRAPHIC
- VARBINARY if the data type is VARBINARY or BINARY

The result of the function is a varying length string that has the same string unit and code page as *string-expression*. The value of *string-expression* and the value of *pad* must have compatible data types. If the *string-expression* and *pad* have different code pages, then *pad* is converted to the code page of *string-expression*. If either *string-expression* or *pad* is FOR BIT DATA or binary, no character conversion occurs.

The length attribute of the result depends on whether the value for *integer* is available when the SQL statement containing the function invocation is compiled (for example, if it is specified as a constant or a constant expression) or available only when the function is executed (for example, if it is specified as the result of invoking a function). When the value is available when the SQL statement containing the function invocation is compiled, if *integer* is greater than zero, the length attribute of the result is *integer*. If *integer* is 0, the length attribute of the result is 1. When the value is available only when the function is executed, the length attribute of the result is determined according to the following table:

Table 94. Determining the result length when integer is available only when the function is executed

Data type of *string-expression*	**Result data type length**
CHAR(*n*) or VARCHAR(*n*), BINARY(*n*) or VARBINARY(*n*)	Minimum of *n*+100 and 32 672
GRAPHIC(*n*) or VARGRAPHIC(*n*)	Minimum of *n*+100 and 16 336

Table 94. Determining the result length when integer is available only when the function is executed (continued)

Data type of *string-expression*	Result data type length
CHAR(*n*) or VARCHAR(*n*) or GRAPHIC(*n*) or VARGRAPHIC(*n*) with string units of CODEUNITS32 (Unicode database only)	Minimum of *n*+100 and 8 168

The actual length of the result is determined from *integer*. If *integer* is 0 the actual length is 0, and the result is the empty result string. If *integer* is less than the actual length of *string-expression*, the actual length is *integer* and the result is truncated.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Assume that NAME is a VARCHAR(15) column containing the values "Chris", "Meg", and "Jeff". The following query will completely pad out a value on the right with periods:

```
SELECT RPAD(NAME,15,'.' ) AS NAME FROM T1;
```

  returns:

```
NAME
---------------
Chris..........
Meg............
Jeff...........
```

- *Example 2*: Assume that NAME is a VARCHAR(15) column containing the values "Chris", "Meg", and "Jeff". The following query will completely pad out a value on the right with *pad* (note that in some cases there is a partial instance of the padding specification):

```
SELECT RPAD(NAME,15,'123' ) AS NAME FROM T1;
```

  returns:

```
NAME
---------------
Chris1231231231
Meg123123123123
Jeff12312312312
```

- *Example 3*: Assume that NAME is a VARCHAR(15) column containing the values "Chris", "Meg", and "Jeff". The following query will only pad each value to a length of 5:

```
SELECT RPAD(NAME,5,'.' ) AS NAME FROM T1;
```

  returns:

```
NAME
-----
Chris
Meg..
Jeff.
```

- *Example 4*: Assume that NAME is a CHAR(15) column containing the values "Chris", "Meg", and "Jeff". Note that the result of RTRIM is a varying length string with the blanks removed:

```
SELECT RPAD(RTRIM(NAME),15,'.' ) AS NAME FROM T1;
```

  returns:

```
NAME
---------------
```

```
Chris.....
Meg.......
Jeff......
```

- *Example 5*: Assume that NAME is a VARCHAR(15) column containing the values "Chris", "Meg", and "Jeff". Note that "Chris" is truncated, "Meg" is padded, and "Jeff" is unchanged:

```
SELECT RPAD(NAME,4,'.' ) AS NAME FROM T1;
```

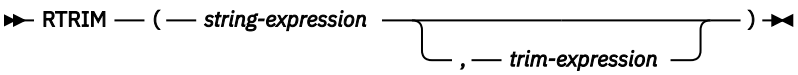
returns:

```
NAME
----
Chri
Meg.
Jeff
```

RTRIM

The RTRIM function removes any of the specified characters from the end of a string.

The RTRIM function removes any of the characters contained in a trim expression from the end of a string expression. The character search compares the binary representation of each character (consisting of one or more bytes) in the trim expression to the binary representation of each character (consisting of one or more bytes) at the end of the string expression. The database collation does not affect the search. If the string expression is defined as FOR BIT DATA or is a binary type, the search compares each byte in the trim expression to the byte at the end of the string expression.



The schema is SYSIBM. (The SYSFUN version of this function that uses a single parameter continues to be available with support for CLOB arguments.)

string-expression

An expression that specifies the source string.

- If only one argument is specified, the expression must return a built-in character string, graphic string, binary string, CLOB or DBCLOB value, Boolean value, numeric value, or datetime value. If the value is:
  - Not a CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BINARY, VARBINARY, or DBCLOB value, it is implicitly cast to VARCHAR before the function is evaluated
  - A DBCLOB value, it is implicitly cast to VARGRAPHIC before the function is evaluated

  The data type of the string expression cannot be BLOB (SQLSTATE 42815).
- If both arguments are specified, the expression must return a built-in character string, Boolean value, numeric value, or datetime value. If the data type of the *string-expression* value is not a string data type, the value is implicitly cast to VARCHAR before the function is evaluated. The actual length of a CLOB value is limited to the maximum size of a VARCHAR data type (SQLSTATE 22001). The actual length of a BLOB value is limited to the maximum size of a VARBINARY data type (SQLSTATE 22001). The actual length of a DBCLOB value is limited to the maximum size of a VARGRAPHIC data type (SQLSTATE 22001).

trim-expression

An expression that specifies the characters that are being removed from the end of a *string-expression*. The expression must be a value that is a built-in string, numeric, or datetime data type.

- If the data type of the *trim-expression* is not a string, then the value is implicitly cast to VARCHAR before the function is evaluated.
- If the data type of the *trim-expression* is a CLOB, then the actual length of the value is limited to the maximum size of a VARCHAR (SQLSTATE 22001).

- If the data type of the *trim-expression* is a DBCLOB, then the actual length of the value is limited to the maximum size of a VARGRAPHIC (SQLSTATE 22001).
- If the data type of *trim-expression* is a BLOB, then the actual length of the value is limited to the maximum size of a VARBINARY (SQLSTATE 22001).
- If the *string-expression* is not defined as FOR BIT DATA, then the *trim-expression* cannot be defined as FOR BIT DATA (SQLSTATE 42815).

When a *trim-expression* is not specified, the data type of the *string-expression* determines the default value used:

- A double byte blank if the *string-expression* is a graphic string in a DBCS or EUC database
- A UCS-2 blank if the *string-expression* is a graphic string in a Unicode database
- A value of X'20' if the *string-expression* is a FOR BIT DATA string
- A value of X'00' if the *string-expression* is a binary string
- A single-byte blank for all other cases

The *string-expression* and *trim-expression* values must have compatible data types. If one of these arguments is a FOR BIT DATA character string, the other argument cannot be a graphic string (SQLSTATE 42846). A combination of character string and graphic string arguments can be used only in a Unicode database (SQLSTATE 42815).

Result

The data type of the result depends on the data type of the *string-expression*.

- VARCHAR if the data type is VARCHAR or CHAR
- CLOB if the data type is CLOB
- VARBINARY if the data type is VARBINARY or BINARY
- BLOB if the data type is BLOB
- VARGRAPHIC if the data type is VARGRAPHIC or GRAPHIC
- DBCLOB if the data type is DBCLOB

The length attribute of the result data type is the same as the length attribute of the *string-expression* data type.

The actual length of the result for character or binary strings is the length of *string-expression* minus the number of string units removed. The actual length of the result for graphic strings is the length of *string-expression* minus the number of string units removed. If all of the characters are removed, the result is an empty string with a length of zero.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Use the RTRIM function when the host variable HELLO is defined as CHAR(6) and has a value of 'Hello '.

```
VALUES RTRIM(:HELLO)
```

  The result is 'Hello'. When a *trim-expression* is not specified only blanks are removed. The host variable is declared as CHAR(9) and is blank-padded up to 9 bytes.
- *Example 2*: Use the RTRIM function to remove the characters specified in the *trim-expression* from the end of the *string-expression*.

```
VALUES RTRIM('...$VAR$...', '$.')
```

  The result is `'...$VAR'`.

- *Example 3*: Use the RTRIM function to remove the characters specified in the *trim-expression* from the end of the *string-expression*.

```
VALUES RTRIM('((-78.0) )', '-0. ()')
```

The result is `'((-78'`. When removing characters and blanks, you must include a blank in the *trim-expression*.

RTRIM (SYSFUN schema)

Returns the characters of the argument with trailing blanks removed.

The schema is SYSFUN.

expression
The *expression* can be of any built-in character string data type. For a VARCHAR the maximum length is 4 000 bytes and for a CLOB the maximum length is 1 048 576 bytes.

The result of the function is:

- VARCHAR(4000) if *expression* is VARCHAR (not exceeding 4 000 bytes) or CHAR
- CLOB(1M) if *expression* is CLOB or LONG VARCHAR

The result can be null; if *expression* is null, the result is the null.

SECLABEL

The SECLABEL function returns an unnamed security label with a data type of DB2SECURITYLABEL. Use the SECLABEL function to insert a security label with given component values without having to create a named security label.

The schema is SYSIBM.

security-policy-name
A string that specifies a security policy that exists at the current server (SQLSTATE 42704). The string must be a character or graphic string constant or host variable.

security-label-string
An expression that returns a valid representation of a security label for the security policy named by *security-policy-name* (SQLSTATE 4274I). The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type.

Examples

- *Example 1*: The following statement inserts a row in table REGIONS which is protected by the security policy named CONTRIBUTIONS. The security label for the row to be inserted is given by the SECLABEL function. The security policy CONTRIBUTIONS has two components. The security label given has the element LIFE MEMBER for first component, the elements BLUE and YELLOW for the second component.

```
INSERT INTO REGIONS
  VALUES (SECLABEL('CONTRIBUTIONS', 'LIFE MEMBER:(BLUE,YELLOW)'),
    1, 'Northeast')
```

- *Example 2*: The following statement inserts a row in table CASE_IDS which is protected by the security policy named TS_SECPOLICY, which has three components. The security label is provided by the

SECLABEL function. The security label inserted has the element HIGH PROFILE for the first component, the empty value for the second component and the element G19 for the third component.

```
INSERT INTO CASE_IDS
  VALUES (SECLABEL('TS_SECPOLICY', 'HIGH PROFILE:():G19') , 3, 'KLB')
```

SECLABEL_BY_NAME

The SECLABEL_BY_NAME function returns the specified security label. The security label returned has a data type of DB2SECURITYLABEL. Use this function to insert a named security label.

The schema is SYSIBM.

security-policy-name
A string that specifies a security policy that exists at the current server (SQLSTATE 42704). The string must be a character or graphic string constant or host variable.

security-label-name
An expression that returns the name of a security label that exists at the current server for the security policy named by *security-policy-name* (SQLSTATE 4274I). The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type.

Examples

In the following examples, Tina is trying to insert a row in table REGIONS which is protected by the security policy named CONTRIBUTIONS. Tina wants the row to be protected by the security label named EMPLOYEESECLABEL.

- *Example 1*: This statement fails because CONTRIBUTIONS.EMPLOYEESECLABEL is an unknown identifier:

```
INSERT INTO REGIONS
VALUES (CONTRIBUTIONS.EMPLOYEESECLABEL, 1, 'Southwest')   -- incorrect
```

- *Example 2*: This statement fails because the first value is a string, it does not have a data type of DB2SECURITYLABEL:

```
INSERT INTO REGIONS
VALUES ('CONTRIBUTIONS.EMPLOYEESECLABEL', 1, 'Southwest') -- incorrect
```

- *Example 2*: This statement succeeds because the SECLABEL_BY_NAME function returns a security label that has a data type of DB2SECURITYLABEL:

```
INSERT INTO REGIONS
VALUES (SECLABEL_BY_NAME('CONTRIBUTIONS', 'EMPLOYEESECLABEL'),
  1, 'Southwest')                                        -- correct
```

SECLABEL_TO_CHAR

The SECLABEL_TO_CHAR function accepts a security label and returns a string that contains all elements in the security label. The string is in the security label string format.

The schema is SYSIBM.

security-policy-name
A string that specifies a security policy that exists at the current server (SQLSTATE 42704). The string must be a character or graphic string constant or host variable.

security-label
An expression that returns a security label value that is valid for the security policy named by *security-policy-name* (SQLSTATE 4274I). The expression must return a value that is a built-in SYSPROC.DB2SECURITYLABEL distinct type.

The result of the function is VARCHAR(32672 OCTETS). The result can be null; if the second argument is null, the result is the null value.

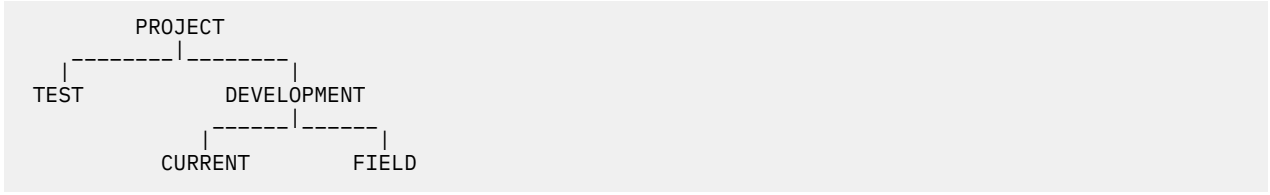
Notes

- If the authorization ID of the statement executes this function on a security label being read from a column with a data type of DB2SECURITYLABEL then that authorization ID's LBAC credentials might affect the output of the function. In such a case an element is not included in the output if the authorization ID does not have read access to that element. An authorization ID has read access to an element if its LBAC credentials would allow it to read data that was protected by a security label containing only that element, and no others.

  For the rule set DB2LBACRULES only components of type TREE can contain elements that you do not have read access to. For other types of component, if any one of the elements block read access then you will not be able to read the row at all. So only components of type tree will have elements excluded in this way.

Example

The EMP table has two columns, RECORDNUM and LABEL; RECORDNUM has data type INTEGER, and LABEL has type DB2SECURITYLABEL. Table EMP is protected by security policy DATA_ACCESSPOLICY, which uses the DB2LBACRULES rule set and has only one component (GROUPS, of type TREE). GROUPS has five elements: PROJECT, TEST, DEVELOPMENT, CURRENT, AND FIELD. The following diagram shows the relationship of these elements to one another:



The EMP table contains the following data:

```
RECORDNUM  LABEL
---------  ----------------
        1  PROJECT
        2  (TEST, FIELD)
        3  (CURRENT, FIELD)
```

The user whose ID is Djavan holds a security label for reading that contains only the DEVELOPMENT element. This means that Djavan has read access to the DEVELOPMENT, CURRENT, and FIELD elements:

```
   SELECT RECORDNUM, SECLABEL_TO_CHAR('DATA_ACCESSPOLICY', LABEL) FROM EMP
```

returns:

```
RECORDNUM  LABEL
---------  ----------------
        2  FIELD
        3  (CURRENT, FIELD)
```

The row with a RECORDNUM value of 1 is not included in the output, because Djavan's LBAC credentials do not allow him to read that row. In the row with a RECORDNUM value of 2, element TEST is not included in the output, because Djavan does not have read access to that element; Djavan would not have been able to access the row at all if TEST were the only element in the security label. Because Djavan has read access to elements CURRENT and FIELD, both elements appear in the output.

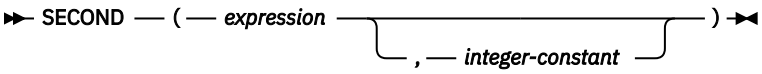
Now Djavan is granted an exemption to the DB2LBACREADTREE rule. This means that no element of a TREE type component will block read access. The same query returns:

```
RECORDNUM  LABEL
---------  ----------------
        1  PROJECT
        2  (TEST, FIELD)
        3  (CURRENT, FIELD)
```

This time the output includes all rows and all elements, because the exemption gives Djavan read access to all of the elements.

SECOND

The SECOND function returns the seconds part of a value with optional fractional seconds.



The schema is SYSIBM.

expression
An expression that returns a value of one of the following built-in data types: DATE, TIME, TIMESTAMP, time duration, timestamp duration, or a valid character string representation of a date, time, or timestamp that is not a CLOB. If *expression* is a DATE or a valid string representation of a date, it is first converted to a TIMESTAMP(0) value, assuming a time of exactly midnight (00.00.00). If expression is a valid string representation of a timestamp, it is first converted to a TIMESTAMP(12) value. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

integer-constant
An integer constant representing the scale for the fractional seconds. The value must be in the range 0 through 12.

The result of the function with a single argument is a large integer. The result of the function with two arguments is DECIMAL(2+*s*,*s*) where *s* is the value of the *integer-constant*. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The other rules depend on the data type of the first argument and the number of arguments:

- If the first argument is a DATE, TIME, TIMESTAMP, or valid string representation of a date, time, or timestamp:
  - If only one argument is specified, the result is the seconds part of the value (0 and 59).
  - If both arguments are specified, the result is the seconds part of the value (0 to 59) and *integer-constant* digits of the fractional seconds part of the value where applicable. If there are no fractional seconds in the value, then zeros are returned.
- If the first argument is a time duration or timestamp duration:
  - If only one argument is specified, the result is the seconds part of the value (-99 to 99).
  - If both arguments are specified, the result is the seconds part of the value (-99 to 99) and *integer-constant* digits of the fractional seconds part of the value where applicable. If there are no fractional seconds in the value then zeros are returned. A nonzero result has the same sign as the argument.

Examples

- *Example 1*: Assume that the host variable TIME_DUR (decimal(6,0)) has the value 153045.

```
SELECT SECOND(:TIME_DUR)
  FROM SYSIBM.SYSDUMMY1
```

  Returns the value 45.

- *Example 2*: Assume that the column RECEIVED (whose data type is TIMESTAMP) has an internal value equivalent to 1988-12-25-17.12.30.000000.

```
SELECT SECOND(RECEIVED)
  FROM IN_TRAY
```

  Returns the value 30.

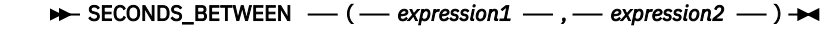
- *Example 3*: Get the seconds with fractional seconds from a current timestamp with milliseconds.

```
SELECT SECOND (CURRENTTIMESTAMP(3), 3)
  FROM SYSIBM.SYSDUMMY1
```

  Returns a DECIMAL(5,3) value based on the current timestamp that could be something like 54.321.

SECONDS_BETWEEN

The SECONDS_BETWEEN function returns the number of full seconds between the specified arguments.



The schema is SYSIBM.

expression1
An expression that specifies the first datetime value to compute the number of full seconds between two datetime values. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression1* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

expression2
An expression that specifies the second datetime value to compute the number of full seconds between two datetime values. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression2* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

If there is less than a full second between *expression1* and *expression2*, the result is zero. If *expression1* is later than *expression2*, the result is positive. If *expression1* is earlier than *expression2*, the result is negative. In NPS compatibility mode, this function always returns a positive number. If *expression1* or *expression2* contains time information, this information is also used to determine the number of full seconds. If *expression1* or *expression2* does not contain time information, a time of midnight (00.00.00) is used for the argument that is missing time information.

The result of the function is a BIGINT. If either argument can be null, the result can be null. If either argument is null, the result is the null value.

Examples

1. Set the host variable NUM_SECONDS to the number of full seconds between 2012-03-01-01.00.00 and 2012-02-28-00.00.00.

```
SET :NUM_SECONDS = SECONDS_BETWEEN(TIMESTAMP '2012-03-01-01.00.00',
                                   TIMESTAMP '2012-02-28-00.00.00')
```

   The host variable NUM_SECONDS is set to 176400; 86400 of the seconds are incurred because of February 29, 2012.

2. Set the host variable NUM_SECONDS to the number of full seconds between 2013-09-11-23.59.59.999999 and 2013-09-01-00.00.00.000000.

```
SET :NUM_SECONDS = SECONDS_BETWEEN(TIMESTAMP '2013-09-11-23.59.59.999999',
                                   TIMESTAMP '2013-09-01-00.00.00.000000')
```

The host variable NUM_SECONDS is set to 950399 because there are 0.000001 seconds less than a full 950400 seconds between the arguments. It is positive because the first argument is later than the second argument.

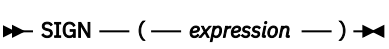
3. Set the host variable NUM_SECONDS to the number of full seconds between 2013-09-01-00.00.00.000000 and 2013-09-11-23.59.59.999999.

```
SET :NUM_SECONDS = SECONDS_BETWEEN(TIMESTAMP '2013-09-01-00.00.00.000000',
                                   TIMESTAMP '2013-09-11-23.59.59.999999')
```

The host variable NUM_SECONDS is set to -950399 because there are 0.000001 seconds less than a full 950400 seconds between the arguments. It is negative because the first argument is earlier than the second argument.

SIGN

Returns an indicator of the sign of the argument.



The schema is SYSIBM. (The SYSFUN version of the SIGN function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type. DECIMAL and REAL values are converted to double-precision floating-point numbers for processing by the function.

If the argument is less than zero, -1 is returned. If the argument is the decimal floating-point value of -0, the decimal floating-point value of -0 is returned. If argument equals zero, 0 is returned. If argument is greater than zero, 1 is returned.

The result of the function is:

- SMALLINT if the argument is SMALLINT
- INTEGER if the argument is INTEGER
- BIGINT if the argument is BIGINT
- DECFLOAT(*n*) if the argument is DECFLOAT(*n*)
- DOUBLE otherwise.

The result can be null; if the argument is null, the result is the null value.

Example

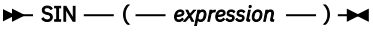
Assume that host variable PROFIT is a large integer with a value of 50000.

```
VALUES SIGN(:PROFIT)
```

Returns the value 1.

SIN

Returns the sine of the argument, where the argument is an angle expressed in radians.



The schema is SYSIBM. (The SYSFUN version of the SIN function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

SINH

Returns the hyperbolic sine of the argument, where the argument is an angle expressed in radians.

The schema is SYSIBM.

expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

SMALLINT

The SMALLINT function returns a small integer (a binary integer with a precision of 15 bits) representation of a value of a different data type.

Numeric to SMALLINT

String to SMALLINT

Boolean to SMALLINT

The schema is SYSIBM.

Numeric to SMALLINT

numeric-expression
An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a small integer column or variable. The fractional part of the argument is truncated. If the whole part of the argument is not within the range of small integers, an error is returned (SQLSTATE 22003).

String to SMALLINT

string-expression
An expression that returns a value that is a character-string or Unicode graphic-string representation of a number with a length not greater than the maximum length of a character constant.

The result is the same number that would result from CAST(*string-expresssion* AS SMALLINT). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018). If the whole part of the argument is not within the range of small integers, an error is returned (SQLSTATE 22003). The data type of *string-expression* must not be CLOB or DBCLOB (SQLSTATE 42884).

Boolean to SMALLINT

boolean-expression
An expression that returns a Boolean value (TRUE or FALSE). The result is either 1 (for TRUE) or 0 (for FALSE).

Result

The result of the function is a small integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Notes

- **Increasing portability of applications:** If the first argument is numeric, or if the first argument is a string and the length argument is specified, use the "CAST specification" on page 152 instead of this function to increase the portability of your applications.

Examples

- *Example 1*: Using the EMPLOYEE table, select a list containing salary (SALARY) divided by education level (EDLEVEL). Truncate any decimal in the calculation. The list should also contain the values used in the calculation and the employee number (EMPNO).

```
SELECT SMALLINT(SALARY / EDLEVEL), SALARY, ESDLEVEL, EMPNO
  FROM EMPLOYEE
```

- *Example 2*: The following statement returns the value 1 of data type SMALLINT.

```
values SMALLINT(TRUE)
```

- *Example 3*: The following statement returns the value 0 of data type SMALLINT.

```
values SMALLINT(3>3)
```

SOUNDEX

Returns a 4-character code representing the sound of the words in the argument. The result can be used to compare with the sound of other strings.

The schema is SYSFUN.

expression
An expression that returns a value of CHAR or VARCHAR data type. The length of the value must not exceed 4 000 bytes. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed. The function interprets data that is passed to it as if it were ASCII characters, even if it is encoded in UTF-8.

The result of the function is CHAR(4). The result can be null; if the argument is null, the result is the null value.

The SOUNDEX function is useful for finding strings for which the sound is known but the precise spelling is not. It makes assumptions about the way that letters and combinations of letters sound that can help to search out words with similar sounds. The comparison can be done directly or by passing the strings as arguments to the DIFFERENCE function.

Example

Using the EMPLOYEE table, find the EMPNO and LASTNAME of the employee with a surname that sounds like 'Loucesy'.

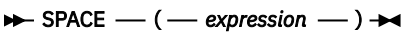
```
SELECT EMPNO, LASTNAME FROM EMPLOYEE
  WHERE SOUNDEX(LASTNAME) = SOUNDEX('Loucesy')
```

This example returns the following output:

```
EMPNO  LASTNAME
------ ---------------
000110 LUCCHESSI
```

SPACE

Returns a character string consisting of blanks with length specified by the argument.



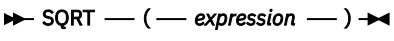
The schema is SYSFUN.

expression
An expression that returns a value of built-in SMALLINT or INTEGER data type.

The result of the function is VARCHAR(4000 OCTETS). The result can be null; if the argument is null, the result is the null value.

SQRT

The SQRT function returns the square root of a number.



The schema is SYSIBM. (The SYSFUN version of the SQRT function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type. If the argument is decimal floating-point, the operation is performed in decimal floating-point; otherwise, the argument is converted to double-precision floating-point for processing by the function.

If the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*); otherwise, the result is a double-precision floating-point number.

The result can be null; if the argument is null, the result is the null value.

Notes

- ***Results involving DECFLOAT special values:*** If the argument is a special decimal floating-point value, the rules for general arithmetic operations for decimal floating-point apply. See "General arithmetic operation rules for decimal floating-point" on page 142 in "Expressions" on page 132.

Example

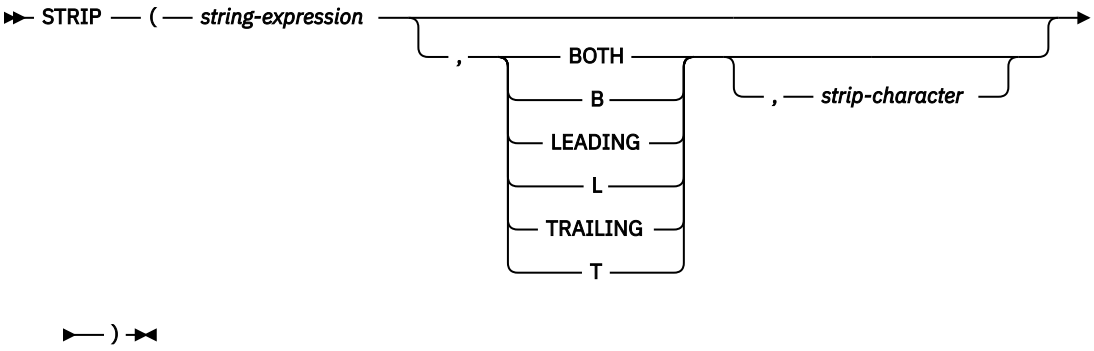
Assume that SQUARE is a DECIMAL(2,1) host variable with a value of 9.0.

```
VALUES SQRT(:SQUARE)
```

Returns the approximate value 3.00.

STRIP

The STRIP function removes blanks or another specified character from the end, the beginning, or both ends of a string expression.



The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature.

The STRIP function is identical to the TRIM scalar function.

string-expression
An expression that specifies the string from which the result is derived. The expression must return a value that is a built-in CHAR, VARCHAR, BINARY, VARBINARY, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, BINARY, VARBINARY, GRAPHIC, or VARGRAPHIC data type, it is cast to VARCHAR before the function is evaluated.

BOTH, LEADING, or TRAILING
Specifies whether characters are removed from the beginning, the end, or from both ends of the string expression. If this argument is not specified, the characters are removed from both the end and the beginning of the string.

strip-character
A single-character constant that specifies the character that is to be removed. The *strip-character* can be any character whose UTF-32 encoding is a single character or a single digit numeric value. The binary representation of the character is matched.

If *strip-character* is not specified and:

- If the *string-expression* is a DBCS graphic string, the default *strip-character* is a DBCS blank, whose code point is dependent on the database code page
- If the *string-expression* is a UCS-2 graphic string, the default *strip-character* is a UCS-2 blank (X'0020')
- If the *string-expression* is a binary string, the default *strip-character* is a hexadecimal zero (X'00')
- Otherwise, the default *strip-character* is an SBCS blank (X'20')

The value for *string-expression* and the value for *strip-character* must have compatible data types.

The data type of the result depends on the data type of the *string-expression*:

- VARCHAR if the data type is VARCHAR or CHAR
- VARGRAPHIC if the data type is VARGRAPHIC or GRAPHIC
- VARBINARY if the data type is VARBINARY or BINARY

The result is a varying-length string with the same maximum length as the length attribute of the *string-expression*. The actual length of the result is the length of the *string-expression* minus the number of string units that are removed. If all of the characters are removed, the result is an empty varying-length string. The code page of the result is the same as the code page of the *string-expression*.

Example

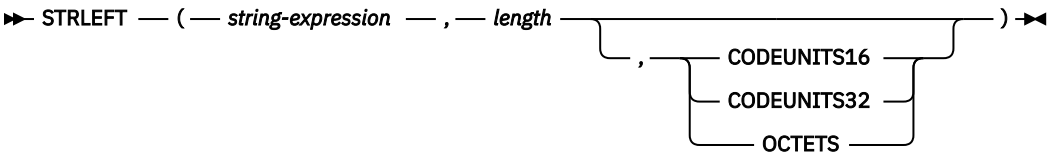
Assume that the host variable BALANCE of type CHAR(9) has a value of '000345.50'.

```
SELECT STRIP(:BALANCE, LEADING, '0'),
FROM SYSIBM.SYSDUMMY1
```

returns the value '345.50'.

STRLEFT

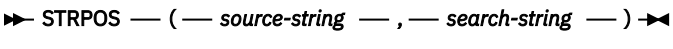
The STRLEFT function returns the leftmost string of *string-expression* of length *length*, expressed in the specified string unit.



The schema is SYSIBM.

The STRLEFT scalar function is a synonym for the LEFT scalar function.

STRPOS

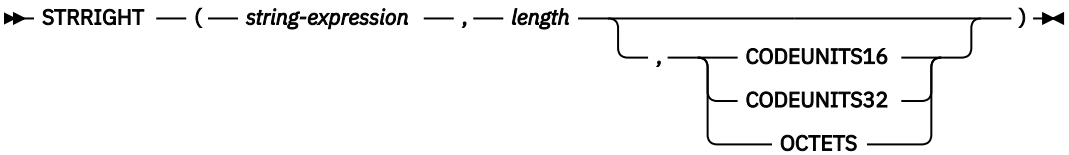


The schema is SYSIBM.

STRPOS is a synonym for POSSTR.

STRRIGHT

The STRRIGHT function returns the right most string of *string-expression* of length *length*, expressed in the specified string unit.

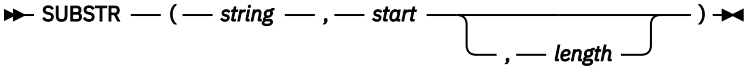


The schema is SYSIBM.

The STRRIGHT scalar function is a synonym for the RIGHT scalar function.

SUBSTR

The SUBSTR function returns a substring of a string.



The schema is SYSIBM.

string
The input expression, which specifies the string from which the substring is to be derived. The expression must return a value that is a built-in character string, numeric value, Boolean value, or datetime value. If the value is not a character string, it is implicitly cast to VARCHAR before the

function is evaluated. Any number (zero or more) contiguous string units of this expression constitute a substring of this expression.

start
An expression that specifies the position, relative to the beginning of the input expression, from which the substring is to be calculated. For example:

- Position 1 is the first string unit of the input expression. The statement SUBSTR('abcd',1,2) returns 'ab'.
- Position 2 is one position to the right of position 1. The statement SUBSTR('abcd',2,2) returns 'bc'.

The expression must return a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC value. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

length
An expression that specifies the length of the result. If specified, the expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value of the integer must be in the range of 0 to *n*, where *n* equals (the length attribute of *string* in string units) - *start* + 1 (SQLSTATE 22011 if out of range).

If *length* is explicitly specified, *string* is effectively padded on the right with the necessary number of blank characters (single-byte for character strings; double-byte for graphic strings) or hexadecimal zero characters (for binary strings) so that the specified substring exists. The default length is the number of string units from the string unit specified by *start* to the last string unit of *string*. However, if *string* is a varying-length string with a length less than *start*, the default is zero and the result is the empty string. It must be specified as number of string units in the context of the database code page and not the application code page. (For example, the column NAME with a data type of VARCHAR(18) and a value of 'MCKNIGHT' will yield an empty string with SUBSTR(NAME,10)).

If *string* is:

- A fixed-length string, the default length is LENGTH(*string*) - *start* + 1
- A varying-length string, the default length is either zero or LENGTH(*string*) - *start* + 1, whichever is greater.

Result

If *string* is a character string, the result is a character string represented in the code page and string units of its first argument. If it is a binary string, the result is a binary string. If it is a graphic string, the result is a graphic string represented in the code page and string units of its first argument. If the first argument is a host variable that is not a binary string and not a FOR BIT DATA character string, the code page of the result is the database code page. If any argument of the SUBSTR function can be null, the result can be null; if any argument is null, the result is the null value.

Table 95 on page 507 shows that the result type and length of the SUBSTR function depend on the type and attributes of its inputs.

Table 95. Data Type and Length of SUBSTR Result

String Argument Data Type	Length Argument	Result Data Type
CHAR(A)	constant (*l*<*n*) If the units of *string* are: • OCTETS, *n*=256 • CODEUNITS32, *n*=64	CHAR(*l*)
CHAR(A)	not specified but *start* argument is a constant	CHAR(A-*start*+1)
CHAR(A)	not a constant	VARCHAR(A)

Table 95. Data Type and Length of SUBSTR Result (continued)

String Argument Data Type	Length Argument	Result Data Type
VARCHAR(A)	constant (*l*<*n*) If the units of *string* are: • OCTETS, *n*=256 • CODEUNITS32, *n*=64	CHAR(*l*)
VARCHAR(A)	constant (*m*<*l*<*n*) If the units of *string* are: • OCTETS, *m*=256 and *n*=32673 • CODEUNITS32, *m*=63 and *n*=8169	VARCHAR(*l*)
VARCHAR(A)	not a constant or not specified	VARCHAR(A)
CLOB(A)	constant (*l*)	CLOB(*l*)
CLOB(A)	not a constant or not specified	CLOB(A)
GRAPHIC(A)	constant (*l*<*n*) If the units of *string* are: • double-bytes or CODEUNITS16, *n*=128 • CODEUNITS32, *n*=64	GRAPHIC(*l*)
GRAPHIC(A)	not specified but *start* argument is a constant	GRAPHIC(A-*start*+1)
GRAPHIC(A)	not a constant	VARGRAPHIC(A)
VARGRAPHIC(A)	constant (*l*<*n*) If the units of *string* are: • double-bytes or CODEUNITS16, *n*=128 • CODEUNITS32, *n*=64	GRAPHIC(*l*)
VARGRAPHIC(A)	constant (*m*<*l*<*n*) If the units of *string* are: • double-bytes or CODEUNITS16, *m*=127 and *n*=16337 • CODEUNITS32, *m*=63 and *n*=8169	VARGRAPHIC(*l*)
VARGRAPHIC(A)	not a constant	VARGRAPHIC(A)
DBCLOB(A)	constant (*l*)	DBCLOB(*l*)
DBCLOB(A)	not a constant or not specified	DBCLOB(A)
BINARY(A)	constant (*l*<256)	BINARY(*l*)
BINARY(A)	not specified but *start* argument is a constant	BINARY(A-*start*+1)
BINARY(A)	not a constant	VARBINARY(A)
VARBINARY(A)	constant (*l*<256)	BINARY(*l*)
VARBINARY(A)	constant (255<*l*<32673)	VARBINARY(*l*)
VARBINARY(A)	not a constant or not specified	VARBINARY(A)
BLOB(A)	constant (*l*)	BLOB(*l*)

Table 95. Data Type and Length of SUBSTR Result (continued)

String Argument Data Type	Length Argument	Result Data Type
BLOB(A)	not a constant or not specified	BLOB(A)

Note: The LONG VARCHAR and LONG VARGRAPHIC data types continue to be supported but are deprecated and not recommended.

Notes

- In dynamic SQL, *string*, *start*, and *length* can be represented by a parameter marker. If a parameter marker is used for *string*, the data type of the operand will be VARCHAR, and the operand will be nullable.
- Though not explicitly stated in the result definitions mentioned previously, the semantics imply that if *string* is a mixed single- and multi-byte character string, the result might contain fragments of multi-byte characters, depending upon the values of *start* and *length*. For example, the result could possibly begin with the second byte of a multi-byte character, or end with the first byte of a multi-byte character. The SUBSTR function does not detect such fragments, nor provide any special processing should they occur.

Examples

- *Example 1*: Assume that the host variable `NAME (VARCHAR(50))` has the value `'BLUE JAY'`:
  - The following statement returns the value `'BLUE'`:

    ```
    SUBSTR(:NAME,1,4)
    ```

  - The following statement returns the value `'JAY'`:

    ```
    SUBSTR(:NAME,6)
    ```

  - The following statement returns the value `'JA'`:

    ```
    SUBSTR(:NAME,6,2)
    ```

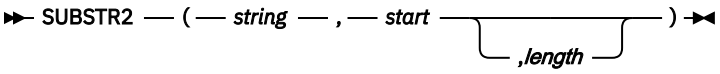
- *Example 2*: Select all rows from the PROJECT table for which the project name (PROJNAME) starts with the word 'OPERATION'.

  ```
  SELECT * FROM PROJECT
    WHERE SUBSTR(PROJNAME,1,10) = 'OPERATION '
  ```

  The space at the end of the constant is necessary to exclude words such as 'OPERATIONAL'.

SUBSTR2

The SUBSTR2 function returns a substring from a string. The resulting substring starts at a specified position in the string and continues for a specified or default length. The start and length arguments are expressed in 16-bit UTF-16 string units (CODEUNITS16).



The schema is SYSIBM.

string
An expression that specifies the string from which the resulting substring is derived. The expression must return a value that is a built-in character string, graphic string, numeric, or datetime data type. A character string cannot have the FOR BIT DATA attribute (SQLSTATE 428GC). If the value is not a

string data type, it is implicitly cast to VARCHAR before evaluating the function. A substring of *string* is zero or more contiguous bytes of *string*.

start
An expression that specifies the starting position in *string* for the beginning of the result substring in 16-bit UTF-16 string units. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

If *start* is positive, then the starting position is calculated from the beginning of the string. If *start* is greater than the length of *string* in 16-bit UTF-16 string units, then a zero length string is returned.

If *start* is negative, then the starting position is calculated from the end of the string by counting backwards. If the absolute value of *start* is greater than the length of *string* in 16-bit UTF-16 string units, then a zero length string is returned.

If *start* is 0, then a starting position of 1 is used.

length
An expression that specifies the length of the resulting substring in 16-bit UTF-16 string units. If *length* is specified, the expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

If the value for *length* is greater than the number of 16-bit UTF-16 string units from the starting position to the end of the string, then the length of the resulting substring is the length of the first argument in 16-bit UTF-16 string units minus the starting position plus one.

If the value for *length* is less than or equal to zero, the result is a zero length string.

The default value for *length* is the number of CODEUNITS16 from the position specified by *start* to the last byte of *string*.

If *string* is a CHAR or VARCHAR data type, the result of the function is a VARCHAR data type. If *string* is a CLOB, the result of the function is a CLOB. If *string* is a GRAPHIC or VARGRAPHIC data type, the result of the function is a VARGRAPHIC data type. If *string* is a DBCLOB, the result of the function is a DBCLOB. If the first argument is a host variable, the code page of the result is the section code page; otherwise, it is the code page of the first argument.

The length attribute of the result is the same as the length attribute of the first argument, unless the start or length arguments are specified as constants. When a constant is specified, the length attribute of the result is based on the first applicable row in the following table.

Table 96. Length Attribute of SUBSTR2 Result when Arguments Include Constants

String Argument	Start Argument[1]	Length Argument	Length Attribute of Result[2]
character type with length attribute A	any valid argument	constant value L<=0	0
character type with length attribute A	constant value S and |(S)|>A	not specified or any valid argument	0
character type with length attribute A	not a constant	constant value L>0	MIN(A, L×4)
character type with length attribute A	constant value S>0	not specified or not a constant	A-S+1
character type with length attribute A	constant value S<0	not specified or not a constant	MIN(A, |(S)×4|)
character type with length attribute A	constant value S>0	constant value L>0	MIN(A-S+1, L×4)

Table 96. Length Attribute of SUBSTR2 Result when Arguments Include Constants (continued)

String Argument	Start Argument[1]	Length Argument	Length Attribute of Result[2]
character type with length attribute A	constant value S<0	constant value L>0	MIN(A,|(S)×4|, L×4)
graphic type with length attribute A	any valid argument	constant value L<=0	0
graphic type with length attribute A	constant value S and |(S)|>A	not specified or any valid argument	0
graphic type with length attribute A	not a constant	constant value L>0	MIN(A, L)
graphic type with length attribute A	constant value S>0	not specified or not a constant	A-S+1
graphic type with length attribute A	constant value S<0	not specified or not a constant	|(S)|
graphic type with length attribute A	constant value S>0	constant value L>0	MIN(A-S+1, L)
graphic type with length attribute A	constant value S<0	constant value L>0	MIN(|(S)|, L)|

Notes:

[1] If a start argument value of 0 is specified, then use a value of 1 for S when referencing this table.

[2] The length attribute of the result for some of the character result types involves a constant that is multiplied by a factor of 4. This multiplier covers the worst case expansion derived from the following factors:

- Multiplying by 2 to switch from counting in 16-bit UTF-16 string units to counting in bytes used for the length attributes of a character data type.
- Multiplying by 2 again because a 2-byte character in UTF-16 can be represented by up to 4 bytes in a character string.

If any argument of the SUBSTR2 function can be null, the result can be null. If any argument is null, the result is the null value.

Notes

- In dynamic SQL, *string*, *start*, and *length* can be represented by a parameter marker. If an untyped parameter marker is used for string, the operand will be nullable and if the database supports graphic data types the data type of the operand will be VARGRAPHIC(16336). Otherwise, the data type will be VARCHAR(32672).
- If *string* is a mixed single-byte and multi-byte character string, the result might contain fragments of multi-byte characters depending on the values of *start* and *length*. For example, the result might begin with the third byte of a multi-byte character, or end with the first byte of a multi-byte character. The SUBSTR2 function detects these partial characters and replaces each byte of an incomplete character with a single blank character.
- SUBSTR2 is similar to the SUBSTR function, with the following exceptions:
  - SUBSTR2 supports a negative start value, which indicates that processing is to start from the end of the string.

- SUBSTR2 supports a *length* value that is greater than the calculated result length. In such cases, a shorter string is returned, rather than an error.
- SUBSTR2 returns a result data type of VARCHAR if the input data type is CHAR. VARGRAPHIC is the result data type returned if the input type is GRAPHIC.
- The length attribute of the result for SUBSTR2 is either the same as the length attribute of the first argument, or it is derived based on the start or length attributes, if either of these are constants.
- SUSTR2 returns a result with a length attribute that is the same as the length attribute of the first argument, unless the start or length arguments are specified as constants. When a constant is specified, the length attribute of the result is derived based on the start or length attributes (see the preceding table).

Examples

- *Example 1*: Given the following host variables:
  - `NAME (VARGRAPHIC(50)` with a value of 'Roméo Jürgen'
  - `SURNAME POS (INTEGER)` with a value of 7

```
SUBSTR2(:NAME, :SURNAME_POS)
```

  returns the value `Jürgen`

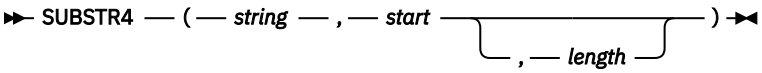
```
SUBSTR2(:NAME, :SURNAME_POS,2)
```

  returns the value `Jü`
- *Example 2*: Select all rows from the PROJECT table which end in 'ING'

```
SELECT * FROM PROJECT
   WHERE SUBSTR2(PROJNAME,-3) = 'ING'
```

SUBSTR4

The SUBSTR4 function returns a substring from a string. The resulting substring starts at a specified position in the string and continues for a specified or default length. The start and length arguments are expressed in 32-bit UTF-32 string units (CODEUNITS32).



The schema is SYSIBM.

string
An expression that specifies the string from which the resulting substring is derived. The expression must return a value that is a built-in character string, graphic string, numeric, or datetime data type. A character string cannot have the FOR BIT DATA attribute (SQLSTATE 428GC). If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. A substring of *string* is zero or more contiguous string units of *string*.

start
An expression that specifies the starting position in *string* for the beginning of the result substring in 32-bit UTF-32 string units. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

If *start* is positive, then the starting position is calculated from the beginning of the string. If *start* is greater than the length of *string* in 32-bit UTF-32 string units, then a zero length string is returned.

If *start* is negative, then the starting position is calculated from the end of the string by counting backwards. If the absolute value of *start* is greater than the length of *string* in 32-bit UTF-32 string units, then a zero length string is returned.

If *start* is 0, then a starting position of 1 is used.

length

An expression that specifies the length of the resulting substring in 32-bit UTF-32 string units. If *length* is specified, the expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

If the value for *length* is greater than the number of 32-bit UTF-32 string units from the starting position to the end of the string, then the length of the resulting substring is the length of the first argument in 32-bit UTF-32 string units minus the starting position plus one.

If the value for *length* is less than or equal to zero, the result is a zero length string.

The default value for *length* is the number of CODEUNITS32 from the position specified by *start* to the last string unit of *string*.

If *string* is a CHAR or VARCHAR data type, the result of the function is a VARCHAR data type. If *string* is a CLOB, the result of the function is a CLOB. If *string* is a GRAPHIC or VARGRAPHIC data type, the result of the function is a VARGRAPHIC data type. If *string* is a DBCLOB, the result of the function is a DBCLOB. If the first argument is a host variable, the code page of the result is the section code page; otherwise, it is the code page of the first argument.

The length attribute of the result is the same as the length attribute of the first argument, unless the start or length arguments are specified as constants. When a constant is specified, the length attribute of the result is based on the first applicable row in the following table. The string unit of the result is the same as the string unit of the first argument.

Table 97. Length attribute of SUBSTR4 result when arguments include constants

String Argument	Start Argument[1]	Length Argument	Length Attribute of Result[2]
character type with length attribute A	any valid argument	constant value L<=0	0
character type with length attribute A	constant value S and |(S)|>A	not specified or any valid argument	0
character type with length attribute A	not a constant	constant value L>0	MIN(A, L×4) if the string units of *string* is OCTETS MIN(A, L) if the string units of *string* is CODEUNITS32
character type with length attribute A	constant value S>0	not specified or not a constant	A-S+1
character type with length attribute A	constant value S<0	not specified or not a constant	MIN(A, |(S)×4|) if the string units of *string* is OCTETS MIN(A, |S|) if the string units of *string* is CODEUNITS32
character type with length attribute A	constant value S>0	constant value L>0	MIN(A-S+1, L×4) if the string units of *string* is OCTETS MIN(A-S+1, L) if the string units of *string* is CODEUNITS32

Table 97. Length attribute of SUBSTR4 result when arguments include constants (continued)

String Argument	Start Argument[1]	Length Argument	Length Attribute of Result[2]
character type with length attribute A	constant value S<0	constant value L>0	MIN(A,|(S)×4|, L×4) if the string units of *string* is OCTETS MIN(A, |S|, L) if the string units of *string* is CODEUNITS32
graphic type with length attribute A	any valid argument	constant value L<=0	0
graphic type with length attribute A	constant value S and |(S)|>A	not specified or any valid argument	0
graphic type with length attribute A	not a constant	constant value L>0	MIN(A, L*2) if the string units of *string* is double-bytes or CODEUNITS16 MIN(A, L) if the string units of *string* is CODEUNITS32
graphic type with length attribute A	constant value S>0	not specified or not a constant	A-S+1
graphic type with length attribute A	constant value S<0	not specified or not a constant	MIN(A, |(S) * 2|) if the string units of *string* is double-bytes or CODEUNITS16 MIN(A, |S|) if the string units of *string* is CODEUNITS32
graphic type with length attribute A	constant value S>0	constant value L>0	MIN(A-S+1, L*2) if the string units of *string* is double-bytes or CODEUNITS16 MIN(A-S+1, L) if the string units of *string* is CODEUNITS32
graphic type with length attribute A	constant value S<0	constant value L>0	MIN(A, |(S)*2|, L*2) if the string units of *string* is double-bytes or CODEUNITS16 MIN(A, |S|, L) if the string units of *string* is CODEUNITS32

Table 97. Length attribute of SUBSTR4 result when arguments include constants (continued)

String Argument	Start Argument[1]	Length Argument	Length Attribute of Result[2]

Notes:

[1] If a start argument value of 0 is specified, then use a value of 1 for S when referencing this table.

[2] The length attribute of the result for some of the character result types involves a constant that is multiplied by a factor of 4. This multiplier covers the worst case expansion derived from multiplying by 4 to switch from counting in 32-bit UTF-32 string units to counting in bytes used for the length attributes of a character data type with string units of OCTETS.
The length attribute of the result for some of the graphic result types involves a constant that is multiplied by a factor of 2. This multiplier covers the worst case expansion derived from multiplying by 2 because a 4-byte character in UTF-32 could be represented by up to 2 double-byte characters in a graphic string.

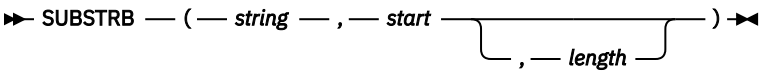
If any argument of the SUBSTR4 function can be null, the result can be null. If any argument is null, the result is the null value.

Notes

- If *string* contains combining characters, the result might contain base characters without their combining characters or combining characters without their base characters depending on the values of start and length.
- SUBSTR4 is similar to the SUBSTR function, with the following exceptions:
  - SUBSTR4 supports a negative start value, which indicates that processing is to start from the end of the string.
  - SUBSTR4 supports a *length* value that is greater than the calculated result length. In such cases, a shorter string is returned, rather than an error.
  - SUBSTR4 returns a result data type of VARCHAR if the input data type is CHAR. VARGRAPHIC is the result data type returned if the input type is GRAPHIC.
  - The length attribute of the result for SUBSTR4 is either the same as the length attribute of the first argument, or it is derived based on the start or length attributes, if either of these are constants.
  - SUBSTR4 returns a result with a length attribute that is the same as the length attribute of the first argument, unless the start or length arguments are specified as constants. When a constant is specified, the length attribute of the result is derived based on the start or length attributes (see the preceding table).

SUBSTRB

The SUBSTRB function returns a substring of a string, beginning at a specified position in the string. Lengths are calculated in bytes.



The schema is SYSIBM.

The SUBSTRB function is available starting with version 9.7 Fix Pack 1.

string
An expression that specifies the string from which the result is derived.

The expression must return a value that is a built-in character string, numeric value, Boolean value, or datetime value. If the value is not a character string, it is implicitly cast to VARCHAR before the

function is evaluated. In a Unicode database, if the value is a graphic data type, it is implicitly cast to a character string data type before the function is evaluated. Any number (zero or more) contiguous bytes of this expression constitute a substring of this expression.

start
An expression that specifies the start position in *string* of the beginning of the result substring. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

If *start* is positive, then the start position is calculated from the beginning of the string. If *start* is greater than the length of *string*, then a zero length string is returned.

If *start* is negative, then the start position is calculated from the end of the string and by counting backwards. If the absolute value of *start* is greater than the length of *string*, then a zero length string is returned.

If *start* is 0, then a start position of 1 is used.

length
An expression that specifies the length of the result in bytes. If specified, the expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

If the length is:

- Greater than the number of bytes from the start position to the end of the string, the result length is the length of the first argument minus the start position plus one
- Less than or equal to zero, the result of SUBSTRB is a zero length string

The default length is the number of bytes from the position specified by *start* to the last byte of *string*.

Result

If *string* is a CHAR or VARCHAR data type, the result of the function is a VARCHAR data type. If *string* is a CLOB, the result of the function is a CLOB. If *string* is a BINARY or VARBINARY data type, the result of the function is a VARBINARY data type. If *string* is a BLOB, the result of the function is a BLOB. If the first argument is a host variable that is not a binary string and not a FOR BIT DATA character string, the code page of the result is the section code page; otherwise, it is the code page of the first argument.

The length attribute of the result is the same as the length attribute of the first argument unless both *start* and *length* arguments are specified and defined as constants. In this case, the length attribute of the result is determined as follows:

- If *length* is a constant which is less than or equal to zero, the length attribute of the result is zero.
- If *start* is not a constant, but *length* is a constant, the length attribute of the result is the minimum of the length attribute of the first argument and *length*.
- If *start* is a constant, but *length* is not a constant or not specified, the length attribute of the result is the length attribute of the first argument minus the start position, plus one.
- If *start* and *length* are constants, the length attribute of the result is the minimum of the following values:
  - *length*
  - The length attribute of the first argument minus the start position plus one

If any argument of the SUBSTRB function can be null, the result can be null; if any argument is null, the result is the null value.

Notes

- In dynamic SQL, *string*, *start*, and *length* can be represented by a parameter marker. If a parameter marker is used for *string*, the data type of the operand will be VARCHAR, and the operand will be nullable.
- Though not explicitly stated in the previously mentioned result definitions, the semantics imply that if *string* is a mixed single-byte and multi-byte character string, the result might contain fragments of multi-byte characters, depending on the values of *start* and *length*. For example, the result could possibly begin with the second byte of a multi-byte character, or end with the first byte of a multi-byte character. The SUBSTRB function will detect these partial characters and will replace each byte of an incomplete character with a single blank character.
- SUBSTRB is similar to the existing SUBSTR function, with the following exceptions:
  - SUBSTRB supports a negative *start* value, which indicates the processing should start from the end of the string.
  - SUBSTRB allows *length* to be greater than the calculated result length. In this case, a shorter string will be returned, rather than returning an error.
  - Graphic input data is not natively supported for the first argument of SUBSTRB. In a Unicode database, graphic data is supported, but it is first converted to character data before evaluating the function, and lengths are calculated in bytes.
  - The result data type of SUBSTRB is VARCHAR if the input data type is CHAR.
  - The length attribute of the result for SUBSTRB is either the same as the length attribute of the first argument, or it is derived based on the *start* or *length* attributes, if either of these are constants.

Examples

- *Example 1*: Assume the host variable NAME (VARCHAR(50)) has a value of 'BLUE JAY' and the host variable SURNAME_POS (INTEGER) has a value of 6.

```
SUBSTRB(:NAME, :SURNAME_POS)
```

  Returns the value 'JAY'.

```
SUBSTRB(:NAME, :SURNAME_POS,1)
```

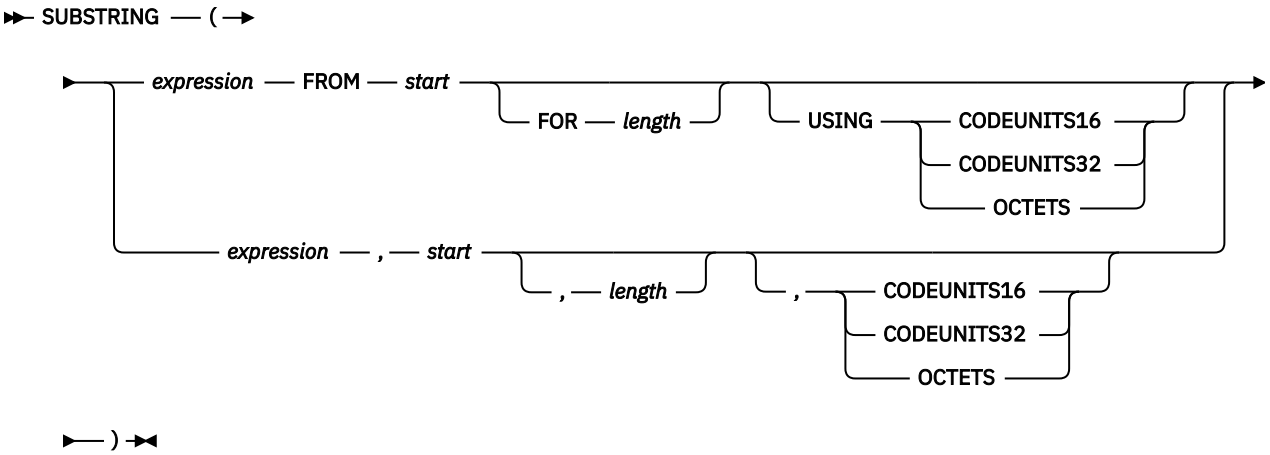
  Returns the value 'J'.

- *Example 2*: Select all rows from the PROJECT table which end in 'ING'.

```
SELECT * FROM PROJECT
   WHERE SUBSTRB(PROJNAME,-3) = 'ING'
```

SUBSTRING

The SUBSTRING function returns a substring of a string.



The schema is SYSIBM.

expression
An expression that specifies the string from which the result is derived. The expression must return a built-in character string, numeric value, Boolean value, or datetime value. If the value is not a character string, it is implicitly cast to VARCHAR before the function is evaluated.

A substring of the input expression comprises zero or more contiguous string units of the input expression.

start
An expression that specifies the position, relative to the beginning of the input expression, from which the substring is to be calculated. For example:

- Position 1 is the first string unit of the input expression. The statement `SUBSTRING('abc',1,2)` returns `'ab'`.
- Position 2 is one position to the right of position 1. The statement `SUBSTRING('abc',2,2)` returns `'bc'`.
- Position 0 is one position to the left of position 1. The statement `SUBSTRING('abc',0,2)` returns `'a'`.
- Position -1 is two positions to the left of position 1. The statement `SUBSTRING('abc',-1,2)` returns a zero-length string.

The expression must return a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC value. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

The start value can be positive, negative, or zero. If OCTETS is specified and the input expression contains graphic data, the start value must be odd (SQLSTATE 428GC).

length
An expression that specifies the length of the result. The expression must return a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC value. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

If the input expression is:

- A fixed-length string, the default length is `CHARACTER_LENGTH(`*expression* `USING` *string-unit*`) - ` *start* ` + 1`. This is the number of string units (CODEUNITS16, CODEUNITS32, or OCTETS) from the start position to the final position of the input expression.
- A varying-length string, the default length is zero or `CHARACTER_LENGTH(`*expression* `USING` *string-unit*`) - ` *start* ` + 1`, whichever is greater.

If the specified length is zero, the result is the empty string.

If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value must be greater than or equal to zero. If a value greater than *n* is specified, where *n* is the (length attribute of *expression*) - *start* + 1, then *n* is used as the length of the resulting substring. The value is expressed in the string units that are explicitly specified. If OCTETS is specified, and if the input expression contains graphic data, the length must be an even number (SQLSTATE 428GC).

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string units of *start* and *length*. CODEUNITS16 specifies that *start* and *length* are to be expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *start* and *length* are to be expressed in 32-bit UTF-32 code units. OCTETS specifies that *start* and *length* are to be expressed in bytes.

If string units are specified as CODEUNITS16 or CODEUNITS32, and *expression* is a binary string or a FOR BIT DATA string, an error is returned (SQLSTATE 428GC).

If a string units argument is not specified and *expression* is a character string that is not FOR BIT DATA or is a graphic string, the default is CODEUNITS32. Otherwise, the default is OCTETS.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

When the SUBSTRING function is invoked using OCTETS, and the *source-string* is encoded in a code page that requires more than one byte per code point (mixed or MBCS), the SUBSTRING operation might split a multi-byte code point and the resulting substring might begin or end with a partial code point. If this occurs, the function replaces the bytes of leading or trailing partial code points with blanks in a way that does not change the byte length of the result. (See a related example in the Examples section.)

Result

The data type of the result depends on the data type of the first argument, as shown in the following table.

Table 98. Data type of the result of SUBSTRING

Data type of the first argument	**Data type of the result**
CHAR or VARCHAR	VARCHAR
CLOB	CLOB
GRAPHIC or VARGRAPHIC	VARGRAPHIC
DBCLOB	DBCLOB
BINARY or VARBINARY	VARBINARY
BLOB	BLOB

The length attribute of the result is equal to the length attribute of the input expression. If any argument of the function can be null, the result can be null; if any argument is null, the result is the null value. The result is not padded with any character. If the input expression has an actual length of 0, the result also has an actual length of 0.

Notes

- The length attribute of the result is equal to the length attribute of the input expression. This behavior is different from the behavior of the SUBSTR function, where the length attribute is derived from the *start* and the *length* arguments of the function.

Examples

- *Example 1*: FIRSTNAME is a VARCHAR(12) column in table T1. One of its values is the 6-character string 'Jürgen'. When FIRSTNAME has this value:

```
Function ...                          Returns ...
------------------------------------  ------------------------------------
SUBSTRING(FIRSTNAME,1,2,CODEUNITS32)  'Jü' -- x'4AC3BC'
SUBSTRING(FIRSTNAME,1,2,CODEUNITS16)  'Jü' -- x'4AC3BC'
SUBSTRING(FIRSTNAME,1,2,OCTETS)       'J ' -- x'4A20' (a truncated string)
SUBSTRING(FIRSTNAME,8,CODEUNITS16)    a zero-length string
SUBSTRING(FIRSTNAME,8,4,OCTETS)       a zero-length string
SUBSTRING(FIRSTNAME,0,2,CODEUNITS32)  'J' -- x'4AC3BC'
```

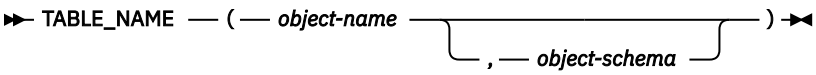
- *Example 2*: The following example illustrates how SUBSTRING replaces the bytes of leading or trailing partial multi-byte code points with blanks when the string length unit is OCTETS. Assume that UTF8_VAR contains the UTF-8 representation of the Unicode string '&N~AB', where '&' is the musical symbol G clef and '~' is the combining tilde character.

```
   SUBSTRING(UTF8_VAR, 2, 5, OCTETS)
```

  Three blank bytes precede the 'N', and one blank byte follows the 'N'.

TABLE_NAME

The TABLE_NAME function returns an unqualified name of the object found after any alias chains have been resolved.



The schema is SYSIBM.

The specified *object-name* (and *object-schema*) are used as the starting point of the resolution. If the starting point does not refer to an alias, the unqualified name of the starting point is returned. The resulting name may be of a table, view, or undefined object. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

object-name
A character expression representing the unqualified name (usually of an existing alias) to be resolved. *object-name* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.

object-schema
A character expression representing the schema used to qualify the supplied *object-name* value before resolution. *object-schema* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.

If *object-schema* is not supplied, the default schema is used for the qualifier.

The data type of the result of the function is VARCHAR(128 OCTETS). If *object-name* can be null, the result can be null; if *object-name* is null, the result is the null value. If *object-schema* is the null value, the default schema name is used. The result is the character string representing an unqualified name. The result name could represent one of the following objects:

table
The value for *object-name* was either a table name (the input value is returned) or an alias name that resolved to the table whose name is returned.

view
The value for *object-name* was either a view name (the input value is returned) or an alias name that resolved to the view whose name is returned.

undefined object

The value for *object-name* was either an undefined object (the input value is returned) or an alias name that resolved to the undefined object whose name is returned.

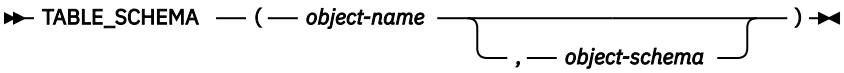
Therefore, if a non-null value is given to this function, a value is always returned, even if no object with the result name exists.

Notes

- To improve performance in partitioned database configurations by avoiding the unnecessary communication that occurs between the coordinator partition and catalog partition when using the TABLE_SCHEMA and TABLE_NAME scalar functions, the BASE_TABLE table function can be used instead.

TABLE_SCHEMA

The TABLE_SCHEMA function returns the schema name of the object found after any alias chains have been resolved.



The schema is SYSIBM.

The specified *object-name* (and *object-schema*) are used as the starting point of the resolution. If the starting point does not refer to an alias, the schema name of the starting point is returned. The resulting schema name may be of a table, view, or undefined object. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

object-name
A character expression representing the unqualified name (usually of an existing alias) to be resolved. *object-name* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.

object-schema
A character expression representing the schema used to qualify the supplied *object-name* value before resolution. *object-schema* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.

If *object-schema* is not supplied, the default schema is used for the qualifier.

The data type of the result of the function is VARCHAR(128 OCTETS). If *object-name* can be null, the result can be null; if *object-name* is null, the result is the null value. If *object-schema* is the null value, the default schema name is used. The result is the character string representing a schema name. The result schema could represent the schema name for one of the following objects:

table
The value for *object-name* was either a table name (the input or default value of *object-schema* is returned) or an alias name that resolved to a table for which the schema name is returned.

view
The value for *object-name* was either a view name (the input or default value of *object-schema* is returned) or an alias name that resolved to a view for which the schema name is returned.

undefined object
The value for *object-name* was either an undefined object (the input or default value of *object-schema* is returned) or an alias name that resolved to an undefined object for which the schema name is returned.

Therefore, if a non-null *object-name* value is given to this function, a value is always returned, even if the object name with the result schema name does not exist. For example, `TABLE_SCHEMA('DEPT', 'PEOPLE')` returns 'PEOPLE ' if the catalog entry is not found.

Notes

- To improve performance in partitioned database configurations by avoiding the unnecessary communication that occurs between the coordinator partition and catalog partition when using the TABLE_SCHEMA and TABLE_NAME scalar functions, the BASE_TABLE table function can be used instead.

Examples

- *Example 1*: PBIRD tries to select the statistics for a given table from SYSCAT.TABLES using an alias PBIRD.A1 defined on the table HEDGES.T1.

```
SELECT NPAGES, CARD FROM SYSCAT.TABLES
  WHERE TABNAME = TABLE_NAME ('A1')
  AND TABSCHEMA = TABLE_SCHEMA ('A1')
```

  The requested statistics for HEDGES.T1 are retrieved from the catalog.

- *Example 2*: Select the statistics for an object called HEDGES.X1 from SYSCAT.TABLES using HEDGES.X1. Use TABLE_NAME and TABLE_SCHEMA since it is not known whether HEDGES.X1 is an alias or a table.

```
SELECT NPAGES, CARD FROM SYSCAT.TABLES
  WHERE TABNAME = TABLE_NAME ('X1','HEDGES')
  AND TABSCHEMA = TABLE_SCHEMA ('X1','HEDGES')
```

  Assuming that HEDGES.X1 is a table, the requested statistics for HEDGES.X1 are retrieved from the catalog.

- *Example 3*: Select the statistics for a given table from SYSCAT.TABLES using an alias PBIRD.A2 defined on HEDGES.T2 where HEDGES.T2 does not exist.

```
SELECT NPAGES, CARD FROM SYSCAT.TABLES
  WHERE TABNAME = TABLE_NAME ('A2','PBIRD')
  AND TABSCHEMA = TABLE_SCHEMA ('A2',PBIRD')
```

  The statement returns 0 records as no matching entry is found in SYSCAT.TABLES where TABNAME = 'T2' and TABSCHEMA = 'HEDGES'.

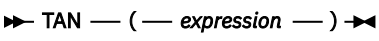
- *Example 4*: Select the qualified name of each entry in SYSCAT.TABLES along with the final referenced name for any alias entry.

```
SELECT TABSCHEMA AS SCHEMA,  TABNAME AS NAME,
  TABLE_SCHEMA (BASE_TABNAME, BASE_TABSCHEMA) AS REAL_SCHEMA,
  TABLE_NAME (BASE_TABNAME, BASE_TABSCHEMA) AS REAL_NAME
  FROM SYSCAT.TABLES
```

  The statement returns the qualified name for each object in the catalog and the final referenced name (after alias has been resolved) for any alias entries. For all non-alias entries, BASE_TABNAME and BASE_TABSCHEMA are null so the REAL_SCHEMA and REAL_NAME columns will contain nulls.

TAN

Returns the tangent of the argument, where the argument is an angle expressed in radians.



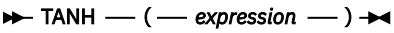
The schema is SYSIBM. (The SYSFUN version of the TAN function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

TANH

Returns the hyperbolic tangent of the argument, where the argument is an angle expressed in radians.



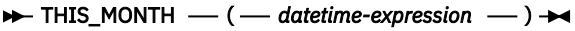
The schema is SYSIBM.

expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

THIS_MONTH

The THIS_MONTH function returns the first day of the month in the specified date.



The schema is SYSIBM.

datetime-expression
An expression that specifies a date for which first day of the month is to be returned. The expression must return a value that is a DATE, a TIMESTAMP, a CHAR, or a VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported using implicit casting. If *datetime-expression* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string accepted by the TIMESTAMP scalar function.

The result of the function is DATE. If the argument can be null, the result can be null. If the argument is null, the result is the null value.

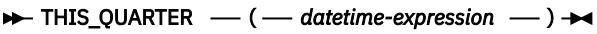
Example

Returns the date value of the first day of the month specified by the input.

```
values this_month('2007-02-18')
      Result: 2007-02-01
```

THIS_QUARTER

The THIS_QUARTER function returns the first day of the quarter that contains the specified date.



The schema is SYSIBM.

datetime-expression
An expression that specifies a date for which first day of the quarter is to be returned. The expression must return a value that is a DATE, a TIMESTAMP, a CHAR, or a VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported using implicit casting. If *datetime-expression* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string accepted by the TIMESTAMP scalar function.

The result of the function is DATE. If the argument can be null, the result can be null. If the argument is null, the result is the null value.

Example

Returns the date value of the first day of the quarter of the date specified by the input.

```
values this_quarter('2007-05-18')
      Result: 2007-04-01
```

THIS_WEEK

The THIS_WEEK function returns the first day of the week that contains the specified date. Sunday is considered the first day of that week.

The schema is SYSIBM.

datetime-expression
An expression that specifies a date for which first day of the week is to be returned. The expression must return a value that is a DATE, a TIMESTAMP, a CHAR, or a VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported using implicit casting. If *datetime-expression* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string accepted by the TIMESTAMP scalar function.

This function returns a value of data type DATE. If the argument can be null, the result can be null. If the argument is null, the result is the null value.

Example

```
values this_week ('1996-02-29')
      Result: 1996-02-25
```

THIS_YEAR

The THIS_YEAR function returns the first day of the year in the specified date.

The schema is SYSIBM.

datetime-expression
An expression that specifies a date for which first day of the year is to be returned. The expression must return a value that is a DATE, a TIMESTAMP, a CHAR, or a VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported using implicit casting. If *datetime-expression* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string accepted by the TIMESTAMP scalar function.

The result of the function is DATE. If the argument can be null, the result can be null. If the argument is null, the result is the null value.

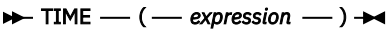
Example

The following example returns the date value of the first day of the year of the date specified by the input.

```
values this_year('2007-02-18')
      Result: 2007-01-01
```

TIME

The TIME function returns a time from a value.



The schema is SYSIBM.

expression
An expression that returns a value of one of the following built-in data types: DATE, TIME, TIMESTAMP, or a valid character string representation of a date, time, or timestamp that is not a CLOB. In a Unicode database, if an expression returns a value of a graphic string data type, the value is first converted to a character string before the function is executed.

The result of the function is a TIME. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a DATE or string representation of a date:
  - The result is midnight.
- If the argument is a TIME:
  - The result is that time.
- If the argument is a TIMESTAMP:
  - The result is the time part of the timestamp.
- If the argument is a string representation of time or timestamp:
  - The result is the time represented by the string.

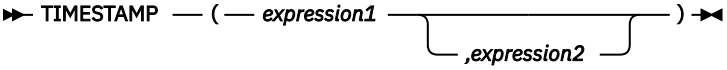
Example

Select all notes from the IN_TRAY sample table that were received at least one hour later in the day (any day) than the current time.

```
SELECT * FROM IN_TRAY
  WHERE TIME(RECEIVED) >= CURRENT TIME + 1 HOUR
```

TIMESTAMP

The TIMESTAMP function returns a timestamp from a value or a pair of values.



The schema is SYSIBM.

Only Unicode databases support an argument that is a graphic string representation of a date, a time, or a timestamp. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

expression1* and *expression2

The rules for the arguments depend on whether **expression2** is specified and the data type of **expression2**.

- If only one argument is specified it must be an expression that returns a value of one of the following built-in data types: a DATE, a TIMESTAMP, or a character string that is not a CLOB. If **expression1** is a character string, it must be one of the following:

  - A valid character string representation of a date or a timestamp. For the valid formats of string representations of date or timestamp values, see "String representations of datetime values" in Datetime values.
  - A character string with an actual length of 13 that is assumed to be a result from the GENERATE_UNIQUE function.
  - A string of length 14 that is a string of digits that represents a valid date and time in the form *yyyyxxddhhmmss*, where *yyyy* is the year, *xx* is the month, *dd* is the day, *hh* is the hour, *mm* is the minute, and *ss* is the seconds.
- If **expression1** and **expression2** are specified:
  - If the data type of **expression2** is not an integer:
    - **expression1** must be a DATE or a valid string representation of a date and the **expression2** must be a TIME or a valid string representation of a time.
  - If the data type of **expression2** is an integer:
    - **expression1** must be a DATE, TIMESTAMP, or a valid string representation of a timestamp or date. **expression2** must be an integer constant in the range 0 to 12 representing the timestamp precision.

The result of the function is a TIMESTAMP.

The timestamp precision and other rules depend on whether the second argument is specified:

- If both arguments are specified and the second argument is not an integer:
  - The result is a TIMESTAMP(6) with the date specified by the first argument and the time specified by the second argument. The fractional seconds part of the timestamp is zero.
- If both arguments are specified and the second argument is an integer:
  - The result is a TIMESTAMP with the precision specified in the second argument.
- If only one argument is specified and it is a TIMESTAMP(*p*):
  - The result is that TIMESTAMP(*p*).
- If only one argument is specified and it is a DATE:
  - The result is that date with an assumed time of midnight cast to TIMESTAMP(0).
- If only one argument is specified and it is a string:
  - The result is the TIMESTAMP(6) value represented by that string extended as described earlier with any missing time information. If the argument is a string of length 14, the TIMESTAMP has a fractional seconds part of zero.

If the arguments include only date information, the time information in the result value is all zeros. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Examples

- *Example 1*: Assume the column START_DATE (whose data type is DATE) has a value equivalent to 1988-12-25, and the column START_TIME (whose data type is TIME) has a value equivalent to 17.12.30.

```
TIMESTAMP(START_DATE, START_TIME)
```

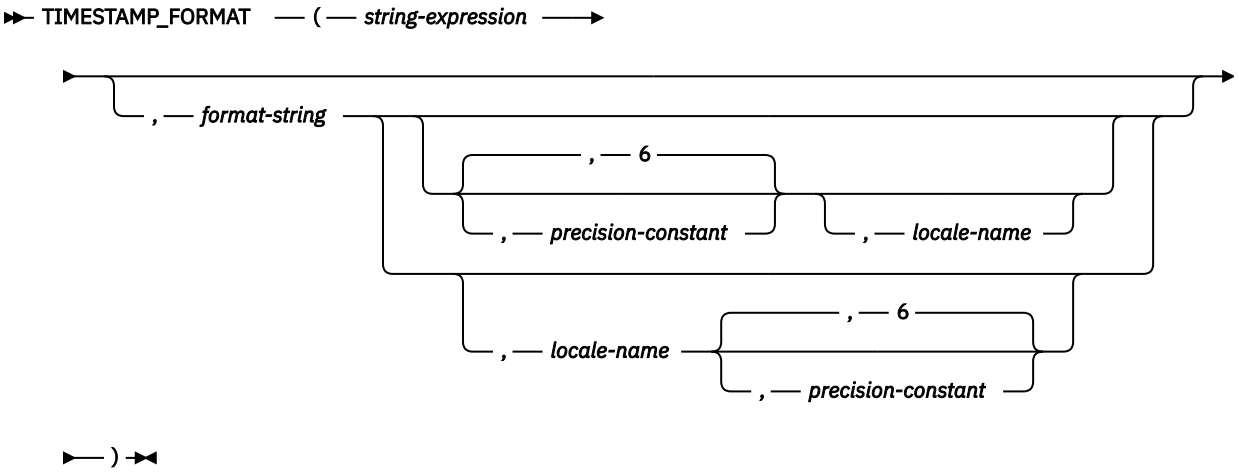
  Returns the value '1988-12-25-17.12.30.000000'.
- *Example 2*: Convert a timestamp string with 7 digits of fractional seconds to a TIMESTAMP(9) value.

```
TIMESTAMP('2007-09-24-15.53.37.2162474',9)
```

  Returns the value '2007-09-24-15.53.37.216247400'.

TIMESTAMP_FORMAT

The TIMESTAMP_FORMAT function returns a timestamp that is based on the interpretation of the input string using the specified format.



The schema is SYSIBM.

string-expression
The expression must return a value that is a built-in CHAR or VARCHAR data type. In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function. The *string-expression* must contain the components of a timestamp that correspond to the format specified by *format-string*.

format-string
The expression must return a value that is a built-in CHAR or VARCHAR data type. In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function. The actual length must not be greater than 255 bytes (SQLSTATE 22007). The value is a template for how *string-expression* is interpreted and then converted to a timestamp value.

A valid *format-string* must contain at least one format element, must not contain multiple specifications for any component of a timestamp, and can contain any combination of the format elements, unless otherwise noted in Table 99 on page 528 (SQLSTATE 22007). For example, *format-string* cannot contain both YY and YYYY, because they are both used to interpret the year component of *string-expression*. Refer to the table to determine which format elements cannot be specified together. Two format elements can optionally be separated by one or more of the following separator characters:

- minus sign (-)
- period (.)
- slash (/)
- comma (,)
- apostrophe (')
- semi-colon (;)
- colon (:)
- blank ( )

Separator characters can also be specified at the start or end of *format-string*. These separator characters can be used in any combination in the format string, for example 'YYYY/MM-DD HH:MM.SS'. Separator characters specified in a *string-expression* are used to separate components and are not required to match the separator characters specified in the *format-string*.

Table 99. Format elements for the TIMESTAMP_FORMAT function

Format element	Related components of a timestamp	Description
AM or PM	hour	Meridian indicator (morning or evening) without periods. This format element is dependent on *locale-name*, if specified. Otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
A.M. or P.M.	hour	Meridian indicator (morning or evening) with periods. This format element uses the exact strings "A.M." or "P.M." and is independent of the locale name in effect.
DAY, Day, or day	none	Name of the day in uppercase, titlecase, or lowercase format. The language used is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
DY, Dy, or dy	none	Abbreviated name of the day in uppercase, titlecase, or lowercase format. The language used is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
D	none	Day of the week (1-7). The first day of the week is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
DD	day	Day of month (01-31).
DDD	month, day	Day of year (001-366).

Table 99. Format elements for the TIMESTAMP_FORMAT function (continued)

Format element	Related components of a timestamp	Description
FF or FF*n*	fractional seconds	Fractional seconds (0-999999999999). The number *n* is used to specify the number of digits expected in the *string-expression*. Valid values for *n* are 1-12 with no leading zeros. Specifying FF is equivalent to specifying FF6. When the component in *string-expression* corresponding to the FF format element is followed by a separator character or is the last component, the number of digits for the fractional seconds can be less than what is specified by the format element. In this case zero digits are padded onto the right of the specified digits.
HH	hour	HH behaves the same as HH12.
HH12	hour	Hour of the day (01-12) in 12-hour format. AM is the default meridian indicator.
HH24	hour	Hour of the day (00-24) in 24-hour format.
J	year, month, and day	Julian day (number of days since January 1, 4713 BC).
MI	minute	Minute (00-59).
MM	month	Month (01-12).
MONTH, Month, or month	month	Name of the month in uppercase, titlecase, or lowercase format. The language used is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
MON, Mon, or mon	month	Abbreviated name of the month in uppercase, titlecase, or lowercase format. The language used is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
NNNNNN	microseconds	Microseconds (000000-999999). Same as FF6.

Table 99. Format elements for the TIMESTAMP_FORMAT function (continued)		
Format element	**Related components of a timestamp**	**Description**
RR	year	Last two digits of the adjusted year (00-99).
RRRR	year	4-digit adjusted year (0000-9999).
SS	seconds	Seconds (00-59).
SSSSS	hours, minutes, and seconds	Seconds since previous midnight (00000-86400).
Y	year	Last digit of the year (0-9). First three digits of the current year are used to determine the full 4-digit year.
YY	year	Last two digits of the year (00-99). First two digits of the current year are used to determine the full 4-digit year.
YYY	year	Last three digits of the year (000-999). First digit of the current year is used to determine the full 4-digit year.
YYYY	year	4-digit year (0000-9999).

Note: The format elements in Table 99 on page 528 are not case sensitive, except for the following:

- AM, PM
- A.M., P.M.
- DAY, Day, day
- DY, Dy, dy
- D
- MONTH, Month, month
- MON, Mon, mon

The DAY, Day, day, DY, Dy, dy, and D format elements do not contribute to any components of the resulting timestamp. However, a specified value for any of these format elements must be correct for the combination of the year, month, and day components of the resulting timestamp (SQLSTATE 22007). For example, assuming a value of 'en_US' is used for *locale-name*, a value of 'Monday 2008-10-06' for *string-expression* is valid for a value of 'Day YYYY-MM-DD'. However, value of 'Tuesday 2008-10-06' for *string-expression* would result in error for the same *format-string*.

The RR and RRRR format elements can be used to alter how a specification for a year is to be interpreted by adjusting the value to produce a 2-digit value or a 4-digit value depending on the leftmost two digits of the current year according to the following table.

Last two digits of the current year	**Two-digit year in *string-expression***	**First two digits of the year component of timestamp**
00-50	00-49	First two digits of the current year
51-99	00-49	First two digits of the current year + 1

Last two digits of the current year	Two-digit year in *string-expression*	First two digits of the year component of timestamp
00-50	50-99	First two digits of the current year - 1
51-99	50-99	First two digits of the current year

For example, if the current year is 2007, '86' with format 'RR' means 1986, but if the current year is 2052, it means 2086.

The following defaults are used when a *format-string* does not include a format element for one of the following components of a timestamp:

Timestamp component	Default
year	current year, as 4 digits
month	current month, as 2 digits
day	01 (first day of the month)
hour	00
minute	00
second	00
fractional seconds	a number of zeros matching the timestamp precision of the result

Leading zeros can be specified for any component of the timestamp value (that is, month, day, hour, minutes, seconds) that does not have the maximum number of significant digits for the corresponding format element in the *format-string*.

A substring of the *string-expression* representing a component of a timestamp (such as year, month, day, hour, minutes, seconds) can include less than the maximum number of digits for that component of the timestamp indicated by the corresponding format element. Any missing digits default to zero. For example, with a *format-string* of 'YYYY-MM-DD HH24:MI:SS', an input value of '999-3-9 5:7:2' would produce the same result as '0999-03-09 05:07:02'.

If *format-string* is not specified, *string-expression* will be interpreted using a default format based on the value of the special register CURRENT LOCALE LC_TIME.

precision-constant
An integer constant that specifies the timestamp precision of the result. The value must be in the range 0 to 12. If not specified, the timestamp precision defaults to 6.

locale-name
A character constant that specifies the locale used for the following format elements:

- AM, PM
- DAY, Day, day
- DY, Dy, dy
- D
- MONTH, Month, month
- MON, Mon, mon

The value of *locale-name* is not case sensitive and must be a valid locale (SQLSTATE 42815).

For information about valid locales and their naming, see "Locale names for SQL and XQuery" in the *Globalization Guide* .

If *locale-name* is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result of the function is a TIMESTAMP with a precision based on *precision-constant*. If either of the first two arguments can be null, the result can be null; if either of the first two arguments is null, the result is the null value.

Notes

- ***Julian and Gregorian calendar***: The transition from the Julian calendar to the Gregorian calendar on 15 October 1582 is taken into account by this function.
- ***Determinism:*** TIMESTAMP_FORMAT is a deterministic function. However, the following invocations of the function depend on the value of either the special register CURRENT LOCALE LC_TIME or CURRENT TIMESTAMP.
  - When *format-string* is not explicitly specified, or when *locale-name* is not explicitly specified and one of the following is true:
    - *format-string* is not a constant
    - *format-string* is a constant and includes format elements that are locale sensitive
    - *format-string* is a constant and does not include a format element that fully defines the year (that is, J or YYYY) and so uses the value of the current year
    - *format-string* is a constant and does not include a format element that fully defines the month (for example, J, MM, MONTH, or MON) and so uses the value of the current month

  These invocations that depend on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621, 428EC, or 429BX).
- ***Syntax alternatives:*** TO_DATE is a synonym for TIMESTAMP_FORMAT. TO_TIMESTAMP is a similar function, and the only difference is that the default value for *precision-constant* is 12.

Examples

- *Example 1*: Insert a row into the IN_TRAY table with a receiving timestamp that is equal to one second before the beginning of the year 2000 (December 31, 1999 at 23:59:59).

```
INSERT INTO IN_TRAY (RECEIVED)
  VALUES (TIMESTAMP_FORMAT('1999-12-31 23:59:59',
    'YYYY-MM-DD HH24:MI:SS'))
```

- *Example 2*: An application receives strings of date information into a variable called INDATEVAR. This value is not strictly formatted and might include two or four digits for years, and one or two digits for months and days. Date components might be separated with minus sign (-) or slash (/) characters and are expected to be in day, month, and year order. Time information consists of hours (in 24-hour format) and minutes, and is usually separated by a colon. Sample values include '15/12/98 13:48' and '9-3-2004 8:02'. Insert such values into the IN_TRAY table.

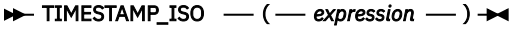
```
INSERT INTO IN_TRAY (RECEIVED)
  VALUES (TIMESTAMP_FORMAT(:INDATEVAR,
    'DD/MM/RRRR HH24:MI'))
```

  The use of RRRR in the format allows for 2- and 4-digit year values and assigns missing first two digits based on the current year. If YYYY were used, input values with a 2-digit year would have leading zeros. The slash separator also allows the minus sign character. Assuming a current year of 2007, resulting timestamps from the sample values are:

```
'15/12/98 13:48' --> 1998-12-15-13.48.00.000000
'9-3-2004 8:02'     --> 2004-03-09-08.02.00.000000
```

TIMESTAMP_ISO

Returns a timestamp value based on a date, time, or timestamp argument.



The schema is SYSFUN.

expression
An expression that returns a value of one of the following built-in data types: CHAR, VARCHAR, DATE, TIME, or TIMESTAMP data type. In a Unicode database, if a supplied argument has a GRAPHIC or VARGRAPHIC data type, it is first converted to a character string before evaluating the function. A string expression must return a valid character string representation of a date or timestamp.

If the argument is a date value, TIMESTAMP_ISO inserts zero for all the time elements. If the argument is a time value, TIMESTAMP_ISO inserts the value of the CURRENT DATE special register for the date elements, and zero for the fractional seconds element. The result of the function is a TIMESTAMP(6). The result can be null; if the argument is null, the result is the null value.

The TIMESTAMP_ISO function is generally defined as deterministic. If the first argument has the TIME data type, then the function is not deterministic because the CURRENT DATE is used for the date portion of the timestamp value.

The result of the function is a TIMESTAMP. The result can be null; if the argument is null, the result is the null value.

TIMESTAMPDIFF

Returns an estimated number of intervals of the type defined by the first argument, based on the difference between two timestamps.

The schema is SYSIBM. The SYSFUN version of the TIMESTAMPDIFF function continues to be available.

numeric-expression
An expression that returns a value of built-in INTEGER or SMALLINT data type. Valid values represent an interval as defined in the following table.

Table 100. Valid interval values

Value	Interval
1	Microseconds
2	Seconds
4	Minutes
8	Hours
16	Days
32	Weeks
64	Months
128	Quarters
256	Years

string-expression
An expression that returns a value of built-in CHAR or VARCHAR data type. The value is expected to be the result of subtracting two timestamps and converting the result to CHAR. If the value is not a CHAR or VARCHAR data type, it is implicitly cast to VARCHAR before evaluating the function. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed

If a positive or negative sign is present, it is the first character of the string. The following table describes the elements of the character string duration.

Table 101. TIMESTAMPDIFF string elements		
String Elements	**Valid Values**	**Character position from the decimal point (negative is left)**
Years	1-9998 or blank	-14 to -11
Months	0-11 or blank	-10 to -9
Days	0-30 or blank	-8 to -7
Hours	0-24 or blank	-6 to -5
Minutes	0-59 or blank	-4 to -3
Seconds	0-59	-2 to -1
Decimal Points	Period	0
Microseconds	000000-999999	1 to 6

The result of the function is INTEGER with the same sign as the second argument. The result can be null; if the argument is null, the result is the null value.

The returned value is determined for each interval as indicated by the following table:

Table 102. TIMESTAMPDIFF computations	
Result interval	**Computation using duration elements**
Years	Years
Quarters	integer value of (months+(years*12))/3
Months	months + (years*12)
Weeks	integer value of ((days+(months*30))/7)+ (years*52)
Days	days + (months*30)+(years*365)
Hours	hours + ((days+(months*30)+(years*365))*24)
Minutes (the absolute value of the duration must not exceed 40850913020759.999999)	minutes + (hours+ ((days+(months*30)+(years*365))*24)) *60
Seconds (the absolute value of the duration must be less than 680105031408.000000)	seconds + (minutes+(hours+ ((days+(months*30)+(years*365)) *24))*60)*60
Microseconds (the absolute value of the duration must be less than 3547.483648)	microseconds + (seconds+(minutes*60))*1000000

The following assumptions may be used in estimating a difference:

- There are 365 days in a year.
- There are 30 days in a month.
- There are 24 hours in a day.
- There are 60 minutes in an hour.
- There are 60 seconds in a minute.

These assumptions are used when converting the information in the second argument, to the interval type specified in the first argument. The returned estimate may vary by a number of days. For example, if the number of days (interval 16) is requested for the difference between '1997-03-01-00.00.00' and

'1997-02-01-00.00.00', the result is 30. This is because the difference between the timestamps is 1 month, and the assumption of 30 days in a month applies.

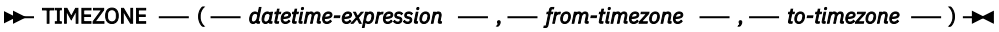
Example

The following example returns 4277, the number of minutes between two timestamps:

```
TIMESTAMPDIFF(4,CHAR(TIMESTAMP('2001-09-29-11.25.42.483219') -
  TIMESTAMP('2001-09-26-12.07.58.065497')))
```

TIMEZONE

The TIMEZONE scalar function converts a date and time in one timezone into a timestamp in another timezone.



The schema is SYSIBM.

datetime-expression
An expression that returns a value of data type DATE, TIMESTAMP, CHAR, or VARCHAR. In a Unicode database, the expression can also be of data type GRAPHIC or VARGRAPHIC. A value of data type CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC must be a valid string that is accepted by the “TIMESTAMP ” on page 525 scalar function. Such a string is implicitly cast to a timestamp before conversion.

from-timezone
An expression that specifies the time zone of the value returned by the input datetime expression.

to-timezone

An expression that specifies the time zone of the output timestamp.

The value returned by a to-timezone or from-timezone expression:

- Must be a built-in character string data type (in a Unicode database, it can also be a graphic string data type) that contains a time zone name as specified in the Internet Assigned Numbers Authority (IANA) time zone database (SQLSTATE 22546).
- Cannot be a FOR BIT DATA subtype (SQLSTATE 42815).

If the value is not of data type VARCHAR, it is cast to VARCHAR before the TIMEZONE function is evaluated.

The standard format for a time zone name in the IANA database is *area*/*location*, where:

area
The English name of a continent or ocean, or the special area **Etc**.

location
The English name of a location within the specified area. This is usually a city or a small island.

For example:

```
"America/Toronto"  [The North American city of Toronto]
"Asia/Sakhalin"    [The Asian island of Sakhalin]
"Etc/UTC"          [Coordinated Universal Time]
```

For a complete list of valid time zone names and the rules associated with those time zones, refer to the IANA time zone database. The database server uses version 2010c of the IANA time zone database. If a newer version of the IANA time zone database is required, contact IBM support.

The precision of the output timestamp depends on the data type of the value returned by the input datetime expression:

- If the input value is a timestamp, the output value is a timestamp with the same precision as the input value.
- If the data type of the input value is DATE, the data type of the output value is TIMESTAMP(0).
- Otherwise, the data type of the output value is TIMESTAMP(6).

If any argument of the function can be null, the result can be null. If any argument is null, the result is the null value.

Examples

- The data type of column col1 of table T1 is TIMESTAMP(3), so the data type of the output of the following statement is also TIMESTAMP(3):

```
select TIMEZONE(col1, 'America/New_York', 'America/Los_Angeles')from T1;
```

  This statement returns: `2016-12-19-14.00.00.123`.
- The data type of column col3 of table T5 is DATE, so the data type of the output of the following statement is TIMESTAMP(0):

```
select TIMEZONE(col3, 'America/New_York', 'America/Los_Angeles')from T5;
```

  This statement returns: `2016-07-14-23.00.00`.
- The input of each of the following statements is a string literal, so the data type of their output is TIMESTAMP(6):

```
values TIMEZONE('2016-09-24 17:00:00.12345678', 'America/New_York', 'America/Los_Angeles');
```

  This statement returns: `2016-09-24-17.00.00.123456`.

```
values TIMEZONE('2016-09-24 17:00:00.123', 'America/New_York', 'America/Los_Angeles');
```

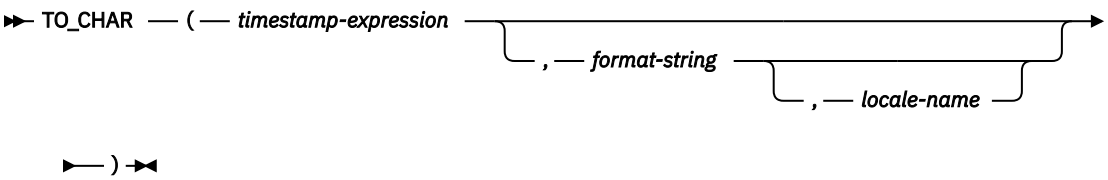
  This statement returns: `2016-09-24-17.00.00.123000`.

TO_CHAR

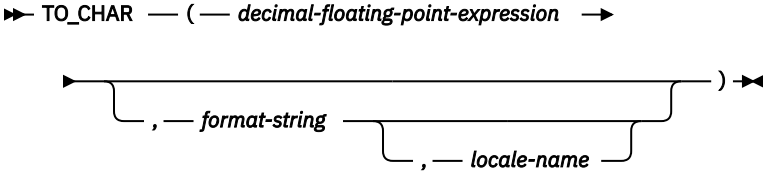
The TO_CHAR function returns a character representation of an input expression.

Character to varchar

Timestamp to varchar



Decimal floating-point to varchar

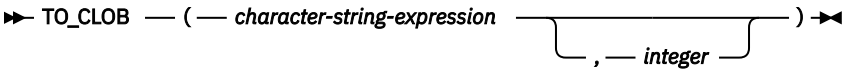


The schema is SYSIBM.

The TO_CHAR scalar function is a synonym for the VARCHAR_FORMAT scalar function.

TO_CLOB

The TO_CLOB function returns a CLOB representation of a character string type.

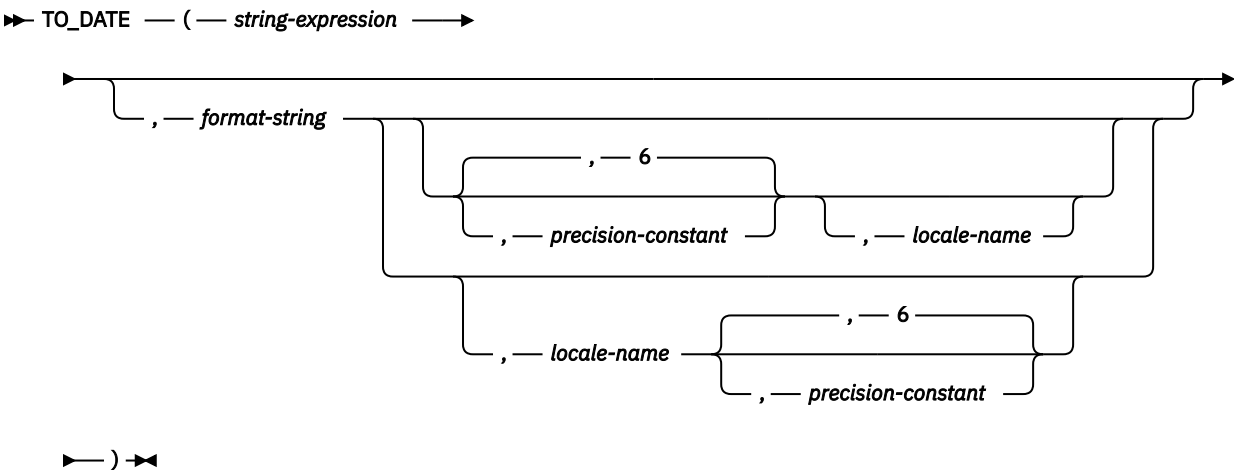


The schema is SYSIBM.

The TO_CLOB scalar function is a synonym for the CLOB scalar function.

TO_DATE

The TO_DATE function returns a timestamp that is based on the interpretation of the input string using the specified format.

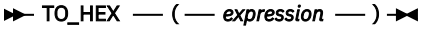


The schema is SYSIBM.

The TO_DATE scalar function is a synonym for the TIMESTAMP_FORMAT scalar function.

TO_HEX

The TO_HEX function converts a numeric expression into the hexadecimal representation.



The schema is SYSIBM.

expression
The expression must return a value that is a built-in character string, Boolean value, or numeric value. In a Unicode database, the expression can also return a graphic string, in which case it is first converted to a character string before the function is evaluated. If the data type of the input is not SMALLINT, INTEGER, or BIGINT, it is implicitly cast to BIGINT before the function is evaluated.

Result

The data type of the result depends on the data type of the input expression:

- For SMALLINT input, the result is VARCHAR(4).
- For INTEGER input, the result is VARCHAR(8).

- For BIGINT input, the result is VARCHAR(16).

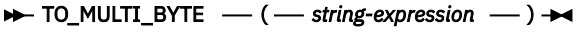
If the argument can be null, the result can be null. If the argument is null, the result is the null value.

Example

```
values to_hex(565);
     returns value 235.
```

TO_MULTI_BYTE

The TO_MULTI_BYTE function returns a Unicode string in which the single-byte characters in a string-expression are converted to their multi-byte (full width) equivalents.



The schema is SYSIBM.

string-expression
An expression that returns a value of a built-in character string.

A character string must not be bit data.

The argument can be also a numeric data type.

The numeric argument is implicitly cast to a VARCHAR data type.

Result

The result of the function is VARCHAR.

The result string unit is the same as string-expression.

The result length attribute depends on the string units as follows:

- If string unit is OCTETS, the result length attribute is the minimum of three times the length attribute of string-expression and the maximum length for the result data type.
- Otherwise, the result length attribute is the same as string-expression.

If the actual length of the result string exceeds the maximum for the return type, an error occurs (SQLSTATE 54006).

If the argument can be null, the result can be null. If the argument is null, the result is the null value.

Example

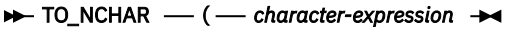
In the following example, the query returns the original single-byte string ‘ABC’, the full width equivalent, and the hexadecimal representation of the full width equivalent:

```
SELECT S, TO_MULTI_BYTE(S) MB, HEX(TO_MULTI_BYTE(S)) HEX
FROM (VALUES 'ABC') T(S)
S   MB        HEX

------------------------------
ABC  A B C EFBCA1EFBCA2EFBCA3

1 record(s) selected.
```

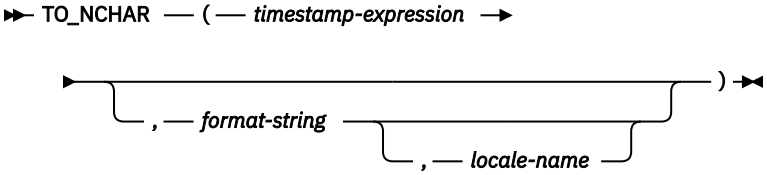
TO_NCHAR

The TO_NCHAR function returns a national character representation of an input expression that has been formatted using a character template.

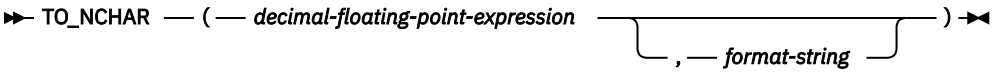
Character to nvarchar



Timestamp to nvarchar



Decimal floating-point to nvarchar



The schema is SYSIBM.

The TO_NCHAR function can be specified only in a Unicode database (SQLSTATE 560AA).

The TO_NCHAR scalar function is equivalent to invoking the TO_CHAR function and casting its result to NVARCHAR.

For more information about TO_NCHAR refer to VARCHAR_FORMAT.

TO_NCLOB

The TO_NCLOB function returns any type of national character string.

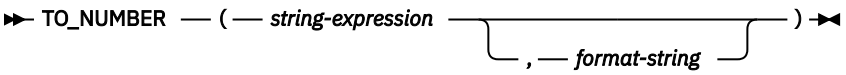
The schema is SYSIBM.

The TO_NCLOB scalar function is a synonym for the NCLOB scalar function.

The TO_NCLOB function can be specified only in a Unicode database (SQLSTATE 560AA).

TO_NUMBER

The TO_NUMBER function returns a DECFLOAT(34) value that is based on the interpretation of the input string using the specified format.

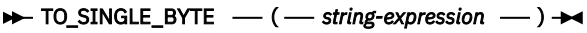


The schema is SYSIBM.

The TO_NUMBER scalar function is a synonym for the DECFLOAT_FORMAT scalar function.

TO_SINGLE_BYTE

The TO_SINGLE_BYTE function returns a string in which multi-byte characters are converted to the equivalent single-byte character where an equivalent character exists.



The schema is SYSIBM.

Only characters that have an equivalent to the single-byte characters represented by the characters in the UTF-8 code point range U+0020 to U+007E will be converted. If a multi-byte character does not have a single-byte equivalent, then it remains unchanged.

string-expression
An expression that specifies the string which gets converted. The expression must return a value that is a built-in CHAR or VARCHAR data type. In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function. The expression cannot be a character string defined as FOR BIT DATA (SQLSTATE 42815).

The data type, code page and length attribute of the result is the same as the data type, code page and length attribute of the argument. If the argument can be null, the result can be null. If the argument is null, the result is the null value.

Example

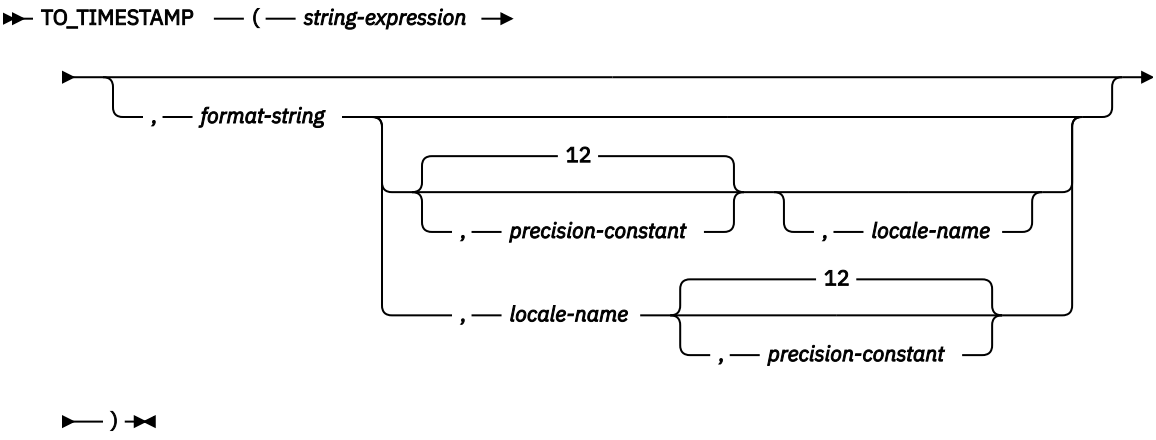
Convert the full width UTF-8 characters 'ABC' (x'efbca1efbca2efbca3' in hex format) to their single byte equivalents.

```
VALUES TO_SINGLE_BYTE(x'efbca1efbca2efbca3')
```

The result is the value 'ABC' (x'414243' in hex format).

TO_TIMESTAMP

The TO_TIMESTAMP function returns a timestamp that is based on the interpretation of the input string using the specified format.



The schema is SYSIBM.

The TO_TIMESTAMP scalar function is a synonym for the TIMESTAMP_FORMAT scalar function except that the default value for *precision-constant* is 12.

TO_UTC_TIMESTAMP

The TO_UTC_TIMESTAMP scalar function returns a TIMESTAMP that is converted to Coordinated Universal Time from the timezone that is specified by the timezone string. TO_UTC_TIMESTAMP is a a statement deterministic function.

The schema is SYSIBM.

expression
An expression that specifies the timestamp that is in the *timezone-expression* time zone. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. If *expression* does not contain time information, a time of midnight (00.00.00) is used for the argument. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported through implicit casting. If expression is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

timezone-expression
An expression that specifies the time zone that the expression is to be adjusted from. The expression must return a value that is a built-in character string, numeric, or datetime data type. In a Unicode database, the expression can also be a graphic string data type. Numeric and datetime data types are supported through implicit casting. If the expression is not a VARCHAR, it is cast to VARCHAR before the function is evaluated. The expression must not be a FOR BIT DATA subtype (SQLSTATE 42815). *timezone-expression* must not be null if *expression* is not null (SQLSTATE 42815).

The value of the *timezone-expression* must be a time zone name from the Internet Assigned Numbers Authority (IANA) time zone database. The standard format for a time zone name in the IANA database is `Area/Location`, where:

- `Area` is the English name of a continent, ocean, or the special area 'Etc'.
- `Location` is the English name of a location within the area; usually a city, or small island.

Examples:

- "America/Toronto"
- "Asia/Sakhalin"
- "Etc/UTC" (which represents Coordinated Universal Time)

For complete details on the valid set of time zone names and the rules that are associated with those time zones, refer to the IANA time zone database. The database server uses version **2010c** of the IANA time zone database. Contact IBM support if a newer version of the IANA time zone database is required.

The result of the function is a timestamp with the same precision as *expression*, if *expression* is a timestamp. If expression is a DATE, the result of the function is a TIMESTAMP(0). Otherwise, the result of the function is a TIMESTAMP(6).

The result can be null; if the *expression* is null, the result is the null value. The *timezone-expression* cannot be null if a not-null value was supplied for the *expression* (SQLSTATE 42815).

The result is the *expression*, adjusted to the Coordinated Universal Time time zone from the time zone specified by the *timezone-expression*. If the *timezone-expression* returns a value that is not a time zone in the IANA time zone database, then the value of *expression* is returned without being adjusted.

The timestamp adjustment is done by first applying the raw offset from Coordinated Universal Time of the *timezone-expression*. If Daylight Saving Time is in effect at the adjusted timestamp for the time zone that is specified by the *timezone-expression*, then the Daylight Saving Time offset is also applied to the timestamp.

Time zones that use Daylight Saving Time have ambiguities at the transition dates. When a time zone changes from standard time to Daylight Saving Time, a range of time does not occur as it is skipped during

the transition. When a time zone changes from Daylight Saving Time to standard time, a range of time occurs twice. Ambiguous timestamps are treated as if they occurred when standard time was in effect for the time zone.

Examples

1. Convert the timestamp '1970-01-01 00:00:00' to the Coordinated Universal Time timezone from the 'America/Denver' timezone. The following returns a TIMESTAMP with the value '1970-01-01 07:00:00'.

   ```
   TO_UTC_TIMESTAMP(TIMESTAMP'1970-01-01 00:00:00', 'America/Denver')
   ```

2. The database administrator created a read-only global variable, SERVER_TIMEZONE, which contains the server's timezone. In this example, the SERVER_TIMEZONE user-defined global variable is set to 'America/Denver'.

   Convert the timestamp '1970-01-01 00:00:00' to the Coordinated Universal Time timezone from the server's timezone. The following returns a TIMESTAMP with the value '1970-01-01 07:00:00'.

   ```
   TO_UTC_TIMESTAMP(TIMESTAMP'1970-01-01 00:00:00', SERVER_TIMEZONE)
   ```

TOTALORDER

The TOTALORDER function returns a SMALLINT value of -1, 0, or 1 that indicates the comparison order of two arguments.

The schema is SYSIBM.

decfloat-expression1
An expression that returns a value of any built-in numeric data type. If the argument is not DECFLOAT(34), it is logically converted to DECFLOAT(34) for processing.

decfloat-expression2
An expression that returns a value of any built-in numeric data type. If the argument is not a decimal floating-point value, it is converted to DECFLOAT(34) for processing.

Numeric comparison is exact, and the result is determined for finite operands as if range and precision were unlimited. An overflow or underflow condition cannot occur.

If one value is DECFLOAT(16) and the other is DECFLOAT(34), the DECFLOAT(16) value is converted to DECFLOAT(34) before the comparison is made.

The semantics of the TOTALORDER function are based on the total order predicate rules of IEEE 754R. TOTALORDER returns the following values:

- -1 if *decfloat-expression1* is lower in order compared to *decfloat-expression2*
- 0 if both *decfloat-expression1* and *decfloat-expression2* have the same order
- 1 if *decfloat-expression1* is higher in order compared to *decfloat-expression2*

The ordering of the special values and finite numbers is as follows:

```
-NAN<-SNAN<-INFINITY<-0.10<-0.100<-0<0<0.100<0.10<INFINITY<SNAN<NAN
```

The result of the function is a SMALLINT value. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Examples

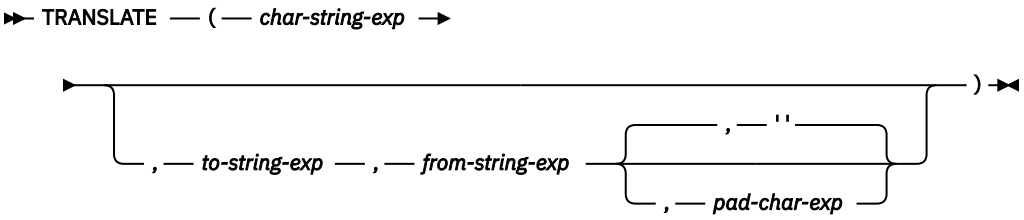
The following examples show the use of the TOTALORDER function to compare decimal floating point values:

```
TOTALORDER(-INFINITY, -INFINITY) = 0
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-1.0)) = 0
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-1.00)) = -1
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-0.5)) = -1
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(0.5)) = -1
TOTALORDER(DECFLOAT(-1.0), INFINITY) = -1
TOTALORDER(DECFLOAT(-1.0), SNAN) = -1
TOTALORDER(DECFLOAT(-1.0), NAN) = -1
TOTALORDER(NAN, DECFLOAT(-1.0)) = 1
TOTALORDER(-NAN, -NAN) = 0
TOTALORDER(-SNAN, -SNAN) = 0
TOTALORDER(NAN, NAN) = 0
TOTALORDER(SNAN, SNAN) = 0
TOTALORDER(-1.0, -1.0) = 0
TOTALORDER(-1.0, -1.00) = -1
TOTALORDER(-1.0, -0.5) = -1
TOTALORDER(-1.0, 0.5) = -1
TOTALORDER(-1.0, INFINITY) = -1
TOTALORDER(-1.0, SNAN) = -1
TOTALORDER(-1.0, NAN) = -1
```

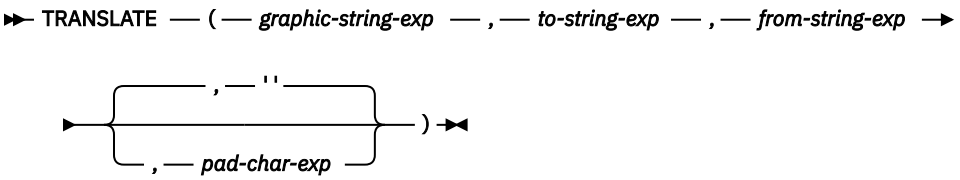
TRANSLATE

The TRANSLATE function returns a value in which one or more characters in a string expression might have been converted to other characters.

character string expression:



graphic string expression:



The schema is SYSIBM.

The function converts all the characters in *char-string-exp* or *graphic-string-exp* that also occur in *from-string-exp* to the corresponding characters in *to-string-exp* or, if no corresponding characters exist, to the pad character specified by *pad-char-exp*.

char-string-exp* or *graphic-string-exp
The string that is to be converted. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

to-string-exp
A string of characters to which certain characters in *char-string-exp* are to be converted.

The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function. If a value for *to-string-exp* is not specified, and the data type is not graphic, all characters in *char-string-exp* will be in monocase; that

is, the characters a-z will be converted to the characters A-Z, and other characters will be converted to their uppercase equivalents, if they exist. For example, in code page 850, é maps to É, but ÿ is not mapped, because code page 850 does not include Ÿ. If the code point length of the result character is not the same as the code point length of the source character, the source character is not converted.

from-string-exp
A string of characters that, if found in *char-string-exp*, are to be converted to the corresponding character in *to-string-exp*.

The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function. If *from-string-exp* contains duplicate characters, the first one found will be used, and the duplicates will be ignored. If *to-string-exp* is longer than *from-string-exp*, the surplus characters will be ignored. If *to-string-exp* is specified, *from-string-exp* must also be specified.

pad-char-exp
A single character used to pad *to-string-exp* if *to-string-exp* is shorter than *from-string-exp*. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function. The value must have a length attribute of zero or one. If a zero-length string is specified, characters in the *from-string-exp* with no corresponding character in the *to-string-exp* are removed from *char-string-exp* or *graphic-string-exp*. If a value is not specified a single-byte blank character is assumed.

With *graphic-string-exp*, only *pad-char-exp* is optional (if a value is not specified, the double-byte blank character is assumed), and each argument, including the pad character, must be of a graphic data type.

The data type and code page of the result is the same as the data type and code page of the first argument. If the first argument is a host variable, the code page of the result is the database code page. In a non-Unicode database, if any argument is graphic string, then all the arguments must be graphic strings (SQLSTATE 42815). In a Unicode database: if the first argument is a FOR BIT DATA character string, then no other argument can be a graphic string (SQLSTATE 42846); if the first argument is a graphic string, then no other argument can be a FOR BIT DATA character string (SQLSTATE 42846).

The length attribute and string units of the result is the same as that of the first argument. If any argument can be null, the result can be null. If any argument is null, the result is the null value.

If the arguments are of data type CHAR or VARCHAR, the corresponding characters in *to-string-exp* and *from-string-exp* must have the same number of bytes (except in the case of a zero-length string). For example, it is not valid to convert a single-byte character to a multi-byte character, or to convert a multi-byte character to a single-byte character. The *pad-char-exp* argument cannot be the first byte of a valid multi-byte character (SQLSTATE 42815).

The characters are matched using a binary comparison. The database collation is not used.

If only *char-string-exp* is specified, single-byte characters will be monocased, and multi-byte characters will remain unchanged.

Examples

For the provided examples, assume that the host variable SITE (VARCHAR(30)) has a value of 'Hanauma Bay'.

- *Example 1*: The following example returns the value 'HANAUMA BAY'.

```
TRANSLATE(:SITE)
```

- *Example 2*: The following example returns the value 'Hanauma jay'.

```
TRANSLATE(:SITE, 'j', 'B')
```

- *Example 3*: The following example returns the value 'Heneume Bey'.

```
TRANSLATE(:SITE, 'ei', 'aa')
```

- *Example 4*: The following example returns the value 'HAnAumA bA%'.

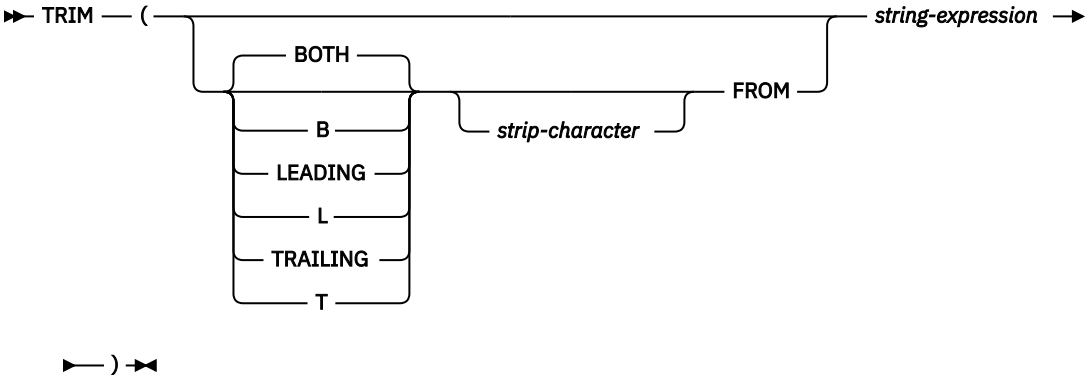
```
TRANSLATE(:SITE, 'bA', 'Bay', '%')
```

- *Example 5*: The following example returns the value 'Hana ma ray'.

```
TRANSLATE(:SITE, 'r', 'Bu')
```

TRIM

The TRIM function removes blanks or another specified character from the end, the beginning, or both ends of a string expression.



The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature.

BOTH, LEADING, or TRAILING
Specifies whether characters are removed from the beginning, the end, or from both ends of the string expression. If this argument is not specified, the characters are removed from both the end and the beginning of the string.

strip-character
A single-character constant that specifies the character that is to be removed. The *strip-character* can be any character whose UTF-32 encoding is a single character or a single digit numeric value. The binary representation of the character is matched.

If *strip-character* is not specified and:

- If the *string-expression* is a DBCS graphic string, the default *strip-character* is a DBCS blank, whose code point is dependent on the database code page
- If the *string-expression* is a UCS-2 graphic string, the default *strip-character* is a UCS-2 blank (X'0020')
- If the *string-expression* is a binary string, the default *strip-character* is a hexadecimal zero (X'00')
- Otherwise, the default *strip-character* is an SBCS blank (X'20')

FROM *string-expression*
An expression that specifies the string from which the result is derived. The expression must return a value that is a built-in CHAR, VARCHAR, BINARY, VARBINARY, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, BINARY, VARBINARY, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before the function is evaluated.

The value for *string-expression* and the value for *strip-character* must have compatible data types.

The data type of the result depends on the data type of the string-expression:

- VARCHAR if the data type is VARCHAR or CHAR
- VARGRAPHIC if the data type is VARGRAPHIC or GRAPHIC
- VARBINARY if the data type is VARBINARY or BINARY

The result is a varying-length string with the same maximum length as the length attribute of the *string-expression*. The actual length of the result is the length of the *string-expression* minus the number of string units that are removed. If all of the characters are removed, the result is an empty varying-length string. The code page of the result is the same as the code page of the *string-expression*.

Examples

- *Example 1*: Assume that the host variable HELLO of type CHAR(9) has a value of ' Hello'.

```
SELECT TRIM(:HELLO),
  TRIM(TRAILING FROM :HELLO)
FROM SYSIBM.SYSDUMMY1
```

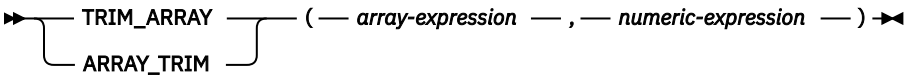
  returns the values 'Hello' and ' Hello', respectively.
- *Example 2*: Assume that the host variable BALANCE of type CHAR(9) has a value of '000345.50'.

```
SELECT TRIM(L '0' FROM :BALANCE),
FROM SYSIBM.SYSDUMMY1
```

  returns the value '345.50'.

TRIM_ARRAY

The TRIM_ARRAY function deletes elements from the end of an array.



The schema is SYSIBM.

array-expression
An SQL variable, SQL parameter, or global variable of an ordinary array type, or a CAST specification of a parameter marker to an ordinary array type. An associative array data type cannot be specified (SQLSTATE 42884).

numeric-expression
Specifies the number of elements trimmed from the end of the array. The numeric expression can be of any numeric data type with a value that can be cast to INTEGER. The value of the numeric expression must be between 0 and the cardinality of the array expression (SQLSTATE 2202E).

Result

The function returns a value with the same array type as the array expression but with the cardinality reduced by the value of INTEGER(*numeric-expression*).

The result can be null; if either argument is null, the result is the null value.

Rules

- The TRIM_ARRAY function is not supported for associative arrays (SQLSTATE 42884).
- The TRIM_ARRAY function can only be used on the right side of an assignment statement in contexts where arrays are supported (SQLSTATE 42884).

Examples

1. *Example 1*: Remove the last element from the array variable RECENT_CALLS.

```
SET RECENT_CALLS = TRIM_ARRAY(RECENT_CALLS, 1)
```

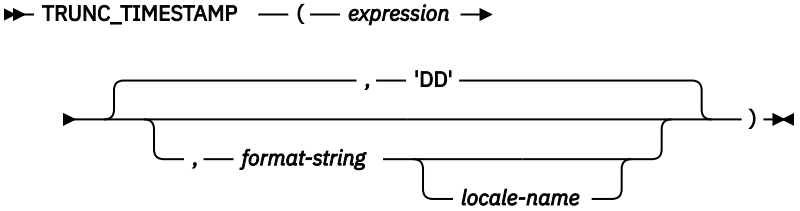
2. *Example 2*: Assign only the first two elements from the array variable SPECIALNUMBERS to the SQL array variable EULER_CONST:

```
SET EULER_CONST = TRIM_ARRAY(SPECIALNUMBERS, 8)
```

   The result is that EULER_CONST will be assigned an array with two elements, the first element value is 2.71828183 and the second element value is the null value.

TRUNC_TIMESTAMP

The TRUNC_TIMESTAMP scalar function returns a TIMESTAMP that is an argument (*expression*) truncated to the unit specified by another argument (*format-string*).



The schema is SYSIBM.

If *format-string* is not specified, *expression* is truncated to the nearest day, as if 'DD' was specified for *format-string*.

expression
An expression that returns a value of one of the following built-in data types: a DATE or a TIMESTAMP.

format-string
An expression that returns a built-in character string data type with an actual length that is not greater than 255 bytes. The format element in *format-string* specifies how *expression* should be truncated. For example, if *format-string* is 'DD', a timestamp that is represented by *expression* is truncated to the nearest day. Leading and trailing blanks are removed from the string, and the resulting substring must be a valid format element for a timestamp (SQLSTATE 22007). The default is 'DD'.

Allowable values for *format-string* are listed in the table of format elements found in the description of the ROUND function.

locale-name
A character constant that specifies the locale used to determine the first day of the week when using format model values DAY, DY, or D. The value of *locale-name* is not case sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming, see "Locale names for SQL and XQuery". If *locale-name* is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result of the function is a TIMESTAMP with the same timestamp precision as *expression*. The result can be null; if any argument is null, the result is the null value.

The result of the function is a TIMESTAMP with a timestamp precision of:

- *p* when the data type of expression is TIMESTAMP(*p*)
- 0 when the data type of expression is DATE
- 6 otherwise

The result can be null; if any argument is null, the result is the null value.

Notes

- **Determinism:** TRUNC_TIMESTAMP is a deterministic function. However, the following invocations of the function depend on the value of the special register CURRENT LOCALE LC_TIME.
  - Truncate of a date or timestamp value when *locale-name* is not explicitly specified and one of the following is true:
    - *format-string* is not a constant
    - *format-string* is a constant and includes format elements that are locale sensitive

    Invocations of the function that depend on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621, 428EC, or 429BX).

Example

Set the host variable TRNK_TMSTMP with the current year rounded to the nearest year value.

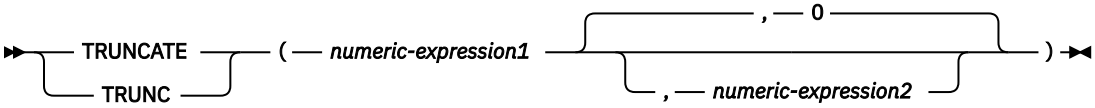
```
SET :TRNK_TMSTMP = TRUNC_TIMESTAMP('2000-03-14-17.30.00', 'YEAR');
```

The host variable TRNK_TMSTMP is set with the value 2000-01-01-00.00.00.000000.

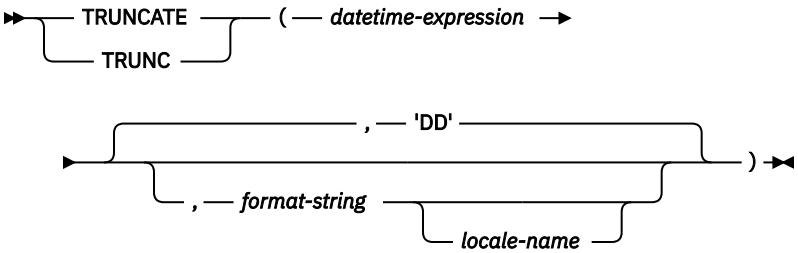
TRUNCATE or TRUNC

The TRUNCATE function returns a truncated value of a number or a datetime value.

TRUNCATE numeric:



TRUNCATE datetime:



The schema is SYSIBM. The SYSFUN version of the TRUNCATE numeric function continues to be available.

The data type of the return value depends on the first argument:

- If the result of the first argument is a numeric value, a number is returned, truncated to the specified number of places to the right or left of the decimal point.
- If the first argument is a DATE, TIME, or TIMESTAMP, a datetime value, truncated to the unit specified by *format-string*.

TRUNCATE numeric

If *numeric-expression1* has a numeric data type, the TRUNCATE function returns *numeric-expression1* truncated to *numeric-expression2* places to the right of the decimal point if *numeric-expression2* is positive, or to the left of the decimal point if *numeric-expression2* is zero or negative. If *numeric-expression2* is not specified, *numeric-expression1* is truncated to zero places left of the decimal point.

numeric-expression1
An expression that must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, or numeric data type. If the value is not a numeric data type, it is implicitly cast to DECFLOAT(34) before evaluating the function.

If the expression is a decimal floating-point data type, the DECFLOAT rounding mode will not be used. The rounding behavior of TRUNCATE corresponds to a value of ROUND_DOWN. If a different rounding behavior is wanted, use the QUANTIZE function.

numeric-expression2
An expression that returns a value that is a built-in numeric data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

If *numeric-expression2* is not negative, *numeric-expression1* is truncated to the absolute value of *numeric-expression2* number of places to the right of the decimal point.

If *numeric-expression2* is negative, *numeric-expression1* is truncated to the absolute value of *numeric-expression2* + 1 number of places to the left of the decimal point. If the absolute value of a negative *numeric-expression2* is larger than the number of digits to the left of the decimal point, the result is 0. For example:

```
TRUNCATE(748.58,-4) = 0
```

The data type, length, and scale attributes of the result are the same as the data type, length, and scale attributes of the first argument.

If either argument can be null, the result can be null. If either argument is null, the result is the null value.

TRUNCATE datetime
If *datetime-expression* has a datetime data type, the TRUNCATE function returns *datetime-expression* rounded to the unit specified by the *format-string*. If *format-string* is not specified, *datetime-expression* is truncated to the nearest day, as if 'DD' is specified for *format-string*.

datetime-expression
An expression that must return a value that is a DATE, a TIME, or a TIMESTAMP. String representations of these data types are not supported and must be explicitly cast to a DATE, TIME, or TIMESTAMP for use with this function; alternatively, you can use the TRUNC_TIMESTAMP function for a string representation of a date or timestamp.

format-string
An expression that returns a built-in character string data type with an actual length that is not greater than 255 bytes. The format element in *format-string* specifies how *datetime-expression* should be truncated. For example, if *format-string* is 'DD', a timestamp that is represented by *datetime-expression* is truncated to the nearest day. Leading and trailing blanks are removed from the string, and the resulting substring must be a valid format element for the type of *datetime-expression* (SQLSTATE 22007). The default is 'DD', which cannot be used if the data type of *datetime-expression* is TIME.

Allowable values for *format-string* are listed in the table of format elements found in the description of the ROUND function.

locale-name
A character constant that specifies the locale used to determine the first day of the week when using format element values DAY, DY, or D. The value of *locale-name* is not case sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming, see "Locale names for SQL and XQuery". If *locale-name* is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result of the function has the same date type as *datetime-expression*. The result can be null; if any argument is null, the result is the null value.

The data type and length attribute of the result are the same as the data type and length attribute of the first argument.

The result can be null if the argument can be null or if the argument is not a decimal floating-point number and the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

Notes

- **Determinism:** TRUNCATE is a deterministic function. However, the following invocations of the function depend on the value of the special register CURRENT LOCALE LC_TIME.
  - Truncate of a datetime value when *locale-name* is not explicitly specified and one of the following is true:
    - *format-string* is not a constant
    - *format-string* is a constant and includes format elements that are locale sensitive

    Invocations of the function that depend on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621, 428EC, or 429BX).

Examples

- *Example 1*: Using the EMPLOYEE table, calculate the average monthly salary for the highest paid employee. Truncate the result two places to the right of the decimal point.

```
SELECT TRUNCATE(MAX(SALARY)/12,2)
  FROM EMPLOYEE;
```

  Assuming the highest paid employee earns $52750.00 per year, the example returns 4395.83.
- *Example 2*: Display the number 873.726 truncated 2, 1, 0, -1, and -2 decimal places, respectively.

```
VALUES (
  TRUNCATE(873.726,2),
  TRUNCATE(873.726,1),
  TRUNCATE(873.726,0),
  TRUNCATE(873.726,-1),
  TRUNCATE(873.726,-2),
  TRUNCATE(873.726,-3) );
```

  This example returns 873.720, 873.700, 873.000, 870.000, 800.000, and 0.000.
- *Example 3*: Display the decimal-floating point number 873.726 truncated 0 decimal places.

```
VALUES(TRUNCATE(DECFLOAT(873.726),0))
```

  Returns the value 873.
- *Example 4*: Set the variable vTRNK_DT with the input date rounded to the nearest month value.

```
SET vTRNK_DT = TRUNC(DATE('2000-08-16'), 'MONTH');
```

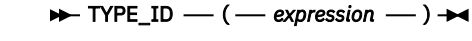
  The value set is 2000-08-01.
- *Example 5*: Set the host variable TRNK_TMSTMP with the current year rounded to the nearest year value.

```
SET :TRNK_TMSTMP = TRUNCATE(DATE('2000-03-14-17.30.00'), 'YEAR');
```

  The value set is 2000-01-01-00.00.00.000000.

TYPE_ID

The TYPE_ID function returns the internal type identifier of the dynamic data type of the *expression*.



The schema is SYSIBM.

expression
An expression that returns a value of a user-defined structured data type.

The data type of the result of the function is INTEGER. If *expression* can be null, the result can be null; if *expression* is null, the result is the null value.

The value returned by the TYPE_ID function is not portable across databases. The value may be different, even though the type schema and type name of the dynamic data type are the same. When coding for portability, use the TYPE_SCHEMA and TYPE_NAME functions to determine the type schema and type name.

Notes

- This function cannot be used as a source function when creating a user-defined function. Because it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.

Example

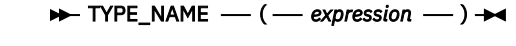
A table hierarchy exists having root table EMPLOYEE of type EMP and subtable MANAGER of type MGR. Another table ACTIVITIES includes a column called WHO_RESPONSIBLE that is defined as REF(EMP) SCOPE EMPLOYEE. For each reference in ACTIVITIES, display the internal type identifier of the row that corresponds to the reference.

```
SELECT TASK, WHO_RESPONSIBLE->NAME,
    TYPE_ID(DEREF(WHO_RESPONSIBLE))
  FROM ACTIVITIES
```

The DEREF function is used to return the object corresponding to the row.

TYPE_NAME

The TYPE_NAME function returns the unqualified name of the dynamic data type of the *expression*.



The schema is SYSIBM.

expression
An expression that returns a value of a user-defined structured data type.

The data type of the result of the function is VARCHAR(128 OCTETS). If *expression* can be null, the result can be null; if *expression* is null, the result is the null value. Use the TYPE_SCHEMA function to determine the schema name of the type name returned by TYPE_NAME.

Notes

- This function cannot be used as a source function when creating a user-defined function. Because it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.

Example

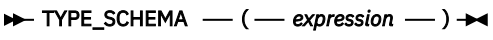
A table hierarchy exists having root table EMPLOYEE of type EMP and subtable MANAGER of type MGR. Another table ACTIVITIES includes a column called WHO_RESPONSIBLE that is defined as REF(EMP) SCOPE EMPLOYEE. For each reference in ACTIVITIES, display the type of the row that corresponds to the reference.

```
SELECT TASK, WHO_RESPONSIBLE->NAME,
    TYPE_NAME(DEREF(WHO_RESPONSIBLE)),
    TYPE_SCHEMA(DEREF(WHO_RESPONSIBLE))
  FROM ACTIVITIES
```

The DEREF function is used to return the object corresponding to the row.

TYPE_SCHEMA

The TYPE_SCHEMA function returns the schema name of the dynamic data type of the *expression*.



The schema is SYSIBM.

expression
An expression that returns a value of a user-defined structured data type.

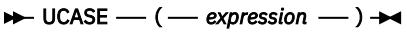
The data type of the result of the function is VARCHAR(128 OCTETS). If *expression* can be null, the result can be null; if *expression* is null, the result is the null value. Use the TYPE_NAME function to determine the type name associated with the schema name returned by TYPE_SCHEMA.

Notes

- This function cannot be used as a source function when creating a user-defined function. Because it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.

UCASE

The UCASE function is identical to the TRANSLATE function except that only the first argument (*char-string-exp*) is specified.

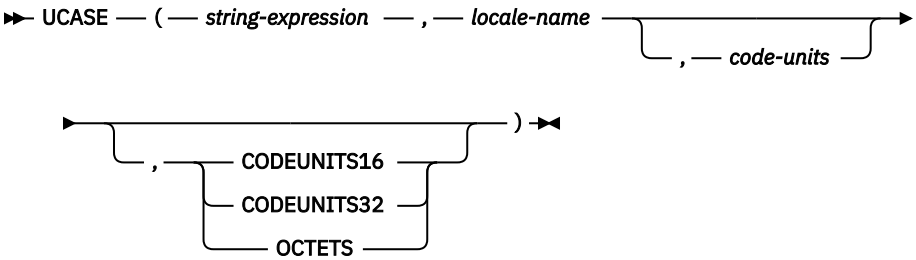


The schema is SYSIBM.

UCASE is a synonym for UPPER.

UCASE (locale sensitive)

The UCASE function returns a string in which all characters have been converted to uppercase characters using the rules associated with the specified locale.

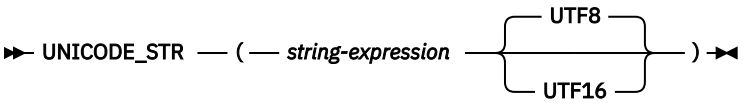


The schema is SYSIBM.

UCASE is a synonym for UPPER.

UNICODE_STR

The UNICODE_STR function returns a string in Unicode UTF-8 or UTF-16, depending on the specified option. The string represents a Unicode encoding of the input string.



The schema is SYSIBM.

string-expression

An expression that returns a value of a built-in character or graphic string.

A character string must not be bit data.

Values that are preceded by a backslash ('\') are treated as Unicode UTF-16 characters, for example, '\0041' is the Unicode UTF-16 representation for 'A').

A double backslash '\\' indicates a backslash in the string.

A backslash that is not part of a double backslash must be following by four hexadecimal digits (SQLSTATE 42815).

A partial surrogate character in the expression is replaced with a blank.

The argument can also be a numeric data type.

The numeric argument is implicitly cast to a VARCHAR data type.

UTF8 or UTF16

Specifies the Unicode encoding of the result.

- If UTF8 is specified, the result is returned as a Unicode UTF-8 character string.
- If UTF16 is specified, the result is returned as a Unicode UTF-16 graphic string.

UTF8 is the default.

Result

The result of the function depends on the second argument:

- If UTF8 is specified, the result is VARCHAR.
- If UTF16 is specified, the result is VARGRAPHIC.

The length attribute of the result depends on the second argument (UTF8 or UTF16).

- If the second argument is UTF8, the length attribute of the result is MIN(*n*,32672) OCTETS, where *n* depends on the length attribute and string units of the string-expression as follows:
  - If the string-expression contains the string unit OCTETS, *n* is the length attribute of the input string.
  - If the string-expression contains the string unit CODEUNITS16, *n* is twice the length attribute of the input string.
  - If the string-expression contains the string unit CODEUNITS32, *n* is four times the length attribute of the input string.
- If the second argument is UTF16, the length attribute of the result is MIN(*n*,16336) CODEUNITS16, where *n* depends on the length attribute and string units of the string-expression as follows:
  - If the string-expression contains the string units OCTETS or CODEUNITS16, *n* is the length attribute of the input string.
  - If the string-expression contains the string unit CODEUNITS32, *n* is twice the length attribute of the input string.

Notes

As a syntax alternative, you can specify UNISTR as a synonym for UNICODE_STR.

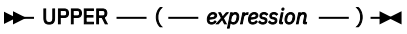
The following example sets the host variable *HV1* to a VARCHAR value that represents the Unicode UTF-8 string that corresponds to the argument:

```
SET :HV1 = UNICODE_STR('Hi, my name is \5CF0');
```

HV1 is assigned a Unicode UTF-8 string with the following value 'Hi, my name is 峰'.

UPPER

The UPPER function is identical to the TRANSLATE function except that only the first argument (*char-string-exp*) is specified.

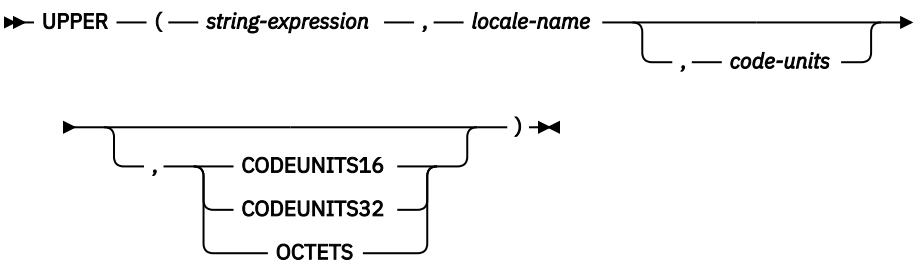


The schema is SYSIBM. (The SYSFUN version of this function continues to be available for upward compatibility.)

expression
An expression that returns either a Boolean value or a built-in character string that is not FOR BIT DATA. In a Unicode database, the expression can also return a graphic string, in which case it is first converted to a character string before the function is evaluated.

UPPER (locale sensitive)

The UPPER function returns a string in which all characters have been converted to uppercase characters using the rules associated with the specified locale.



The schema is SYSIBM.

string-expression
An expression that returns a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC string. If *string-expression* is CHAR or VARCHAR, the expression must not be FOR BIT DATA (SQLSTATE 42815).

locale-name
A character constant that specifies the locale that defines the rules for conversion to uppercase characters. The value of *locale-name* is not case sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming, see "Locale names for SQL and XQuery".

Attention: In the container-only release of Db2 Version 11.5 Mod Pack 1 and later versions, passing UNI_SIMPLE as locale-name will enable use of simple case folding mapping.

code-units
An integer constant that specifies the number of code units in the result. If specified, *code-units* must be an integer between:

- 1 and 32 672 if the string unit of the result is OCTETS
- 1 and 16 336 if the string unit of the result is double bytes or CODEUNITS16

- 1 and 8 168 if the string unit of the result is CODEUNITS32

otherwise an error (SQLSTATE 42815). If *code-units* is not explicitly specified, it is implicitly the length attribute of *string-expression*. If OCTETS is specified and the result is graphic data, the value of *code-units* must be even, otherwise an error is returned (SQLSTATE 428GC). If OCTETS is specified and the string units of the result is CODEUNIT32, the value of *code-units* must be a multiple of 4 (SQLSTATE 428GC). If CODEUNITS16 is specified and the string units of the result is CODEUNITS32, the value of *code-units* must be a multiple of 2 (SQLSTATE 428GC).

CODEUNITS16, CODEUNITS32, or OCTETS
Specifies the string unit of *code-units*.

CODEUNITS16 specifies that *code-units* is expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *code-units* is expressed in 32-bit UTF-32 code units. OCTETS specifies that *code-units* is expressed in bytes.

If a string unit is not explicitly specified, the string unit of *string-expression* determines the unit that is used. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

The result of the function is VARCHAR if *string-expression* is CHAR or VARCHAR, and VARGRAPHIC if *string-expression* is GRAPHIC or VARGRAPHIC. The string units of the result is the same as the string units of *string-expression*

The length attribute of the result is determined by the implicit or explicit value of *code-units*, the implicit or explicit string unit, the result data type, and the result string units, as shown in the following table:

Table 103. Length attribute of the result of UPPER as a function of string unit and result type

Data type and string units of result	**Length attribute for *code-units* in CODEUNITS16**	**Length attribute for *code-units* in CODEUNITS32**	**Length attribute for *code-units* in OCTETS**
VARCHAR in OCTETS	MIN(*code-units* * 3, 32672)	MIN(*code-units* * 4, 32672)	*code-units*
VARCHAR in CODEUNITS32	MIN(*code-units* / 2, 8168)	MIN(*code-units*, 8168)	MIN(*code-units* / 4, 8168)
VARGRAPHIC in CODEUNITS16 or double bytes	*code-units*	MIN(*code-units* * 2, 16336)	MIN(*code-units* / 2, 16336)
VARGRAPHIC in CODEUNITS32	MIN(*code-units* / 2, 8168)	MIN(*code-units*, 8168)	MIN(*code-units* / 4, 8168)

The actual length of the result might be greater than the length of *string-expression*. If the actual length of the result is greater than the length attribute of the result, an error is returned (SQLSTATE 42815). If the number of code units in the result exceeds the value of *code-units*, an error is returned (SQLSTATE 42815).

If *string-expression* is not in UTF-16, this function performs code page conversion of *string-expression* to UTF-16, and of the result from UTF-16 to the code page of *string-expression*. If either code page conversion results in at least one substitution character, the result includes the substitution character, a warning is returned (SQLSTATE 01517), and the warning flag SQLWARN8 in the SQLCA is set to 'W'.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Examples

- *Example 1*: Ensure that the characters in the value of column JOB in the EMPLOYEE table are returned in uppercase characters.

```
SELECT UPPER(JOB, 'en_US')
  FROM EMPLOYEE
  WHERE EMPNO = '000020'
```

  The result is the value 'MANAGER'.

- *Example 2*: Find the uppercase characters for all the 'I' characters in a Turkish string.

```
VALUES UPPER('Iİıi', 'tr_TR', CODEUNITS16)
```

  The result is the string 'IİIİ'.

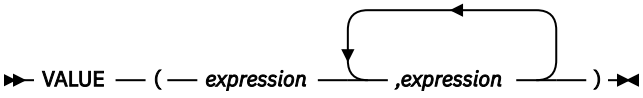
- *Example 3*: Find the uppercase form of the German 'ß' (sharp S).

```
VALUES UPPER('ß', 'de', 2, CODEUNITS16)
```

  The result is the string 'SS'. Note that *code-units* must be specified in this example, because there are more code units in the result than in *string-expression*.

VALUE

The VALUE function returns the first argument that is not null.

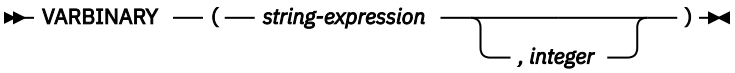


The schema is SYSIBM.

VALUE is a synonym for COALESCE.

VARBINARY

The VARBINARY function returns a VARBINARY (varying-length binary string) representation of a string of any data type.



The schema is SYSIBM.

string-expression
An expression that returns a value of a string data type.

integer
An integer constant value, which specifies the length attribute of the resulting VARBINARY data type. The value must be 1 - 32672. If *integer* is not specified, the length attribute of the result is the lower of the following values:

- The maximum length for the VARBINARY data type
- The length attribute for the data type of *string-expression* expressed in bytes:
  - The length attribute, if *string-expression* is a binary string, a character string that is FOR BIT DATA, or a character string with string units of OCTETS
  - The length attribute multiplied by 2, if *string-expression* is a graphic string with string units of CODEUNITS16 or double bytes
  - The length attribute multiplied by 4, if *string-expression* is a character or graphic string with string units of CODEUNITS32

If the length attribute of *string-expression* is zero and the *integer* argument is not specified, the length attribute of the result is 1.

The result of the function is a VARBINARY. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The actual length of the result is the minimum of the length attribute of the result and the actual length in bytes of the *string-expression*. If the length of *string-expression* that is converted to a binary string is greater than the length attribute of the result, truncation occurs. A warning (SQLSTATE 01004) is returned in the following situations:

- The first argument is a character or graphic string (other than a CLOB or DBCLOB) and non-blank characters are truncated.
- The first argument is a binary string (other than BLOB) and non-hexadecimal zeros are truncated.

Examples

1. The following function returns a varying-length binary string with a length attribute 1, actual length 0, and a value of empty string.

```
SELECT VARBINARY('',1)
  FROM SYSIBM.SYSDUMMY1
```

2. The following function returns a varying-length binary string with a length attribute 5, actual length 3, and a value BX'4B4248'.

```
SELECT VARBINARY('KBH',5)
  FROM SYSIBM.SYSDUMMY1
```

3. The following function returns a varying-length binary string with a length attribute 3, actual length 3, and a value BX'4B4248'.

```
SELECT VARBINARY('KBH')
  FROM SYSIBM.SYSDUMMY1
```

4. The following function returns a varying-length binary string with a length attribute 3, actual length 3, and a value BX'4B4248'.

```
SELECT VARBINARY('KBH ',3)
  FROM SYSIBM.SYSDUMMY1
```

5. The following function returns a varying-length binary string with a length attribute 3, actual length 3, a value BX'4B4248', and a warning (SQLSTATE 01004).

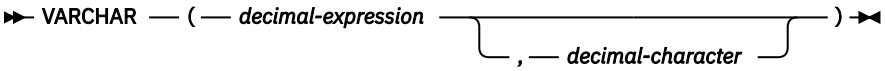
```
SELECT VARBINARY('KBH 93',3)
  FROM SYSIBM.SYSDUMMY1
```

VARCHAR

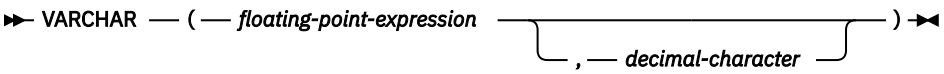
The VARCHAR function returns a varying-length character string representation of a value of a different data type.

Binary integer to VARCHAR

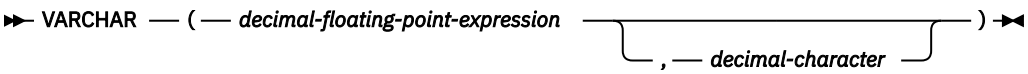
Decimal to VARCHAR



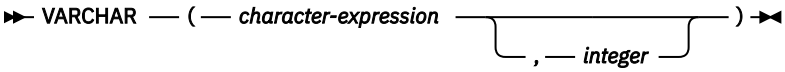
Floating-point to VARCHAR



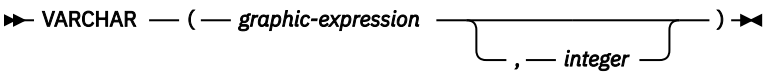
Decimal floating-point to VARCHAR



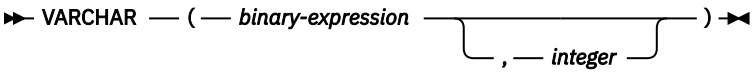
Character string to VARCHAR



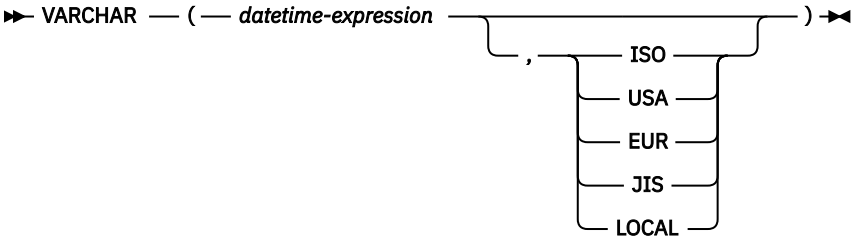
Graphic string to VARCHAR



Binary string to VARCHAR



Datetime to VARCHAR



Boolean to VARCHAR

The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature.

Binary integer to VARCHAR

integer-expression
An expression that returns a value that is of an integer data type (SMALLINT, INTEGER, or BIGINT).

The result is the varying-length string representation of *integer-expression* in the form of an SQL integer constant. The length attribute of the result depends on whether *integer-expression* is a small, large or big integer as follows:

- If the first argument is a small integer, the maximum length of the result is 6.
- If the first argument is a large integer, the maximum length of the result is 11.

- If the first argument is a big integer, the maximum length of the result is 20.

The actual length of the result is the smallest number of characters that can be used to represent the value of the argument. Leading zeros are not included. If the argument is negative, the first character of the result is a minus sign. Otherwise, the first character is a digit.

The code page of the result is the code page of the section.

Decimal to VARCHAR

decimal-expression
An expression that returns a value that is a decimal data type. The DECIMAL scalar function can be used to change the precision and scale.

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a varying-length character string representation of *decimal-expression* in the form of an SQL decimal constant. The length attribute of the result is 2+*p*, where *p* is the precision of *decimal-expression*. The actual length of the result is the smallest number of characters that can be used to represent the result, except that trailing zeros are included. Leading zeros are not included. If decimal-expression is negative, the first character of the result is a minus sign; otherwise, the first character is a digit or the decimal character. If the scale of *decimal-expression* is zero, the decimal character is not returned.

The code page of the result is the code page of the section.

Floating-point to VARCHAR

floating-point-expression
An expression that returns a value that is a floating-point data type (DOUBLE or REAL).

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a varying-length character string representation of *floating-point-expression* in the form of an SQL floating-point constant.

The maximum length of the result is 24. The actual length of the result is the smallest number of characters that can represent the value of floating-point-expression such that the mantissa consists of a single digit other than zero followed by the *decimal-character* and a sequence of digits. If *floating-point-expression* is negative, the first character of the result is a minus sign; otherwise, the first character is a digit. If *floating-point-expression* is zero, the result is 0E0.

The code page of the result is the code page of the section.

Decimal floating-point to VARCHAR

decimal-floating-point-expression
An expression that returns a value that is a decimal floating-point data type (DECFLOAT).

decimal-character
Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a varying-length character string representation of *decimal-floating-point-expression* in the form of an SQL decimal floating-point constant. The maximum length of the result is 42. The actual length of the result is the smallest number of characters that can represent the value of *decimal-floating-point-expression*. If *decimal-floating-point-expression* is negative, the first character of the result is a minus sign; otherwise, the first character is a digit. If *decimal-floating-point-expression* is zero, the result is 0.

If the value of *decimal-floating-point-expression* is the special value Infinity, sNaN, or NaN, the strings "INFINITY", "SNAN", and "NAN", respectively, are returned. If the special value is negative, the first character of the result is a minus sign. The decimal floating-point special value sNaN does not result in a warning when converted to a string.

The code page of the result is the code page of the section.

Character string to VARCHAR

character-expression
An expression that returns a value that is a built-in character string data type.

integer
An integer constant that specifies the length attribute for the resulting varying-length character string. The value must be between 0 and the maximum length for the VARCHAR data type in the string units of the result.

If the second argument is not specified:

- If the *character-expression* is the empty string constant, the length attribute of the result is 0.
- Otherwise, the length attribute of the result is the lower of the following values:
  - The maximum length for the VARCHAR data type in the string units of the result
  - The length attribute of the first argument

The result is a varying-length character string. The length attribute of the result is determined by the value of *integer*. If *character-expression* is the FOR BIT DATA subtype, the result is FOR BIT DATA.

If the length of *character-expression* is greater than the length attribute of the result, several scenarios exist:

- If the string unit of the result is CODEUNITS32, truncation is performed. If only blank characters are truncated and *character-expression* is CHAR or VARCHAR, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004).
- If *integer* is specified, truncation is performed. If only blank characters are truncated and *character-expression* is CHAR or VARCHAR, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004). When part of a multi-byte character is truncated, that partial character is replaced with the blank character. Do not rely on this behavior because it might change in a future release.
- If *integer* is not specified, an error is returned (SQLSTATE 22001).

Graphic string to VARCHAR

graphic-expression
An expression that returns a value that is a built-in graphic string data type.

integer
An integer constant that specifies the length attribute for the resulting varying-length character string. The value must be between 0 and the maximum length for the VARCHAR data type in the string units of the result.

If the second argument is not specified:

- If the *graphic-expression* is the empty string constant, the length attribute of the result is 0.
- If the string units of *graphic-expression* is CODEUNITS32, the length attribute of the result is the lower of the following values:
  - The maximum length for the VARCHAR data type in the string units of the result
  - The length attribute of the first argument
- Otherwise, the length attribute of the result is the lower of the following values:
  - The maximum length for the VARCHAR data type in the string units of the result
  - 3 * length attribute of the first argument

The result is a varying-length character string that is converted from *graphic-expression*. The length attribute of the result is determined by the value of *integer*.

If the length of *graphic-expression* that is converted to a character string is greater than the length attribute of the result, several scenarios exist:

- If the string unit of the result is CODEUNITS32, truncation is performed. If only blank characters are truncated and *graphic-expression* is GRAPHIC or VARGRAPHIC, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004).
- If *integer* is specified and *graphic-expression* is a GRAPHIC or VARGRAPHIC, truncation is performed with no warning returned.
- If *integer* is specified and *graphic-expression* is a DBCLOB, truncation is performed with a warning returned (SQLSTATE 01004).
- If *integer* is not specified and *graphic-expression* is a GRAPHIC or VARGRAPHIC, truncation is performed with no warning returned.
- If *integer* is not specified and *graphic-expression* is a DBCLOB, an error is returned (SQLSTATE 22001).

Binary string to VARCHAR

binary-expression
An expression that returns a value that is a built-in binary string data type.

integer
An integer constant that specifies the length attribute for the resulting varying-length character string. The value must be between 0 and the maximum length for the VARCHAR data type in the string units of the result.

The result is a FOR BIT DATA character string.

Datetime to VARCHAR

datetime-expression
An expression that is of one of the following data types:

DATE
The result is the character string representation of the date in the format specified by the second argument. The length of the result is 10. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

TIME
The result is the character string representation of the time in the format specified by the second argument. The length of the result is 8. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

TIMESTAMP
The result is the character string representation of the timestamp. If the data type of *datetime-expression* is TIMESTAMP(0), the length of the result is 19. If the data type of *datetime-expression* is TIMESTAMP(*n*), where *n* is between 1 and 12, the length of the result is 20+*n*. Otherwise, the length of the result is 26. The second argument must not be specified (SQLSTATE 42815).

The code page of the result is the code page of the section.

Boolean to VARCHAR

boolean-expression
An expression that returns a Boolean value (TRUE or FALSE). The result is either 'TRUE' or 'FALSE'.

Result

The VARCHAR function returns a varying-length character string representation of:

- An integer number, if the only argument is a SMALLINT, INTEGER, or BIGINT value

- A decimal number, if the first argument is a DECIMAL value
- A double-precision floating-point number, if the first argument is a floating-point (DOUBLE or REAL) value
- A decimal floating-point number, if the first argument is a decimal floating-point (DECFLOAT) value
- A character string, if the first argument is a character string (CHAR, VARCHAR, or CLOB) value
- A graphic string (Unicode databases only), if the first argument is a graphic string (GRAPHIC, VARGRAPHIC, or DBCLOB) value
- A datetime value, if the first argument is a datetime (DATE, TIME, or TIMESTAMP) value
- A Boolean value ('TRUE' or 'FALSE'), if the only argument is a BOOLEAN value (TRUE or FALSE)

In a non-Unicode database, the string units of the result is OCTETS. Otherwise, the string units of the result is determined by the data type of the first argument.

- OCTETS, if the first argument is character string or a graphic string with string units of OCTETS, CODEUNITS16, or double bytes.
- CODEUNITS32, if the first argument is character string or a graphic string with string units of CODEUNITS32.
- Determined by the default string unit of the environment, if the first argument is not a character string or a graphic string.

In a Unicode database, when the output string is truncated part-way through a multiple-byte character:

- If the input was a character string, the partial character is replaced with one or more blanks
- If the input was a graphic string, the partial character is replaced by the empty string

Do not rely on either of these behaviors, because they might change in a future release.

If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

Notes

- **Increasing portability of applications:** If the first argument is numeric, or if the first argument is a string and the length argument is specified, use the “CAST specification” on page 152 instead of this function to increase the portability of your applications.

Examples

- *Example 1:* Make EMPNO varying-length with a length of 10.

```
SELECT VARCHAR(EMPNO,10)
  INTO :VARHV
  FROM EMPLOYEE
```

- *Example 2:* Set the host variable JOB_DESC, defined as VARCHAR(8), to the VARCHAR equivalent of the job description (which is the value of the JOB column), defined as CHAR(8), for employee Dolores Quintana.

```
SELECT VARCHAR(JOB)
  INTO :JOB_DESC
  FROM EMPLOYEE
  WHERE LASTNAME = 'QUINTANA'
```

- *Example 3:* The EDLEVEL column is defined as SMALLINT. The following returns the value as a varying-length character string.

```
SELECT VARCHAR(EDLEVEL)
  FROM EMPLOYEE
  WHERE LASTNAME = 'HAAS'
```

  Results in the value '18'.

- *Example 4:* The SALARY and COMM columns are defined as DECIMAL with a precision of 9 and a scale of 2. Return the total income for employee Haas using the comma decimal character.

```
SELECT VARCHAR(SALARY + COMM, ',')
  FROM EMPLOYEE
  WHERE LASTNAME = 'HAAS'
```

  Results in the value '56970,00'.

- *Example 5*: The following statement returns a string of data type VARCHAR with the value 'TRUE'.

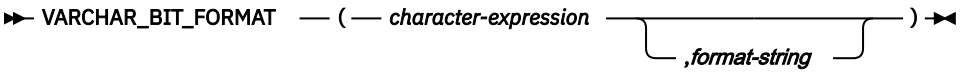
```
values VARCHAR(3=3)
```

- *Example 6*: The following statement returns a string of data type VARCHAR with the value 'FALSE'.

```
values VARCHAR(3>3)
```

VARCHAR_BIT_FORMAT

The VARCHAR_BIT_FORMAT function returns a bit string representation of a character string that has been formatted using a character template.



The schema is SYSIBM.

In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

character-expression
An expression that returns a value that is a built-in character string that is not a CLOB (SQLSTATE 42815). The required length is determined by the specified format string and how the value is interpreted. If a *format-string* argument is not specified, the length must be an even number of characters from the ranges '0' to '9', 'a' to 'f', and 'A' to 'F' (SQLSTATE 42815).

format-string
A character constant that contains a template for how the bytes of *character-expression* are to be interpreted.

Valid format strings include: `'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx'` and `'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX'` (SQLSTATE 42815) where each 'x' or 'X' corresponds to one hexadecimal digit in the result.

The result of the function is a varying-length character string FOR BIT DATA with the length attribute and actual length based on the format string. For the two valid format strings listed previously, the length attribute of the result is 36 and the actual length is 16. If a *format-string* argument is not specified, the length attribute of the result is half the length attribute of *character-expression* and the actual length is half the length of the actual length of *character-expression*. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Examples

- *Example 1*: Represent a Universal Unique Identifier in its binary form:

```
VARCHAR_BIT_FORMAT('d83d6360-1818-11db-9804-b622a1ef5492',
                   'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx')
```

  Result returned:

```
x'D83D6360181811DB9804B622A1EF5492'
```

- *Example 2*: Represent a Universal Unique Identifier in its binary form:

```
VARCHAR_BIT_FORMAT('D83D6360-1818-11DB-9804-B622A1EF5492',
                   'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX')
```

  Result returned:

```
    x'D83D6360181811DB9804B622A1EF5492'
```

- *Example 3*: Represent a string of hexadecimal characters in binary form.

```
VARCHAR_BIT_FORMAT('ef01abC9')
```

  Result returned as a VARCHAR(4) FOR BIT DATA value:

```
    x'EF01ABC9'
```

- *Example 4*: Represent a string of hexadecimal characters as a character string in the code page of the database. The result needs to be cast to a VARCHAR data type with the FOR MIXED DATA clause, assuming the database supports graphic types. Otherwise the result needs to be cast to a VARCHAR data type with the FOR SBCS clause. The following example assumes a Unicode database:

```
VALUES CAST(VARCHAR_BIT_FORMAT(HEX('abcdefg')) AS VARCHAR(10) FOR MIXED DATA)
```

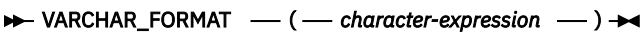
  Result returned:

```
abcdefg
```

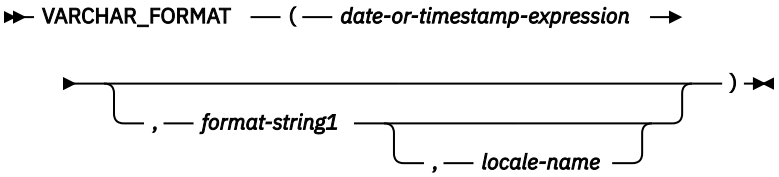
VARCHAR_FORMAT

The VARCHAR_FORMAT function returns a character representation of an input expression.

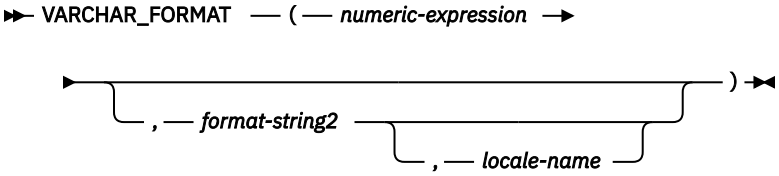
Character string to VARCHAR



DATE or TIMESTAMP to VARCHAR



Signed numeric to VARCHAR



The schema is SYSIBM.

If any argument of the VARCHAR_FORMAT function can be null, the result can be null; if any argument is null, the result is the null value.

Character string to VARCHAR

character-expression
An expression that returns a value that must be a built-in CHAR or VARCHAR data type. In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function.

The result is a VARCHAR with a length attribute that matches the length attribute of the argument. The value of the result is the same as the value of *character-expression*.

The code page of the result is the code page of the section.

DATE or TIMESTAMP to VARCHAR

date-or-timestamp-expression
An expression that returns a value that must be a DATE or TIMESTAMP, or a valid string representation of a date or timestamp that is not a CLOB or DBCLOB. In a Unicode database, if the expression returns a graphic string representation of a date or timestamp, the returned value is first converted to a character string before the function is evaluated.

If the input expression returns:

- A string, the *format-string* argument must also be specified.
- A DATE or a string representation of a date, the returned value is first converted to a TIMESTAMP(0) value with a time component of exactly midnight (00.00.00).
- A string representation of a timestamp, it is first converted to a TIMESTAMP(12) value

For a list of valid formats for string representations of datetime values, see "String representations of datetime values" on page 39.

format-string1
An expression that returns a value with one of the following built-in data types:

- CHAR or VARCHAR
- Signed numeric
- Datetime
- GRAPHIC or VARGRAPHIC (Unicode databases only)

If the data type is not CHAR or VARCHAR, it is implicitly cast to VARCHAR before the function is evaluated. The actual length must not be greater than 255 bytes (SQLSTATE 22007). The value is a template for how *timestamp-expression* is to be formatted.

The default format string is based on the value of the special register CURRENT LOCALE LC_TIME.

A valid format string must contain a combination of the format elements listed in Table 1 (SQLSTATE 22007). Two format elements can optionally be separated by one or more of the following separator characters:

- minus sign (-)
- period (.)
- slash (/)
- comma (,)
- apostrophe (')
- semi-colon (;)
- colon (:)
- blank ( )

Separator characters can also be specified at the start or end of *format-string*.

Table 104. Format elements for DATE or TIMESTAMP to VARCHAR	
Format element	**Description**
AM or PM	Meridian indicator (morning or evening) without periods. This format element is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
A.M. or P.M.	Meridian indicator (morning or evening) with periods. This format element uses the exact strings 'A.M.' or 'P.M.' and is independent of the locale name in effect.
CC	Century (01-99). If the last two digits of the four-digit year are zero, the result is the first two digits of the year; otherwise, the result is the first two digits of the year plus one.
DAY, Day, or day	Name of the day in uppercase, titlecase, or lowercase format. The language used is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
DY, Dy, or dy	Abbreviated name of the day in uppercase, titlecase, or lowercase format. The language used is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
D	Day of the week (1-7). The first day of the week is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
DD	Day of month (01-31).
DDD	Day of year (001-366).
FF or FF*n*	Fractional seconds (0-999999999), where *n* specifies the scale of the returned value. Valid values for *n* are 1 - 12 with no leading zeros. Specifying FF is equivalent to specifying FF6. If the scale of the input timestamp is less than *n*, the result is padded with trailing zeros.
HH	HH behaves the same as HH12.
HH12	Hour of the day (01-12) in 12-hour format.
HH24	Hour of the day (00-24) in 24-hour format.
I	ISO year (0-9). The last digit of the year based on the ISO week that is returned.
ID	ISO day of the week (1-7). 1 is Monday and 7 is Sunday.

Table 104. Format elements for DATE or TIMESTAMP to VARCHAR (continued)	
Format element	**Description**
IW	ISO week of the year (01-53). The week starts on Monday and includes seven days. Week 1 is the first week of the year to contain a Thursday, which is equivalent to the first week of the year to contain January 4.
IY	ISO year (00-99). The last two digits of the year based on the ISO week that is returned.
IYY	ISO year (000-999). The last three digits of the year based on the ISO week that is returned.
IYYY	ISO year (0000-9999). The 4-digit year based on the ISO week that is returned.
J	Julian day (number of days since January 1, 4713 BC).
MI	Minute (00-59).
MM	Month (01-12).
MONTH, Month, or month	Name of the month in uppercase, titlecase, or lowercase format. The language used is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
MON, Mon, or mon	Abbreviated name of the month in uppercase, titlecase, or lowercase format. The language used is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
MS	Milleseconds (000-999). Same as FF3.
NNNNNN	Microseconds (000000-999999). Same as FF6.
Q	Quarter (1-4), where the months January through March return 1.
RR	RR behaves the same as YY.
RRRR	RRRR behaves the same as YYYY.
SS	Seconds (00-59).
SSSSS	Seconds since previous midnight (00000-86400).
US	Microseconds (000000-999999). Same as FF6.
W	Week of the month (1-5), where week 1 starts on the first day of the month and ends on the seventh day.
WW	Week of the year (01-53), where week 1 starts on January 1 and ends on January 7.
Y	Last digit of the year (0-9).
YY	Last two digits of the year (00-99).

Table 104. Format elements for DATE or TIMESTAMP to VARCHAR (continued)	
Format element	**Description**
YYY	Last three digits of the year (000-999).
YYYY	4-digit year (0000-9999).

The format elements in Table 104 on page 566 are not case sensitive, with the following exceptions:

- AM, PM
- A.M., P.M.
- DAY, Day, day
- DY, Dy, dy
- D
- MONTH, Month, month
- MON, Mon, mon

In cases where format elements are ambiguous, the case insensitive format elements will be considered first. For example, 'DDYYYY' would be interpreted as "DD followed by YYYY", not "D followed by DY followed by YYY".

locale-name
A character constant that specifies the locale used for the following format elements:

- AM, PM
- DAY, Day, day
- DY, Dy, dy
- D
- MONTH, Month, month
- MON, Mon, mon

The specified locale name is not case sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming,, see "Locale names for SQL and XQuery" in the *Globalization Guide* . The default is the value of the CURRENT LOCALE LC_TIME special register.

The result is a representation of the input timestamp expression in the format specified by the format string. The format string is interpreted as a series of format elements that can be separated by one or more separator characters. A string of characters in the format string is interpreted as the longest matching format element in Table 104 on page 566. If two format elements that contain the same characters are not delimited by a separator character, the specification is interpreted, starting from the left, as the longest matching format element in the table, and continues until matches are found for the remainder of the format string. For example, 'YYYYYYDD' is interpreted as the format elements 'YYYY', 'YY', and 'DD'.

The result is a varying-length character string. The length attribute is 255. If the string units of the environment or *format-string* is CODEUNITS32, the string units of the result is CODEUNITS32. Otherwise, the string units of the result is OCTETS. The *format-string* determines the actual length of the result. If the resulting string exceeds the length attribute of the result, the result is truncated.

The code page of the result is the code page of the section.

Signed numeric to VARCHAR

numeric-expression
An expression that returns a value of any built-in signed numeric data type. If the data type of the value is not DECFLOAT, it is converted to DECFLOAT(34) for processing.

format-string2
An expression that returns a value that has one of the following built-in data types:

- CHAR or VARCHAR
- Signed numeric
- Datetime
- GRAPHIC or VARGRAPHIC (Unicode databases only)

If the data type is not CHAR or VARCHAR, it is implicitly cast to VARCHAR before the function is evaluated. The actual length cannot be greater than 255 bytes (SQLSTATE 22018). The value is used as a template to format the input decimal floating-point expression. Format elements specified as a prefix can be used only at the beginning of the template. Format elements specified as a suffix can be used only at the end of the template. The template cannot contain more than one of the MI, S, or PR format elements (SQLSTATE 22018).

If a format string is not specified, the function is equivalent to VARCHAR(DECFLOAT(numeric-expression)).

Table 105. Format elements for decimal floating-point to varchar

Format element	**Description**
0	Each 0 represents a significant digit. Leading zeros in a number are displayed as zeros.
9	Each 9 represents a significant digit. Leading zeros in a number are displayed as blanks.
PL or pl	If the input decimal floating-point expression returns a positive number, a plus sign (+) is added at the specified position.
G or g	The group separator specified by the locale is added at the specified position.
D or d	The decimal delimiter specified by the locale is added at the specified position.
,	A comma is added at the specified position, for example as a group separator.
.	A period is added at the specified position, for example as a decimal point.
S or s	Prefix: If the input decimal floating-point expression returns: • A negative number, a leading minus sign (-) is added to the result • A positive number, a leading plus sign (+) is added to the result
$	Prefix: A leading dollar sign ($) is added to the result.
MI or mi	Suffix: If the input decimal floating-point expression returns: • A negative number, a trailing minus sign (-) is added to the result • A positive number, a trailing blank is added to the result
PR or pr	Suffix: If the input decimal floating-point expression returns: • A negative number, a leading less than character (<) and a trailing greater than character (>) are added to the result • A positive number, a leading space and a trailing space are added to the result

The format elements are case sensitive.

locale-name
A character constant that specifies the locale used to determine the group separator and decimal delimiter.

The specified locale name is not case sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming,, see "Locale names for SQL and XQuery" in the *Globalization Guide* . The default is the value of the CURRENT LOCALE LC_TIME special register.

The result is a varying-length character string representation of the input decimal floating-point expression. If a single argument is specified the length attribute is 42. Otherwise the length attribute is 254. If the string units of the environment or the format string is CODEUNITS32, the string units of the result is CODEUNITS32; otherwise, the string units of the result is OCTETS. The actual length of the result is determined by the format string, if specified; otherwise, the actual length of the result is the smallest number of characters that can represent the value of the input decimal floating-point expression. If the resulting string exceeds the length attribute of the result, the result is truncated.

If the value of the input decimal floating-point expression is the special value:

- Infinity, the string "INFINITY" is returned
- sNaN, the string "SNAN" is returned
- NaN, the string "NAN" is returned

If the special value is negative, the first character of the result is a minus sign (-). The decimal floating-point special value sNaN does not result in an exception when converted to a string.

If the format string does not include any of the format elements MI, S, or PR, and if the value of the input expression is negative, then a minus sign (-) is included in the result; otherwise, a blank is included in the result.

If the number of digits to the left of the decimal point in the input expression is greater than the number of digits to the left of the decimal point in the format string, the result is a string of one or more number sign (#) characters. If the number of digits to the right of the decimal point in the input expression is greater than the number of digits to the right of the decimal point in the format string, the result is rounded to the number of digits to the right of the decimal point in the format string. The DECFLOAT rounding mode will not be used. The rounding behavior of VARCHAR_FORMAT corresponds to a value of ROUND_HALF_UP.

The code page of the result is the code page of the section.

Notes

- ***Julian and Gregorian calendar***: For Timestamp to varchar, the transition from the Julian calendar to the Gregorian calendar on 15 October 1582 is taken into account by this function.
- ***Determinism:*** VARCHAR_FORMAT is a deterministic function. However, the following invocations of the function depend on the value of the special register CURRENT LOCALE LC_TIME.
  - Timestamp to varchar, when format-string is not explicitly specified, or when locale-name is not explicitly specified and one of the following statements is true:
    - *format-string* is not a constant
    - *format-string* is a constant and includes format elements that are locale sensitive

    These invocations that depend on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621, 428EC, or 429BX).
- ***Syntax alternatives:*** TO_CHAR is a synonym for VARCHAR_FORMAT.

Examples

- *Example 1*: Display the names and creation timestamps for all system tables that have names that start with **SYSU**.

```
SELECT VARCHAR(TABNAME, 20) AS TABLE_NAME,
  VARCHAR_FORMAT(CREATE_TIME, 'YYYY-MM-DD HH24:MI:SS')
    AS CREATION_TIME
  FROM SYSCAT.TABLES
  WHERE TABNAME LIKE 'SYSU%'
```

This example returns the following output:

```
TABLE_NAME           CREATION_TIME
-------------------- --------------------
SYSUSERAUTH          2000-05-19 08:18:56
SYSUSEROPTIONS       2000-05-19 08:18:56
```

- *Example 2*: The variable TMSTMP is defined as a TIMESTAMP and has the value 2007-03-09-14.07.38.123456. The following examples show invocations of the VARCHAR_FORMAT function and the resulting string values. The data type of each result is VARCHAR(255).

```
Function invocation                          Result
-------------------                          ------
VARCHAR_FORMAT(TMSTMP,'YYYYMMDDHHMISSFF3')   20070309020738123
VARCHAR_FORMAT(TMSTMP,'YYYYMMDDHH24MISS')    20070309140738
VARCHAR_FORMAT(TMSTMP,'YYYYMMDDHHMI')        200703090207
```

```
VARCHAR_FORMAT(TMSTMP,'HH12:MI:SS.MS AM')    02:07:38.123 PM
VARCHAR_FORMAT(TMSTMP,'HH24:MI:SS.US')       14:07:38.123456
```

```
VARCHAR_FORMAT(TMSTMP,'DD/MM/YY')            09/03/07
VARCHAR_FORMAT(TMSTMP,'MM-DD-YYYY')          03-09-2007
VARCHAR_FORMAT(TMSTMP,'J')                   2454169
VARCHAR_FORMAT(TMSTMP,'Q')                   1
VARCHAR_FORMAT(TMSTMP,'W')                   2
VARCHAR_FORMAT(TMSTMP,'IW')                  10
VARCHAR_FORMAT(TMSTMP,'WW')                  10
VARCHAR_FORMAT(TMSTMP,'Month','en_US')       March
VARCHAR_FORMAT(TMSTMP,'MONTH','en_US')       MARCH
VARCHAR_FORMAT(TMSTMP,'MON','en_US')         MAR
VARCHAR_FORMAT(TMSTMP,'Day','en_US')         Friday
VARCHAR_FORMAT(TMSTMP,'DAY','en_US')         FRIDAY
VARCHAR_FORMAT(TMSTMP,'Dy','en_US')          Fri
VARCHAR_FORMAT(TMSTMP,'Month','de_DE')       März
VARCHAR_FORMAT(TMSTMP,'MONTH','de_DE')       MÄRZ
VARCHAR_FORMAT(TMSTMP,'MON','de_DE')         MÄRZ
VARCHAR_FORMAT(TMSTMP,'Day','de_DE')         Freitag
VARCHAR_FORMAT(TMSTMP,'DAY','de_DE')         FREITAG
VARCHAR_FORMAT(TMSTMP,'Dy','de_DE')          Fr.
```

- *Example 3*: The variable DTE is defined as a DATE and has the following value: 2007-03-09. The following examples show several invocations of the function and the resulting string values. The data type of each result is VARCHAR(255).

```
Function invocation                          Result
-------------------                          ------
VARCHAR_FORMAT(DTE,'YYYYMMDD')               20070309
VARCHAR_FORMAT(DTE,'YYYYMMDDHH24MISS')       20070309000000
```

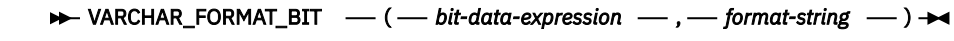
- *Example 4*: The variables POSNUM and NEGNUM are both defined as DECFLOAT(34), and the value of POSNUM is 1234.56 and the value of NEGNUM is -1234.56. The following examples show several invocations of the VARCHAR_FORMAT and the resulting string values. The data type of the first two results is VARCHAR(42) and the rest are VARCHAR(254).

```
Function invocation                           Result
-------------------                           ------
VARCHAR_FORMAT(POSNUM)                        '1234.56'
VARCHAR_FORMAT(NEGNUM)                        '-1234.56'
VARCHAR_FORMAT(POSNUM,'9999.99')              ' 1234.56'
VARCHAR_FORMAT(NEGNUM,'9999.99')              '-1234.56'
VARCHAR_FORMAT(POSNUM,'99999.99')             '  1234.56'
VARCHAR_FORMAT(NEGNUM,'99999.99')             ' -1234.56'
VARCHAR_FORMAT(POSNUM,'00000.00')             ' 01234.56'
VARCHAR_FORMAT(NEGNUM,'00000.00')             '-01234.56'
VARCHAR_FORMAT(POSNUM,'9999.99MI')            '1234.56 '
VARCHAR_FORMAT(NEGNUM,'9999.99MI')            '1234.56-'
VARCHAR_FORMAT(POSNUM,'S9999.99')             '+1234.56'
VARCHAR_FORMAT(NEGNUM,'S9999.99')             '-1234.56'
VARCHAR_FORMAT(POSNUM,'9999.99PR')            ' 1234.56 '
VARCHAR_FORMAT(NEGNUM,'9999.99PR')            '<1234.56>'
VARCHAR_FORMAT(POSNUM,'S$9,999.99')           '+$1,234.56'
VARCHAR_FORMAT(NEGNUM,'S$9,999.99')           '-$1,234.56'
```

```
VARCHAR_FORMAT(POSNUM,'99,99,99')             '    12,35'
VARCHAR_FORMAT(NEGNUM,'99,99,99')             '   -12,35'
VARCHAR_FORMAT(POSNUM,'PL9999.99')            '+1234.56'
VARCHAR_FORMAT(NEGNUM,'PL9999.99')            ' 1234.56'
VARCHAR_FORMAT(POSNUM,'9999PL')               ' 1234+'
VARCHAR_FORMAT(NEGNUM,'9999PL')               '-1234 '
VARCHAR_FORMAT(POSNUM,'9999.9')               ' 1234.6'
VARCHAR_FORMAT(NEGNUM,'9999.9')               '-1234.6'
VARCHAR_FORMAT(POSNUM,'9999')                 ' 1235'
VARCHAR_FORMAT(NEGNUM,'9999')                 '-1235'
VARCHAR_FORMAT(POSNUM,'99.99')                '######'
VARCHAR_FORMAT(NEGNUM,'99.99')                '######'
VARCHAR_FORMAT(POSNUM,'9999D99', 'en_US')     ' 1234.56'
VARCHAR_FORMAT(POSNUM,'9999D99', 'fr_FR')     ' 1234,56'
VARCHAR_FORMAT(POSNUM,'9G999D99', 'en_US')    ' 1,234.56'
VARCHAR_FORMAT(POSNUM,'9G999D99', 'de_DE')    ' 1.234,56'
```

VARCHAR_FORMAT_BIT

The VARCHAR_FORMAT_BIT function returns a character representation of a bit string that has been formatted using a character template.



The schema is SYSIBM.

bit-data-expression
An expression that returns a value that is a built-in character-string FOR BIT DATA data type (SQLSTATE 42815). The required length is determined by the specified format string and how the value is interpreted.

format-string
A character constant that contains a template for how the result is to be formatted.

Valid format strings include: `'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx'` and `'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX'` (SQLSTATE 42815) where each 'x' or 'X' corresponds to one hexadecimal digit from *bit-data-expression*.

The result of the function is a varying-length character string with the length attribute and actual length based on the format string. If the string units of the environment or *format-string* is CODEUNITS32, the string units of the result is CODEUNITS32. Otherwise, the string units of the result is OCTETS For the two valid format strings listed previously, the length attribute is 36 and the actual length is 36 bytes. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Examples

- *Example 1*: Represent a Universal Unique Identifier in its formatted form:

```
VARCHAR_FORMAT_BIT(CAST(x'd83d6360181811db9804b622a1ef5492'
                        AS VARCHAR(16) FOR BIT DATA),
                   'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx')
```

  Result returned:

```
'd83d6360-1818-11db-9804-b622a1ef5492'
```

- *Example 2*: Represent a Universal Unique Identifier in its formatted form:

```
VARCHAR_FORMAT_BIT(CAST(x'd83d6360181811db9804b622a1ef5492'
                        AS CHAR(16) FOR BIT DATA),
                   'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX')
```

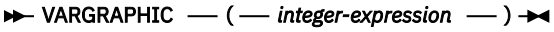
  Result returned:

```
'D83D6360-1818-11DB-9804-B622A1EF5492'
```

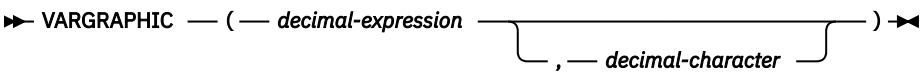
VARGRAPHIC

The VARGRAPHIC function returns a varying-length graphic string representation of a value of a different data type.

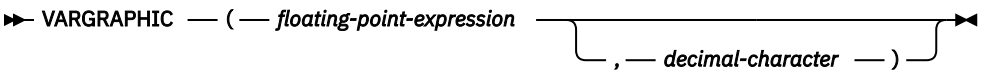
Integer to vargraphic



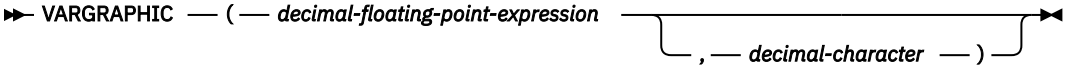
Decimal to vargraphic



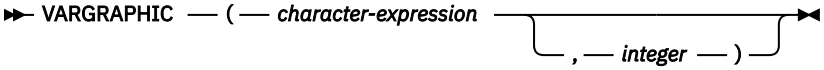
Floating-point to vargraphic



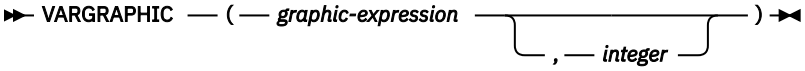
Decimal floating-point to vargraphic



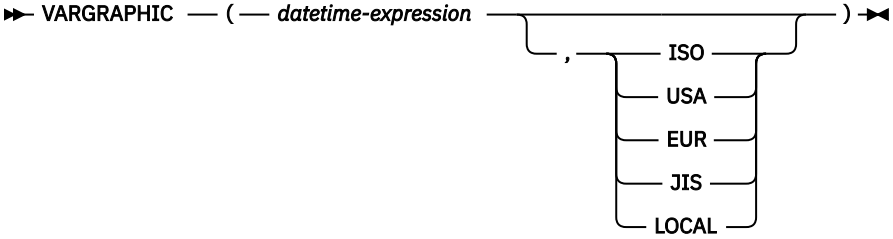
Character to vargraphic



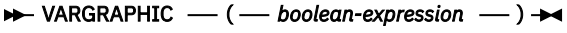
Graphic to vargraphic



Datetime to vargraphic



Boolean to vargraphic



The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature.

Integer to vargraphic

integer-expression
An expression that returns a value that is of an integer data type (SMALLINT, INTEGER, or BIGINT).

The result is the varying-length graphic string representation of *integer-expression* in the form of an SQL integer constant. The length attribute of the result depends on whether *integer-expression* is a small, large or big integer as follows:

- If the first argument is a small integer, the maximum length of the result is 6.
- If the first argument is a large integer, the maximum length of the result is 11.
- If the first argument is a big integer, the maximum length of the result is 20.

The actual length of the result is the smallest number of double-byte characters that can be used to represent the value of the argument. Leading zeros are not included. If the argument is negative, the first double-byte character of the result is a minus sign; otherwise, the first double-byte character is a digit.

The code page of the result is the DBCS code page of the section.

Decimal to vargraphic

decimal-expression
An expression that returns a value that is a decimal data type. The DECIMAL scalar function can be used to change the precision and scale.

decimal-character
Specifies the double-byte character constant that is used to delimit the decimal digits in the result graphic string. The double-byte character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a varying-length graphic string representation of *decimal-expression* in the form of an SQL decimal constant. The length attribute of the result is 2+*p*, where *p* is the precision of *decimal-expression*. The actual length of the result is the smallest number of double-byte characters that can be used to represent the result, except that trailing zeros are included. Leading zeros are not included. If *decimal-expression* is negative, the first double-byte character of the result is a minus sign; otherwise, the first double-byte character is a digit or the decimal character. If the scale of *decimal-expression* is zero, the decimal character is not returned.

The code page of the result is the DBCS code page of the section.

Floating-point to vargraphic

floating-point-expression
An expression that returns a value that is a floating-point data type (DOUBLE or REAL).

decimal-character
Specifies the double-byte character constant that is used to delimit the decimal digits in the result graphic string. The double-byte character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a varying-length graphic string representation of *floating-point-expression* in the form of an SQL floating-point constant.

The maximum length of the result is 24. The actual length of the result is the smallest number of double-byte characters that can represent the value of *floating-point-expression* such that the mantissa consists of a single digit other than zero followed by the *decimal-character* and a sequence of digits. If *floating-point-expression* is negative, the first double-byte character of the result is a minus sign; otherwise, the first double-byte character is a digit. If *floating-point-expression* is zero, the result is 0E0.

The code page of the result is the DBCS code page of the section.

Decimal floating-point to vargraphic

decimal-floating-point-expression
An expression that returns a value that is a decimal floating-point data type (DECFLOAT).

decimal-character
Specifies the double-byte character constant that is used to delimit the decimal digits in the result graphic string. The double-byte character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a varying-length graphic string representation of *decimal-floating-point-expression* in the form of an SQL decimal floating-point constant. The maximum length of the result is 42. The actual length of the result is the smallest number of double-byte characters that can represent the value of *decimal-floating-point-expression*. If *decimal-floating-point-expression* is negative, the first double-byte character of the result is a minus sign; otherwise, the first double-byte character is a digit. If *decimal-floating-point-expression* is zero, the result is 0.

If the value of *decimal-floating-point-expression* is the special value Infinity, sNaN, or NaN, the strings G'INFINITY', G'SNAN', and G'NAN', respectively, are returned. If the special value is negative, the first double-byte character of the result is a minus sign. The decimal floating-point special value sNaN does not result in a warning when converted to a string.

The code page of the result is the DBCS code page of the section.

Character to vargraphic
In Unicode databases:

character-expression
An expression that returns a value that is a built-in character string data type. The expression must not be a FOR BIT DATA subtype (SQLSTATE 42846).

integer
An integer constant that specifies the length attribute for the resulting varying-length graphic string. The value must be between 0 and the maximum length for the VARGRAPHIC data type in the string units of the result.

If the second argument is not specified:

- If the *character-expression* is the empty string constant, the length attribute of the result is 0.
- Otherwise, the length attribute of the result is the lower of the following values:
  - The maximum length for the VARGRAPHIC data type in the string units of the result.
  - The length attribute of the first argument.

The result is a varying-length graphic string that is converted from *character-expression*. The length attribute of the result is determined by the value of *integer*.

If the length of *character-expression* that is converted to a graphic string is greater than the length attribute of the result, several scenarios exist:

- If the string unit of the result is CODEUNITS32, truncation is performed. If only blank characters are truncated and *character-expression* is CHAR or VARCHAR, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004).
- If *integer* is specified, truncation is performed with a warning returned (SQLSTATE 01004). When the output string is truncated, such that the last character is a high surrogate, that surrogate is deleted. Do not rely on this behavior because it might change in a future release.
- If *integer* is not specified and *character-expression* is a VARCHAR, truncation is performed with a warning returned (SQLSTATE 01004).
- If *integer* is not specified and character-expression is a CLOB, an error is returned (SQLSTATE 22001).

This function converts *character-expression* from UTF-8 to UTF-16. Every character of *character-expression* is converted.

In non-Unicode databases:

character-expression
An expression that returns a value that is a built-in CHAR or VARCHAR data type.

The result is a varying-length graphic string that is converted from *character-expression*. The length attribute of the result is the minimum of the length attribute of *character-expression* and the maximum length for the VARGRAPHIC data type.

If the length of *character-expression* that is converted to a graphic string is greater than the length attribute of the result, an error is returned (SQLSTATE 22001).

For databases with a code set that is not Japanese EUC (code page 954) or Traditional Chinese (code page 964), each single-byte character in *character-expression* is converted to its equivalent double-byte representation or to the double-byte substitution character in the result. Each double-byte character in *character-expression* is mapped without extra conversion. If the first byte of a double-byte character appears as the last byte of *character-expression*, it is converted to the double-byte substitution character. The sequential order of the characters in *character-expression* is preserved. No warning or error code is generated if one or more double-byte substitution characters are returned in the result.

For details about the conversion process for databases with a code set that is Japanese EUC (code page 954) or Traditional Chinese (code page 964), refer to the topic "Japanese and traditional-Chinese extended UNIX code (EUC) considerations" in Globalization Guide.

Graphic to vargraphic

graphic-expression
An expression that returns a value that is a built-in graphic string data type.

integer
An integer constant that specifies the length attribute for the resulting varying-length graphic string. The value must be between 0 and the maximum length for the VARGRAPHIC data type in the string units of the result. If the second argument is not specified:

- If the *graphic-expression* is the empty string constant, the length attribute of the result is 0.
- Otherwise, the length attribute of the result is the lower of the following values:
  - The maximum length for the VARGRAPHIC data type in the string units of the result
  - The length attribute of the first argument

The result is a varying-length graphic string. The length attribute of the result is determined by the value of *integer*.

If the length of *graphic-expression* is greater than the length attribute of the result, several scenarios exist:

- If the string unit of the result is CODEUNITS32, truncation is performed. If only blank characters are truncated and *graphic-expression* is GRAPHIC or VARGRAPHIC, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004).
- If integer is specified, truncation is performed. If only blank characters are truncated and *graphic-expression* is GRAPHIC or VARGRAPHIC, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004). In a Unicode database, when the output string is truncated, such that the last character is a high surrogate, that surrogate is converted to the blank character (X'0020'). Do not rely on this behavior, because it might change in a future release.
- If *integer* is not specified, an error is returned (SQLSTATE 22001).

Datetime to vargraphic

datetime-expression
An expression that is one of the following data types:

DATE
The result is the graphic string representation of the date in the format specified by the second argument. The length of the result is 10. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

TIME
The result is the graphic string representation of the time in the format specified by the second argument. The length of the result is 8. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

TIMESTAMP
The result is the graphic string representation of the timestamp. If the data type of *datetime-expression* is TIMESTAMP(0), the length of the result is 19. If the data type of *datetime-expression* is TIMESTAMP(*n*), where *n* is between 1 and 12, the length of the result is 20+*n*. Otherwise, the length of the result is 26. The second argument must not be specified (SQLSTATE 42815).

The code page of the result is the DBCS code page of the section.

Boolean to vargraphic

boolean-expression
An expression that returns a Boolean value (TRUE or FALSE). The result is either 'TRUE' or 'FALSE'.

Result

The VARGRAPHIC function returns a varying-length graphic string representation of:

- An integer number (Unicode database only), if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number (Unicode database only), if the first argument is a decimal number
- A double-precision floating-point number (Unicode database only), if the first argument is a DOUBLE or REAL
- A decimal floating-point number (Unicode database only), if the first argument is a decimal floating-point number (DECFLOAT)
- A character string, if the first argument is any type of character string
- A graphic string, if the first argument is any type of graphic string
- A datetime value (Unicode databases only), if the first argument is a DATE, TIME, or TIMESTAMP
- A Boolean value (TRUE or FALSE)

In a non-Unicode database, the string units of the result is double bytes. Otherwise, the string units of the result is determined by the data type of the first argument.

- CODEUNITS16, if the first argument is character string or a graphic string with string units of OCTETS or CODEUNITS16.
- CODEUNITS32, if the first argument is character string or a graphic string with string units of CODEUNITS32.
- Determined by the default string unit of the environment, if the first argument is not a character string or a graphic string.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Notes

- **Increasing portability of applications:** If the first argument is numeric, or if the first argument is a string and the length argument is specified, use the "CAST specification" on page 152 instead of this function to increase the portability of your applications.

Examples

- *Example 1*: The EDLEVEL column is defined as SMALLINT. The following statement returns the value as a varying-length graphic string.

```
SELECT VARGRAPHIC(EDLEVEL)
  FROM EMPLOYEE
  WHERE LASTNAME = 'HAAS'
```

  Results in the value G'18'.

- *Example 2*: The SALARY and COMM columns are defined as DECIMAL with a precision of 9 and a scale of 2. Return the total income for employee Haas using the comma decimal character.

```
SELECT VARGRAPHIC(SALARY + COMM, ',')
  FROM EMPLOYEE
  WHERE LASTNAME = 'HAAS'
```

  Results in the value G'56970,00'.

- *Example 3*: The following statement returns a string of data type VARGRAPHIC with the value 'TRUE'.

```
values VARGRAPHIC(3=3)
```

- *Example 4*: The following statement returns a string of data type VARGRAPHIC with the value 'FALSE'.

```
values VARGRAPHIC(3>3)
```

VERIFY_GROUP_FOR_USER

The VERIFY_GROUP_FOR_USER function returns a value that indicates whether any of the groups associated with the authorization ID identified by the SESSION_USER special register are in the group names specified by the list of *group-name-expression* arguments.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

group-name-expression
An expression that specifies an authorization name (SQLSTATE 42815). The existence of the authorization name at the current server is not verified. *group-name-expression* must return a built-in character string data type or graphic string data type that is not a LOB (SQLSTATE 42815). The content of the string is not folded to uppercase and is not left-aligned.

The result of the function is an integer. The result cannot be null. The result is 1 if any of the groups associated with the authorization ID identified by the SESSION_USER special register are in the list of *group-name-expression* arguments. Otherwise, the result is 0.

Example 1

The tellers in a bank can only access customers from their own branch. All tellers are members in the group TELLER. A row permission is created by a user with SECADM authority to enforce this rule.

```
CREATE PERMISSION TELLER_ROW_ACCESS ON CUSTOMER
  FOR ROWS WHERE VERIFY_GROUP_FOR_USER(SESSION_USER,'TELLER') = 1 AND
      BRANCH = (SELECT HOME_BRANCH FROM INTERNAL_INFO
          WHERE EMP_ID = USER)
ENFORCED FOR ALL ACCESS
ENABLE
```

Example 2

The determination on whether the user can see the row is determined by the name of the role in the ACCESS_ROLE column of the table being protected. A row permission is created by a user with the SECADM authority to check the session user is in this role.

```
CREATE PERMISSION ROLEACCESS ON CUSTOMER
FOR ROWS WHERE ( VERIFY_GROUP_FOR_USER(SESSION_USER, ACCESS_ROLE ) = 1 )
ENFORCED FOR ALL ACCESS
ENABLE
```

VERIFY_ROLE_FOR_USER

The VERIFY_ROLE_FOR_USER function returns a value that indicates whether any of the roles associated with the authorization ID identified by the SESSION_USER special register are in (or contain any of) the role names specified by the list of *role-name-expression* arguments.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

role-name-expression

An expression that specifies a role name (SQLSTATE 42815). The existence of the role name at the current server is not verified. *role-name-expression* must return a built-in character string data type or graphic string data type that is not a LOB (SQLSTATE 42815). The content of the string is not folded to uppercase and is not left-aligned.

The result of the function is an integer. The result cannot be null. The result is 1 if any of the roles associated with the authorization ID identified by the SESSION_USER special register are in (or contain any of) the role names specified by the list of *role-name-expression* arguments. Otherwise, the result is 0.

Example 1

The tellers in a bank can only access customers from their own branch. All tellers are members in the role TELLER. A row permission is created by a user with the SECADM authority to enforce this rule.

```
CREATE PERMISSION TELLER_ROW_ACCESS ON CUSTOMER
  FOR ROWS WHERE VERIFY_ROLE_FOR_USER(SESSION_USER,'TELLER') = 1 AND
      BRANCH = (SELECT HOME_BRANCH FROM INTERNAL_INFO
          WHERE EMP_ID = USER)
ENFORCED FOR ALL ACCESS
ENABLE
```

Example 2

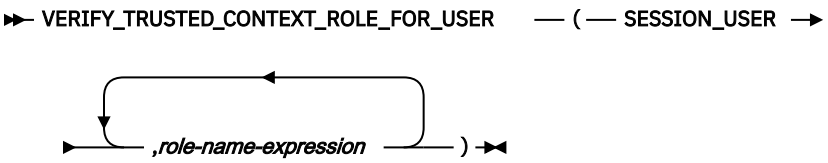
The determination on whether the user can see the row is determined by the name of the role in the ACCESS_ROLE column of the table being protected. A row permission is created by a user with the SECADM authority to check the session user is in this role.

```
CREATE PERMISSION ROLEACCESS ON CUSTOMER
FOR ROWS WHERE (VERIFY_ROLE_FOR_USER(SESSION_USER, ACCESS_ROLE) = 1)
ENFORCED FOR ALL ACCESS
ENABLE
```

VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER

The VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER function returns a value that indicates whether the authorization ID identified by the SESSION_USER special register has acquired a role under a trusted

connection associated with some trusted context and that role is in (or part of) the role names specified by the list of *role-name-expression* arguments.



The schema is SYSIBM. The function name cannot be specified as a qualified name.

role-name-expression
An expression that specifies a role name (SQLSTATE 42815). The existence of the role name at the current server is not verified. *role-name-expression* must return a built-in character string data type or graphic string data type that is not a LOB (SQLSTATE 42815). The content of the string is not folded to uppercase and is not left-aligned.

The result of the function is an integer. The result cannot be null. The result is 1 if the authorization ID identified by the SESSION_USER special register has acquired a role under a trusted connection associated with some trusted context and that role is in (or contained in any of) the role names specified by the list of *role-name-expression* arguments. Otherwise, the result is 0.

Example 1

The tellers in a bank can only access customers from their own branch. All tellers are members in the role TELLER, which can only be acquired through a trusted connection. A row permission is created by a user with SECADM authority to enforce this rule.

```
CREATE PERMISSION TELLER_ROW_ACCESS ON CUSTOMER
  FOR ROWS WHERE
     VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER(SESSION_USER,'TELLER') = 1 AND
         BRANCH = (SELECT HOME_BRANCH FROM INTERNAL_INFO
             WHERE EMP_ID = USER)
ENFORCED FOR ALL ACCESS
ENABLE
```

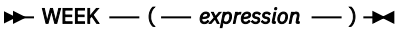
Example 2

The determination on whether the user can see the row is determined by the name of the role in the ACCESS_ROLE column of the table being protected. A row permission is created by a user with the SECADM authority to check the session user is in this role.

```
CREATE PERMISSION ROLEACCESS ON CUSTOMER
FOR ROWS WHERE ( VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER(SESSION_USER, ACCESS_ROLE ) = 1 )
ENFORCED FOR ALL ACCESS
ENABLE
```

WEEK

The WEEK scalar function returns the week of the year of the argument as an integer value in range 1-54. The week starts with Sunday.



The schema is SYSFUN.

expression
An expression that returns a value of one of the following built-in data types: DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

WEEK_ISO

The WEEK_ISO function returns the week of the year of the argument as an integer value in the range 1-53.

The schema is SYSFUN.

expression
An expression that returns a value of one of the following built-in data types: DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The week starts with Monday and always includes 7 days. Week 1 is the first week of the year to contain a Thursday, which is equivalent to the first week containing January 4. It is therefore possible to have up to 3 days at the beginning of a year appear in the last week of the previous year. Conversely, up to 3 days at the end of a year may appear in the first week of the next year.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

Example

The following list shows examples of the result of WEEK_ISO and DAYOFWEEK_ISO.

```
DATE       WEEK_ISO    DAYOFWEEK_ISO
---------- ----------- -------------
1997-12-28          52             7
1997-12-31           1             3
1998-01-01           1             4
1999-01-01          53             5
1999-01-04           1             1
1999-12-31          52             5
2000-01-01          52             6
2000-01-03           1             1
```

WEEKS_BETWEEN

The WEEKS_BETWEEN function returns the number of full weeks between the specified arguments.

The schema is SYSIBM.

expression1
An expression that specifies the first datetime value to compute the number of full weeks between two datetime values. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression1* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

expression2
An expression that specifies the second datetime value to compute the number of full weeks between two datetime values. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If

expression2 is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

If there is less than a full week between *expression1* and *expression2*, the result is zero. If *expression1* is later than *expression2*, the result is positive. If *expression1* is earlier than *expression2*, the result is negative. In NPS compatibility mode, this function always returns a positive number. If *expression1* or *expression2* contains time information, this information is also used to determine the number of full weeks. If *expression1* or *expression2* does not contain time information, a time of midnight (00.00.00) is used for the argument that is missing time information.

The result of the function is an INTEGER. If either argument can be null, the result can be null. If either argument is null, the result is the null value.

Examples

1. Set the host variable NUM_WEEKS with the number of full weeks between 2012-03-06 and 2012-02-28.

   ```
   SET :NUM_WEEKS = WEEKS_BETWEEN(DATE '2012-03-06', DATE '2012-02-28')
   ```

   The host variable NUM_WEEKS is set to 1 because there are 7 days between the arguments, including February 29, 2012.
2. Set the host variable NUM_WEEKS with the number of full weeks between 2012-03-05 and 2012-02-28.

   ```
   SET :NUM_WEEKS = WEEKS_BETWEEN(DATE '2012-03-05', DATE '2012-02-28')
   ```

   The host variable NUM_WEEKS is set to 0 because there are only 6 days between the arguments.
3. Set the host variable NUM_WEEKS with the number of full weeks between 2013-09-21-23.59.59 and 2013-09-01-00.00.00.

   ```
   SET :NUM_WEEKS = WEEKS_BETWEEN(TIMESTAMP '2013-09-21-23.59.59',
                                  TIMESTAMP '2013-09-01-00.00.00')
   ```

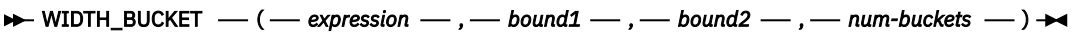
   The host variable NUM_WEEKS is set to 2 because there is 1 second less than 3 full weeks between the arguments. It is positive because the first argument is later than the second argument.
4. Set the host variable NUM_WEEKS with the number of full weeks between 2013-09-01-00.00.00 and 2013-09-21-23.59.59.

   ```
   SET :NUM_WEEKS = WEEKS_BETWEEN(TIMESTAMP '2013-09-01-00.00.00',
                                  TIMESTAMP '2013-09-21-23.59.59')
   ```

   The host variable NUM_WEEKS is set to -2 because there is 1 second less than 3 full weeks between the arguments. It is negative because the first argument is earlier than the second argument.

WIDTH_BUCKET

The WIDTH_BUCKET function is used to create equal-width histograms.



The schema is SYSIBM.

expression
An expression that specifies the value to be assigned into a bucket. The expression must return a value that is a built-in numeric, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If the data type is DECFLOAT, the value must not be a special value such as NaN or INFINITY (SQLSTATE 42815).

bound1
An expression that specifies the left end point. The expression must return a value that is a built-in numeric, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If the data type is DECFLOAT, the value must not be a special value such as NaN or INFINITY (SQLSTATE 42815).

bound2
An expression that specifies the right end point. The expression must return a value that is a built-in numeric, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If the data type is DECFLOAT, the value must not be a special value such as NaN or INFINITY (SQLSTATE 42815). *bound1* must not be equal to *bound2* (SQLSTATE 2201G).

num-buckets
An expression that specifies the number of buckets between *bound1* and *bound2*. The expression must return a value that is a SMALLINT, INTEGER, BIGINT, DECIMAL, DECFLOAT, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If the expression is a CHAR, VARCHAR, a GRAPHIC, or VARGRAPHIC, it is cast to DECFLOAT(34) before the function is evaluated. If the value is a DECIMAL or DECFLOAT, it is truncated to zero places to the left of the decimal point. The value must be greater than 0 (SQLSTATE 2201G). If the data type is DECFLOAT, the value must not be a special value such as NaN or INFINITY (SQLSTATE 42815).

The data type of the result is based on the data type of *num-buckets*.

Table 106. Data type of the result

Data type of *num-buckets*	**Data type of result**
SMALLINT	SMALLINT
INTEGER	INTEGER
BIGINT	BIGINT
DECIMAL(p,s)	DECIMAL(MIN(31, p-s+1), 0)
DECFLOAT(n)	DECFLOAT(n)

This function returns the bucket number that *expression* falls into given *bound1*, *bound2*, and *num-buckets*. The range from *bound1* to *bound2* is divided into *num-buckets* buckets starting from bucket 1 to bucket *num-buckets*.

If any argument can be null, the result can be null. If any argument is null, the result is the null value.

Notes

- If *bound1* is less than *bound2*, each bucket is a left-closed, right-open interval on the real line. If *expression* is less than *bound1*, the result is 0, which represents an underflow bucket. If *expression* is greater than or equal to *bound2*, the result is *num-buckets* + 1, which represents an overflow bucket.
- If *bound1* is greater than *bound2*, each bucket is a left-closed, right-open interval on the real line. If *expression* is greater than *bound1*, the result is 0, which represents an underflow bucket. If *expression* is less than or equal to *bound2*, the result is *num-buckets* + 1, which represents an overflow bucket.
- When *num-buckets* is the maximum value for the data type, an error is returned if the result is *num-buckets* + 1 (SQLSTATE 22003).
- Several arithmetic operations are used to compute the result. If any of these arithmetic operations result in an overflow, an error is returned (SQLSTATE 22003).

Example

Using the EMPLOYEE table, assign a bucket to each employee's salary using a range of 35000 to 100000 divided into 13 buckets.

```
   SELECT EMPNO, SALARY, WIDTH_BUCKET(SALARY, 35000, 100000, 13)
      FROM EMPLOYEE ORDER BY EMPNO
```

15 buckets are assigned with the following ranges:

- Bucket 0: salary < 35000
- Bucket 1: 35000 <= salary < 40000
- Bucket 2: 40000 <= salary < 45000
- Bucket 3: 45000 <= salary < 50000
- Bucket 4: 50000 <= salary < 55000
- Bucket 5: 55000 <= salary < 60000
- Bucket 6: 60000 <= salary < 65000
- Bucket 7: 65000 <= salary < 70000
- Bucket 8: 70000 <= salary < 75000
- Bucket 9: 75000 <= salary < 80000
- Bucket 10: 80000 <= salary < 85000
- Bucket 11: 85000 <= salary < 90000
- Bucket 12: 90000 <= salary < 95000
- Bucket 13: 95000 <= salary < 100000
- Bucket 14: salary >= 100000

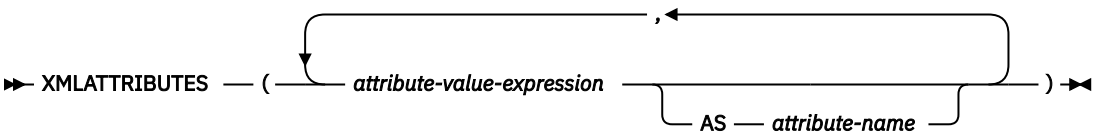
The query has the following output:

```
EMPNO  SALARY      3
------ ----------- -----------
000010   152750.00          14
000020    94250.00          12
000030    98250.00          13
000050    80175.00          10
000060    72250.00           8
000070    96170.00          13
000090    89750.00          11
000100    86150.00          11
000110    66500.00           7
000120    49250.00           3
000130    73800.00           8
000140    68420.00           7
000150    55280.00           5
000160    62250.00           6
000170    44680.00           2
000180    51340.00           4
000190    50450.00           4
000200    57740.00           5
000210    68270.00           7
000220    49840.00           3
000230    42180.00           2
000240    48760.00           3
000250    49180.00           3
000260    47250.00           3
000270    37380.00           1
000280    36250.00           1
000290    35340.00           1
000300    37750.00           1
000310    35900.00           1
000320    39950.00           1
000330    45370.00           3
000340    43840.00           2
200010    46500.00           3
200120    39250.00           1
200140    68420.00           7
200170    64680.00           6
```

```
200220    69840.00          7
200240    37760.00          1
200280    46250.00          3
200310    35900.00          1
200330    35370.00          1
200340    31840.00          0

  42 record(s) selected.
```

XMLATTRIBUTES

The XMLATTRIBUTES function constructs XML attributes from the arguments.



The schema is SYSIBM. The function name cannot be specified as a qualified name.

This function can only be used as an argument of the XMLELEMENT function. The result is an XML sequence containing an XQuery attribute node for each non-null input value.

attribute-value-expression
An expression whose result is the attribute value. The data type of *attribute-value-expression* cannot be an XML type, a BINARY type, a VARBINARY type, or a structured type (SQLSTATE 42884). The expression can be any SQL expression. If the expression is not a simple column reference, an attribute name must be specified.

attribute-name
Specifies an attribute name. The name is an SQL identifier that must be in the form of an XML qualified name, or QName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. The attribute name cannot be `xmlns` or prefixed with `xmlns:`. A namespace is declared using the function XMLNAMESPACES. Duplicate attribute names, whether implicit or explicit, are not allowed (SQLSTATE 42713).

If *attribute-name* is not specified, *attribute-value-expression* must be a column name (SQLSTATE 42703). The attribute name is created from the column name using the fully escaped mapping from a column name to an XML attribute name.

The data type of the result is XML. If the result of *attribute-value-expression* can be null, the result can be null; if the result of every *attribute-value-expression* is null, the result is the null value.

Examples

Note: XMLATTRIBUTES does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- *Example 1*: Produce an element with attributes.

```
SELECT E.EMPNO, XMLELEMENT(
  NAME "Emp",
  XMLATTRIBUTES(
    E.EMPNO, E.FIRSTNME ||' '|| E.LASTNAME AS "name"
  )
)
AS "Result"
FROM EMPLOYEE E WHERE E.EDLEVEL = 12
```

This query produces the following result:

```
EMPNO  Result
000290 <Emp EMPNO="000290" name="JOHN PARKER"></Emp>
000310 <Emp EMPNO="000310" name="MAUDE SETRIGHT"></Emp>
200310 <Emp EMPNO="200310" name="MICHELLE SPRINGER"></Emp>
```

- *Example 2*: Produce an element with a namespace declaration that is not used in any QName. The prefix is used in an attribute value.

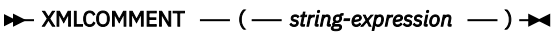
```
VALUES XMLELEMENT(
  NAME "size",
  XMLNAMESPACES(
    'http://www.w3.org/2001/XMLSchema-instance' AS "xsi",
    'http://www.w3.org/2001/XMLSchema' AS "xsd"
  ),
  XMLATTRIBUTES(
    'xsd:string' AS "xsi:type"
  ), '1'
)
```

This query produces the following result:

```
<size xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xsi:type="xsd:string">1</size>
```

XMLCOMMENT

The XMLCOMMENT function returns an XML value with a single XQuery comment node with the input argument as the content.



The schema is SYSIBM. The function name cannot be specified as a qualified name.

string-expression
An expression whose value has a character string type: CHAR, VARCHAR or CLOB. The result of the *string-expression* is parsed to check for conformance to the requirements for an XML comment, as specified in the XML 1.0 rule. The result of the *string-expression* must conform to the following regular expression:

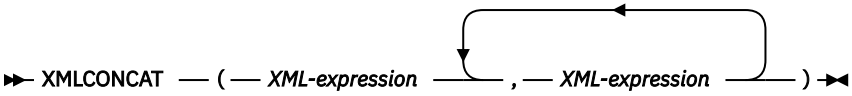
```
((Char - '-') | ('-' (Char - '-')))*
```

where `Char` is defined as any Unicode character excluding surrogate blocks X'FFFE' and X'FFFF'. Basically, the XML comment cannot contain two adjacent hyphens, and cannot end with a hyphen (SQLSTATE 2200S).

The data type of the result is XML. If the result of *string-expression* can be null, the result can be null; if the input value is null, the result is the null value.

XMLCONCAT

The XMLCONCAT function returns a sequence containing the concatenation of a variable number of XML input arguments.



The schema is SYSIBM. The function name cannot be specified as a qualified name.

XML-expression
Specifies an expression of data type XML.

The data type of the result is XML. The result is an XML sequence containing the concatenation of the non-null input XML values. Null values in the input are ignored. If the result of any *XML-expression* can be null, the result can be null; if the result of every input value is null, the result is the null value.

Example

Note: XMLCONCAT does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

Construct a department element for departments A00 and B01, containing a list of employees sorted by first name. Include an introductory comment immediately preceding the department element.

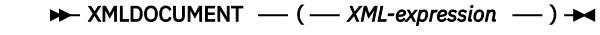
```
SELECT XMLCONCAT(
  XMLCOMMENT(
    'Confirm these employees are on track for their product schedule'
  ),
  XMLELEMENT(
    NAME "Department",
    XMLATTRIBUTES(
      E.WORKDEPT AS "name"
    ),
    XMLAGG(
      XMLELEMENT(
        NAME "emp", E.FIRSTNME
      )
    ORDER BY E.FIRSTNME
    )
  )
)
FROM EMPLOYEE E
WHERE E.WORKDEPT IN ('A00', 'B01')
GROUP BY E.WORKDEPT
```

This query produces the following result:

```
<!--Confirm these employees are on track for their product schedule-->
<Department name="A00">
<emp>CHRISTINE</emp>
<emp>DIAN</emp>
<emp>GREG</emp>
<emp>SEAN</emp>
<emp>VINCENZO</emp>
</Department>
<!--Confirm these employees are on track for their product schedule-->
<Department name="B01">
<emp>MICHAEL</emp>
</Department>
```

XMLDOCUMENT

The XMLDOCUMENT function returns an XML value with a single XQuery document node with zero or more children nodes.



The schema is SYSIBM. The function name cannot be specified as a qualified name.

XML-expression
An expression that returns an XML value. A sequence item in the XML value must not be an attribute node (SQLSTATE 10507).

The data type of the result is XML. If the result of *XML-expression* can be null, the result can be null; if the input value is null, the result is the null value.

The children of the resulting document node are constructed as described in the following steps. The input expression is a sequence of nodes or atomic values, which is referred to in these steps as the content sequence.

1. If the content sequence contains a document node, the document node is replaced in the content sequence by the children of the document node.
2. Each adjacent sequence of one or more atomic values in the content sequence are replaced with a text node containing the result of casting each atomic value to a string with a single blank character inserted between adjacent values.

3. For each node in the content sequence, a new deep copy of the node is constructed. A deep copy of a node is a copy of the whole subtree rooted at that node, including the node itself and its descendants. Each copied node has a new node identity.
4. The nodes in the content sequence become the children of the new document node.

The XMLDOCUMENT function effectively executes the XQuery computed document constructor. The result of

```
XMLQUERY('document {$E}' PASSING BY REF XML-expression AS "E")
```

is equivalent to

```
XMLDOCUMENT( XML-expression )
```

with the exception of the case where *XML-expression* is null and XMLQUERY returns the empty sequence compared to XMLDOCUMENT which returns the null value.

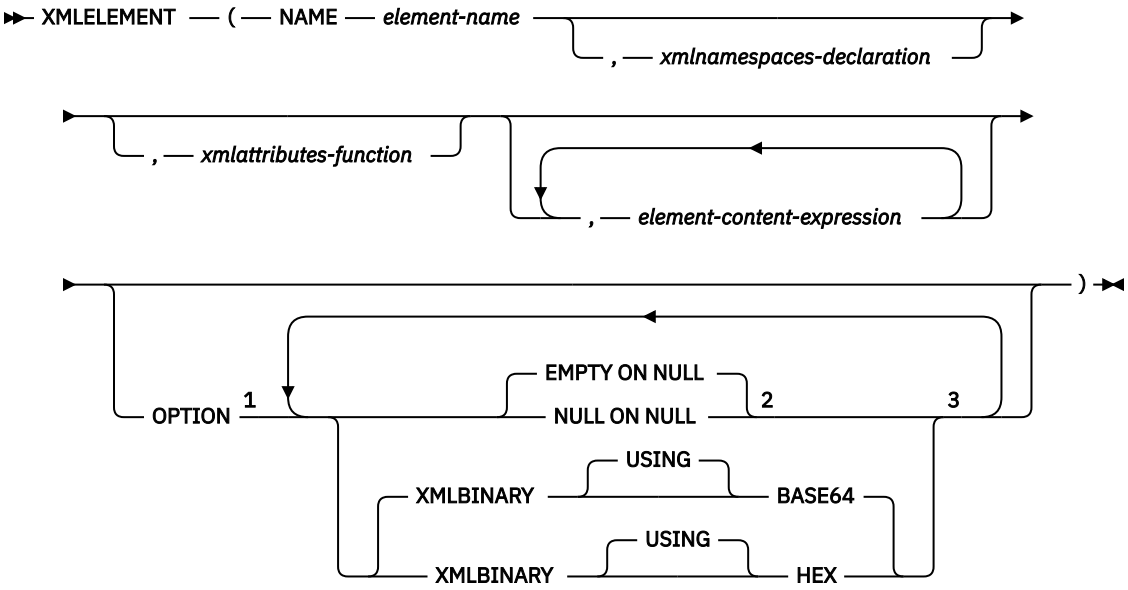
Example

Insert a constructed document into an XML column.

```
INSERT INTO T1 VALUES(
  123, (
    SELECT XMLDOCUMENT(
      XMLELEMENT(
        NAME "Emp", E.FIRSTNME || ' ' || E.LASTNAME, XMLCOMMENT(
          'This is just a simple example'
        )
      )
    )
    FROM EMPLOYEE E
    WHERE E.EMPNO = '000120'
  )
)
```

XMLELEMENT

The XMLELEMENT function returns an XML value that is an XQuery element node.



Notes:

[1] The OPTION clause can only be specified if at least one *xmlattributes-function* or *element-content-expression* is specified.

[2] NULL ON NULL or EMPTY ON NULL can only be specified if at least one *element-content-expression* is specified.

[3] The same clause must not be specified more than once.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

NAME *element-name*

Specifies the name of an XML element. The name is an SQL identifier that must be in the form of an XML qualified name, or QName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope (SQLSTATE 42635).

xmlnamespaces-declaration

Specifies the XML namespace declarations that are the result of the XMLNAMESPACES declaration. The namespaces that are declared are in the scope of the XMLELEMENT function. The namespaces apply to any nested XML functions within the XMLELEMENT function, regardless of whether or not they appear inside another subselect.

If *xmlnamespaces-declaration* is not specified, namespace declarations are not associated with the constructed element.

xmlattributes-function

Specifies the XML attributes for the element. The attributes are the result of the XMLATTRIBUTES function.

element-content-expression

The content of the generated XML element node is specified by an expression or a list of expressions. The data type of *element-content-expression* cannot be a BINARY type, a VARBINARY type, or a structured type (SQLSTATE 42884). The expression can be any SQL expression.

If *element-content-expression* is not specified, an empty string is used as the content for the element and OPTION NULL ON NULL or EMPTY ON NULL must not be specified.

OPTION

Specifies additional options for constructing the XML element. If no OPTION clause is specified, the default is EMPTY ON NULL XMLBINARY USING BASE64. This clause has no impact on nested XMLELEMENT invocations specified in *element-content-expression*.

EMPTY ON NULL or NULL ON NULL

Specifies whether a null value or an empty element is to be returned if the values of each *element-content-expression* is a null value. This option only affects null handling of element contents, not attribute values. The default is EMPTY ON NULL.

EMPTY ON NULL

If the value of each *element-content-expression* is null, an empty element is returned.

NULL ON NULL

If the value of each *element-content-expression* is null, a null value is returned.

XMLBINARY USING BASE64 or XMLBINARY USING HEX

Specifies the assumed encoding of binary input data, character string data with the FOR BIT DATA attribute, or a distinct type that is based on one of these types. The encoding applies to element content or attribute values. The default is XMLBINARY USING BASE64.

XMLBINARY USING BASE64

Specifies that the assumed encoding is base64 characters, as defined for XML schema type xs:base64Binary encoding. The base64 encoding uses a 65-character subset of US-ASCII (10 digits, 26 lowercase characters, 26 uppercase characters, '+', and '/') to represent every six bits of the binary or bit data with one printable character in the subset. These characters are selected so that they are universally representable. Using this method, the size of the encoded data is 33 percent larger than the original binary or bit data.

XMLBINARY USING HEX

Specifies that the assumed encoding is hexadecimal characters, as defined for XML schema type xs:hexBinary encoding. The hexadecimal encoding represents each byte (8 bits) with two

hexadecimal characters. Using this method, the encoded data is twice the size of the original binary or bit data.

This function takes an element name, an optional collection of namespace declarations, an optional collection of attributes, and zero or more arguments that make up the content of the XML element. The result is an XML sequence containing an XML element node or the null value.

The data type of the result is XML. If any of the *element-content-expression* arguments can be null, the result can be null; if all the *element-content-expression* argument values are null and the NULL ON NULL option is in effect, the result is the null value.

Notes

- When constructing elements that will be copied as content of another element that defines default namespaces, default namespaces should be explicitly undeclared in the copied element to avoid possible errors that could result from inheriting the default namespace from the new parent element. Predefined namespace prefixes ('xs', 'xsi', 'xml', and 'sqlxml') must also be declared explicitly when they are used.
- **Constructing an element node**: The resulting element node is constructed as follows:
  1. The *xmlnamespaces-declaration* adds a set of in-scope namespaces for the constructed element. Each in-scope namespace associates a namespace prefix (or the default namespace) with a namespace URI. The in-scope namespaces define the set of namespace prefixes that are available for interpreting QNames within the scope of the element.
  2. If the xmlattributes-function is specified, it is evaluated and the result is a sequence of attribute nodes.
  3. Each *element-content-expression* is evaluated and the result is converted into a sequence of nodes as follows:
     - If the result type is not XML, it is converted to an XML text node whose content is the result of *element-content-expression* mapped to XML according to the rules of mapping SQL data values to XML data values (see the table that describes supported casts from non-XML values to XML values in "Casting between data types").
     - If the result type is XML, then in general the result is a sequence of items. Some of the items in that sequence might be document nodes. Each document node in the sequence is replaced by the sequence of its top-level children. Then for each node in the resulting sequence, a new deep copy of the node is constructed, including its children and attributes. Each copied node has a new node identity. Copied element and attribute nodes preserve their type annotation. For each adjacent sequence of one or more atomic values returned in the sequence, a new text node is constructed, containing the result of casting each atomic value to a string, with a single blank character inserted between adjacent values. Adjacent text nodes in the content sequence are merged into a single text node by concatenating their contents, with no intervening blanks. After concatenation, any text node whose content is a zero-length string is deleted from the content sequence.
  4. The result sequence of XML attributes and the resulting sequences of all *element-content-expression* specifications are concatenated into one sequence which is called the content sequence. Any sequence of adjacent text nodes in the content sequence is merged into a single text node. If all the *element-content-expression* arguments are empty strings, or an *element-content-expression* argument is not specified, an empty element is returned.
  5. The content sequence must not contain an attribute node following a node that is not an attribute node (SQLSTATE 10507). Attribute nodes occurring in the content sequence become attributes of the new element node. Two or more of these attribute nodes must not have the same name (SQLSTATE 10503). A namespace declaration is created corresponding to any namespace used in the names of the attribute nodes if the namespace URI is not in the in-scope namespaces of the constructed element.
  6. Element, text, comment, and processing instruction nodes in the content sequence become the children of the constructed element node.

7. The constructed element node is given a type annotation of `xs:anyType`, and each of its attributes is given a type annotation of `xdt:untypedAtomic`. The node name of the constructed element node is element-name specified after the NAME keyword.

- **Rules for using namespaces within XMLELEMENT**: Consider the following rules about scoping of namespaces:
  - The namespaces declared in the XMLNAMESPACES declaration are the in-scope namespaces of the element node constructed by the XMLELEMENT function. If the element node is serialized, then each of its in-scope namespaces will be serialized as a namespace attribute unless it is an in-scope namespace of the parent of the element node and the parent element is serialized too.
  - If an XMLQUERY or XMLEXISTS is in an *element-content-expression*, then the namespaces becomes the statically known namespaces of the XQuery expression of the XMLQUERY or XMLEXISTS. Statically known namespaces are used to resolve the QNames in the XQuery expression. If the XQuery prolog declares a namespace with the same prefix, within the scope of the XQuery expression, the namespace declared in the prolog will override the namespaces declared in the XMLNAMESPACES declaration.
  - If an attribute of the constructed element comes from an *element-content-expression*, its namespace might not already be declared as an in-scope namespace of the constructed element, in this case, a new namespace is created for it. If this would result in a conflict, which means that the prefix of the attribute name is already bound to a different URI by a in-scope namespace, a prefix is generated that does not cause such a conflict and the prefix used in the attribute name is changed to the new prefix, and a namespace is created for this new prefix. The generated new prefix follows the following pattern: "db2ns-xx", where "x" is a character chosen from the set [A-Z,a-z,0-9]. For example:

```
VALUES XMLELEMENT(
  NAME "c", XMLQUERY(
    'declare namespace ipo="www.ipo.com"; $m/ipo:a/@ipo:b'
      PASSING XMLPARSE(
        DOCUMENT '<tst:a xmlns:tst="www.ipo.com" tst:b="2"/>'
      ) AS "m"
  )
)
```

returns:

```
<c xmlns:tst="www.ipo.com" tst:b="2"/>
```

A second example:

```
VALUES XMLELEMENT(
  NAME "tst:c", XMLNAMESPACES(
    'www.tst.com' AS "tst"
  ),
  XMLQUERY(
    'declare namespace ipo="www.ipo.com"; $m/ipo:a/@ipo:b'
      PASSING XMLPARSE(
        DOCUMENT '<tst:a xmlns:tst="www.ipo.com" tst:b="2"/>'
      ) AS "m"
  )
)
```

returns:

```
<tst:c xmlns:tst="www.tst.com" xmlns:db2ns-a1="www.ipo.com"
  db2ns-a1:b="2"/>
```

Examples

Note: XMLELEMENT does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- *Example 1*: Construct an element with the NULL ON NULL option.

```
SELECT E.FIRSTNME, E.LASTNAME, XMLELEMENT(
  NAME "Emp", XMLELEMENT(
```

```
      NAME "firstname", E.FIRSTNME
    ),
    XMLELEMENT(
      NAME "lastname", E.LASTNAME
    )
    OPTION NULL ON NULL
  )
  AS "Result"
  FROM EMPLOYEE E
  WHERE E.EDLEVEL = 12
```

This query produces the following result:

```
FIRSTNME     LASTNAME        Emp
JOHN         PARKER          <Emp><firstname>JOHN</firstname>
                               <lastname>PARKER</lastname></Emp>
MAUDE        SETRIGHT        <Emp><firstname>MAUDE</firstname>
                               <lastname>SETRIGHT</lastname></Emp>
MICHELLE     SPRINGER        <Emp><firstname>MICHELLE</firstname>
                               <lastname>SPRINGER</lastname></Emp>
```

- *Example 2*: Produce an element with a list of elements nested as child elements.

```
  SELECT XMLELEMENT(
    NAME "Department", XMLATTRIBUTES(
      E.WORKDEPT AS "name"
    ),
    XMLAGG(
      XMLELEMENT(
        NAME "emp", E.FIRSTNME
      )
      ORDER BY E.FIRSTNME
    )
  )
  AS "dept_list"
  FROM EMPLOYEE E
  WHERE E.WORKDEPT IN ('A00', 'B01')
  GROUP BY WORKDEPT
```

This query produces the following result:

```
dept_list
<Department name="A00">
<emp>CHRISTINE</emp>
<emp>SEAN</emp>
<emp>VINCENZO</emp>
</Department>
<Department name="B01">
<emp>MICHAEL</emp>
</Department>
```

- *Example 3*: Creating nested XML elements specifying a default XML element namespace and using a subselect.

```
SELECT XMLELEMENT(
         NAME "root",
         XMLNAMESPACES(DEFAULT 'http://mytest.uri'),
         XMLATTRIBUTES(cid),
            (SELECT
               XMLAGG(
                 XMLELEMENT(
                   NAME "poid", poid
                 )
               )
             FROM purchaseorder
             WHERE purchaseorder.custid = customer.cid
            )
       )
FROM customer
WHERE cid = '1002'
```

The statement returns the following XML document with the default element namespace declared in the root element:

```
<root xmlns="http://mytest.uri" CID="1002">
  <poid>5000</poid>
```

```
  <poid>5003</poid>
  <poid>5006</poid>
</root>
```

- *Example 4*: Using a common table expression with XML namespaces.

  When an XML element is constructed with a common table expression and the element is used in elsewhere in the same SQL statement, any namespace declarations should be specified as part of the element construction. The following statement specifies the default XML namespace in both the common table expression that uses the PURCHASEORDER table to create the poid elements and the SELECT statement that uses the CUSTOMER table to create the root element.

```
WITH tempid(id, elem) AS
  (SELECT custid, XMLELEMENT(NAME "poid",
     XMLNAMESPACES(DEFAULT 'http://mytest.uri'),
       poid)
   FROM purchaseorder )
SELECT XMLELEMENT(NAME "root",
         XMLNAMESPACES(DEFAULT 'http://mytest.uri'),
            XMLATTRIBUTES(cid),
            (SELECT XMLAGG(elem)
             FROM tempid
             WHERE tempid.id = customer.cid )
        )
FROM customer
WHERE cid = '1002'
```

  The statement returns the following XML document with a default element namespace declared in the root element.

```
<root xmlns="http://mytest.uri" CID="1002">
  <poid>5000</poid>
  <poid>5003</poid>
  <poid>5006</poid>
</root>
```

  In the following statement, the default element namespace is declared only in the SELECT statement that uses the CUSTOMER table to create the root element:

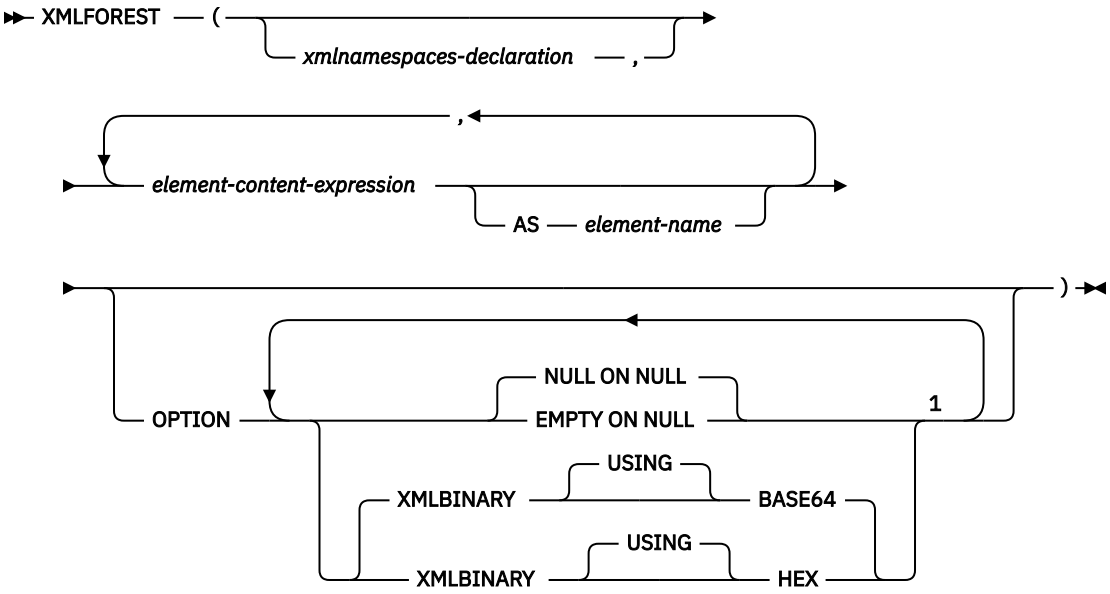
```
WITH tempid(id, elem) AS
    (SELECT custid, XMLELEMENT(NAME "poid", poid)
     FROM purchaseorder )
SELECT XMLELEMENT(NAME "root",
         XMLNAMESPACES(DEFAULT 'http://mytest.uri'),
            XMLATTRIBUTES(cid),
            (SELECT XMLAGG(elem)
             FROM tempid
             WHERE tempid.id = customer.cid )
       )
FROM customer
WHERE cid = '1002'
```

  The statement returns the following XML document with the default element namespace declared in the root element. Because the poid elements are created in the common table expression without a default element namespace declaration, the default element namespace for the poid elements is not defined. In the XML document, the default element namespace for the poid elements is set to an empty string "" because the default element namespace for the poid elements is not defined, and the poid elements do not belong to the default element namespace of the root element `xmlns="http://mytest.uri"`.

```
<root xmlns="http://mytest.uri" CID="1002">
  <poid xmlns="">5000</poid>
  <poid xmlns="">5003</poid>
  <poid xmlns="">5006</poid>
</root>
```

XMLFOREST

The XMLFOREST function returns an XML value that is a sequence of XQuery element nodes.



Notes:

[1] The same clause must not be specified more than once.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

xmlnamespaces-declaration
Specifies the XML namespace declarations that are the result of the XMLNAMESPACES declaration. The namespaces that are declared are in the scope of the XMLFOREST function. The namespaces apply to any nested XML functions within the XMLFOREST function, regardless of whether or not they appear inside another subselect.

If *xmlnamespaces-declaration* is not specified, namespace declarations are not associated with the constructed elements.

element-content-expression
The content of the generated XML element node is specified by an expression. The data type of *element-content-expression* cannot be a BINARY type, a VARBINARY type, or a structured type (SQLSTATE 42884). The expression can be any SQL expression. If the expression is not a simple column reference, an element name must be specified.

AS *element-name*
Specifies the XML element name as an SQL identifier. The element name must be of the form of an XML qualified name, or QName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope (SQLSTATE 42635). If *element-name* is not specified, *element-content-expression* must be a column name (SQLSTATE 42703). The element name is created from the column name using the fully escaped mapping from a column name to an QName.

OPTION
Specifies additional options for constructing the XML element. If no OPTION clause is specified, the default is NULL ON NULL XMLBINARY USING BASE64. This clause has no impact on nested XMLELEMENT invocations specified in *element-content-expression*.

EMPTY ON NULL or NULL ON NULL
Specifies whether a null value or an empty element is to be returned if the values of each *element-content-expression* is a null value. This option only affects null handling of element contents, not attribute values. The default is NULL ON NULL.

EMPTY ON NULL
If the value of each *element-content-expression* is null, an empty element is returned.

NULL ON NULL
If the value of each *element-content-expression* is null, a null value is returned.

XMLBINARY USING BASE64 or XMLBINARY USING HEX
Specifies the assumed encoding of binary input data, character string data with the FOR BIT DATA attribute, or a distinct type that is based on one of these types. The encoding applies to element content or attribute values. The default is XMLBINARY USING BASE64.

XMLBINARY USING BASE64
Specifies that the assumed encoding is base64 characters, as defined for XML schema type xs:base64Binary encoding. The base64 encoding uses a 65-character subset of US-ASCII (10 digits, 26 lowercase characters, 26 uppercase characters, '+', and '/') to represent every six bits of the binary or bit data with one printable character in the subset. These characters are selected so that they are universally representable. Using this method, the size of the encoded data is 33 percent larger than the original binary or bit data.

XMLBINARY USING HEX
Specifies that the assumed encoding is hexadecimal characters, as defined for XML schema type xs:hexBinary encoding. The hexadecimal encoding represents each byte (8 bits) with two hexadecimal characters. Using this method, the encoded data is twice the size of the original binary or bit data.

This function takes an optional set of namespace declarations and one or more arguments that make up the name and element content for one or more element nodes. The result is an XML sequence containing a sequence of XQuery element nodes or the null value.

The data type of the result is XML. If any of the *element-content-expression* arguments can be null, the result can be null; if all the *element-content-expression* argument values are null and the NULL ON NULL option is in effect, the result is the null value.

The XMLFOREST function can be expressed by using XMLCONCAT and XMLELEMENT. For example, the following two expressions are semantically equivalent.

```
XMLFOREST(xmlnamespaces-declaration, arg1 AS name1, arg2 AS name2 ...)
```

```
XMLCONCAT(
  XMLELEMENT(
    NAME name1, xmlnamespaces-declaration, arg1
  ),
  XMLELEMENT(
    NAME name2, xmlnamespaces-declaration, arg2
  )
  ...
)
```

Notes

- When constructing elements that will be copied as content of another element that defines default namespaces, default namespaces should be explicitly undeclared in the copied element to avoid possible errors that could result from inheriting the default namespace from the new parent element. Predefined namespace prefixes ('xs', 'xsi', 'xml', and 'sqlxml') must also be declared explicitly when they are used.

Example

Note: XMLFOREST does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

Construct a forest of elements with a default namespace.

```
   SELECT EMPNO,
     XMLFOREST(
```

```
      XMLNAMESPACES(
        DEFAULT 'http://hr.org', 'http://fed.gov' AS "d"
      ),
      LASTNAME, JOB AS "d:job"
    )
  AS "Result"
  FROM EMPLOYEE
  WHERE EDLEVEL = 12
```

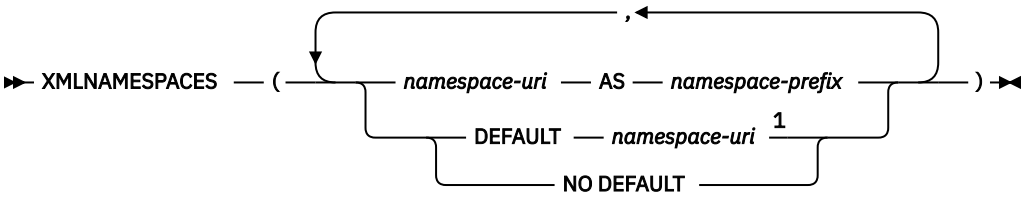
This query produces the following result:

```
EMPNO  Result
000290 <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">PARKER
  </LASTNAME>
<d:job xmlns="http://hr.org" xmlns:d="http://fed.gov">OPERATOR</d:job>

000310 <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">SETRIGHT
  </LASTNAME>
<d:job xmlns="http://hr.org" xmlns:d="http://fed.gov">OPERATOR</d:job>

200310 <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">SPRINGER
  </LASTNAME>
<d:job xmlns="http://hr.org" xmlns:d="http://fed.gov">OPERATOR</d:job>
```

XMLNAMESPACES

The XMLNAMESPACES declaration constructs namespace declarations from the arguments.

xmlnamespaces-declaration



Notes:

1 DEFAULT or NO DEFAULT can only be specified once in arguments of XMLNAMESPACES.

The schema is SYSIBM. The declaration name cannot be specified as a qualified name.

This declaration can only be used as an argument for specific functions such as XMLELEMENT, XMLFOREST and XMLTABLE. The result is one or more XML namespace declarations containing in-scope namespaces for each non-null input value.

namespace-uri
Specifies the namespace universal resource identifier (URI) as an SQL character string constant. This character string constant must not be empty if it is used with a *namespace-prefix* (SQLSTATE 42815).

namespace-prefix
Specifies a namespace prefix. The prefix is an SQL identifier that must be in the form of an XML NCName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. The prefix cannot be `xml` or `xmlns` and the prefix must be unique within the list of namespace declarations (SQLSTATE 42635).

DEFAULT *namespace-uri*
Specifies the default namespace to use within the scope of this namespace declaration. The *namespace-uri* applies for unqualified names in the scope unless overridden in a nested scope by another DEFAULT declaration or a NO DEFAULT declaration.

NO DEFAULT
Specifies that no default namespace is to be used within the scope of this namespace declaration. There is no default namespace in the scope unless overridden in a nested scope by a DEFAULT declaration.

The data type of the result is XML. The result is an XML namespace declaration for each specified namespace. The result cannot be null.

Examples

Note: XMLNAMESPACES does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- *Example 1*: Produce an XML element named adm:employee and an XML attribute adm:department, both associated with a namespace whose prefix is adm.

```
SELECT EMPNO, XMLELEMENT(
  NAME "adm:employee", XMLNAMESPACES(
    'http://www.adm.com' AS "adm"
  ),
  XMLATTRIBUTES(
    WORKDEPT AS "adm:department"
  ),
LASTNAME
)
FROM EMPLOYEE
WHERE JOB = 'ANALYST'
```

This query produces the following result:

```
000130 <adm:employee xmlns:adm="http://www.adm.com" adm:department="C01">
  QUINTANA</adm:employee>
000140 <adm:employee xmlns:adm="http://www.adm.com" adm:department="C01">
  NICHOLLS</adm:employee>
200140 <adm:employee xmlns:adm="http://www.adm.com" adm:department="C01">
  NATZ</adm:employee>
```

- *Example 2*: Produce an XML element named 'employee', which is associated with a default namespace, and a sub-element named 'job', which does not use a default namespace, but whose sub-element named 'department' does use a default namespace.

```
SELECT EMP.EMPNO, XMLELEMENT(
  NAME "employee", XMLNAMESPACES(
    DEFAULT 'http://hr.org'
  ),
  EMP.LASTNAME, XMLELEMENT(
    NAME "job", XMLNAMESPACES(
      NO DEFAULT
    ),
    EMP.JOB, XMLELEMENT(
      NAME "department", XMLNAMESPACES(
        DEFAULT 'http://adm.org'
      ),
      EMP.WORKDEPT
    )
  )
)
FROM EMPLOYEE EMP
WHERE EMP.EDLEVEL = 12
```

This query produces the following result:

```
000290 <employee xmlns="http://hr.org">PARKER<job xmlns="">OPERATOR
  <department xmlns="http://adm.org">E11</department></job></employee>
000310 <employee xmlns="http://hr.org">SETRIGHT<job xmlns="">OPERATOR
  <department xmlns="http://adm.org">E11</department></job></employee>
200310 <employee xmlns="http://hr.org">SPRINGER<job xmlns="">OPERATOR
  <department xmlns="http://adm.org">E11</department></job></employee>
```

XMLPARSE

The XMLPARSE function parses the argument as an XML document and returns an XML value.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

DOCUMENT
Specifies that the character string expression to be parsed must evaluate to a well-formed XML document that conforms to XML 1.0, as modified by the XML Namespaces recommendation (SQLSTATE 2200M).

string-expression
Specifies an expression that returns a character string or BLOB value. If a parameter marker is used, it must explicitly be cast to one of the supported data types.

STRIP WHITESPACE or PRESERVE WHITESPACE
Specifies whether or not whitespace in the input argument is to be preserved. If neither is specified, STRIP WHITESPACE is the default.

STRIP WHITESPACE
Specifies that text nodes containing only whitespace characters up to 1000 bytes in length will be stripped, unless the nearest containing element has the attribute xml:space='preserve'. If any text node begins with more that 1000 bytes of whitespace, an error is returned (SQLSTATE 54059).

The whitespace characters in the CDATA section are also affected by this option. DTDs may have DOCTYPE declarations for elements, but the content models of elements are not used to determine if whitespace is stripped or not.

PRESERVE WHITESPACE
Specifies that all whitespace is to be preserved, even when the nearest containing element has the attribute xml:space='default'.

The data type of the result is XML. If the result of *string-expression* can be null, the result can be null; if the result of *string-expression* is null, the result is the null value.

Notes

- **Encoding of the input string**: The input string may contain an XML declaration that identifies the encoding of the characters in the XML document. If the string is passed to the XMLPARSE function as a character string, it will be converted to the code page at the database server. This code page may be different from the originating code page and the encoding identified in the XML declaration.

  Therefore, applications should avoid direct use of XMLPARSE with character string input and should send strings containing XML documents directly using host variables to maintain the match between the external code page and the encoding in the XML declaration. If XMLPARSE must be used in this situation, a BLOB type should be specified as the argument to avoid code page conversion.
- **Handling of DTDs**: External document type definitions (DTDs) and entities must be registered in a database. Both internal and external DTDs are checked for valid syntax. During the parsing process, the following actions are also performed:
  - Default values that are defined by the internal and external DTDs are applied.
  - Entity references and parameter entities are replaced by their expanded forms.
  - If an internal DTD and an external DTD define the same element, an error is returned (SQLSTATE 2200M).
  - If an internal DTD and an external DTD define the same entity or attribute, the internal definition is chosen.

  After parsing, internal DTDs and entities, as well as references to external DTDs and entities, are not preserved in the stored representation of the value.
- **Character conversion in non-UTF-8 databases**: Code page conversion occurs when an XML document is parsed into a non-Unicode database server, if the document is passed in from a host variable or parameter marker of a character data type, or from a character string literal. Parsing an XML document using a host variable or parameter marker of type XML, BLOB or FOR BIT DATA (CHAR FOR BIT DATA or VARCHAR FOR BIT DATA) prevents code page conversion. When a character data type is used, care must be taken to ensure that all characters in the XML document have a matching code point in the target database code page, otherwise substitution characters may be introduced. The configuration parameter **enable_xmlchar** can be used to help ensure the integrity of XML data stored

in a non-Unicode database. Setting this parameter to "NO" blocks the insertion of XML documents from character data types. The BLOB and FOR BIT DATA data types are still allowed, as documents passed into a database using these data types avoid code page conversion.

Example

Using the PRESERVE WHITESPACE option preserves the white space characters in the XML document inserted into the table, including the white space characters in the description element.

```
INSERT INTO PRODUCT VALUES ('100-103-99','Tool bag',14.95,NULL,NULL,NULL,
XMLPARSE( DOCUMENT
  '<product xmlns="http://posample.org" pid="100-103-99">
    <description>
     <name>Tool bag</name>
     <details>
      Super Deluxe tool bag:
      - 26 inches long, 12 inches wide
      - Curved padded handle
      - Locking latch
      - Reinforced exterior pockets
     </details>
     <price>14.95</price>
     <weight>3 kg</weight>
    </description>
  </product>' PRESERVE WHITESPACE ));
```

Running the following select statement

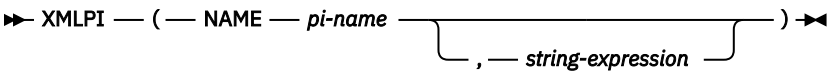
```
SELECT XMLQUERY ('$d/*:product/*:description/*:details' PASSING DESCRIPTION as "d" )
FROM PRODUCT WHERE PID = '100-103-99' ;
```

returns the details element with the white space characters:

```
<details xmlns="http://posample.org">
  Super Deluxe tool bag:
  - 26 inches long, 12 inches wide
  - Curved padded handle
  - Locking latch
  - Reinforced exterior pockets
</details>
```

XMLPI

The XMLPI function returns an XML value with a single XQuery processing instruction node.



The schema is SYSIBM. The function name cannot be specified as a qualified name.

NAME *pi-name*
Specifies the name of a processing instruction. The name is an SQL identifier that must be in the form of an XML NCName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. The name cannot be the word 'xml' in any case combination (SQLSTATE 42634).

string-expression
An expression that returns a value that is a character string. The resulting string is converted to UTF-8 and must conform to the content of an XML processing instruction as specified in XML 1.0 rules (SQLSTATE 2200T):

- The string must not contain the substring '?>' since this substring terminates a processing instruction
- Each character of the string can be any Unicode character excluding the surrogate blocks, X'FFFE' and X'FFFF'.

The resulting string becomes the content of the constructed processing instruction node.

The data type of the result is XML. If the result of *string-expression* can be null, the result can be null; if the result of *string-expression* is null, the result is the null value. If *string-expression* is an empty string or is not specified, an empty processing instruction node is returned.

Examples

- *Example 1*: Generate an XML processing instruction node.

```
SELECT XMLPI(
  NAME "Instruction", 'Push the red button'
)
FROM SYSIBM.SYSDUMMY1
```

This query produces the following result:

```
<?Instruction Push the red button?>
```

- *Example 2*: Generate an empty XML processing instruction node.

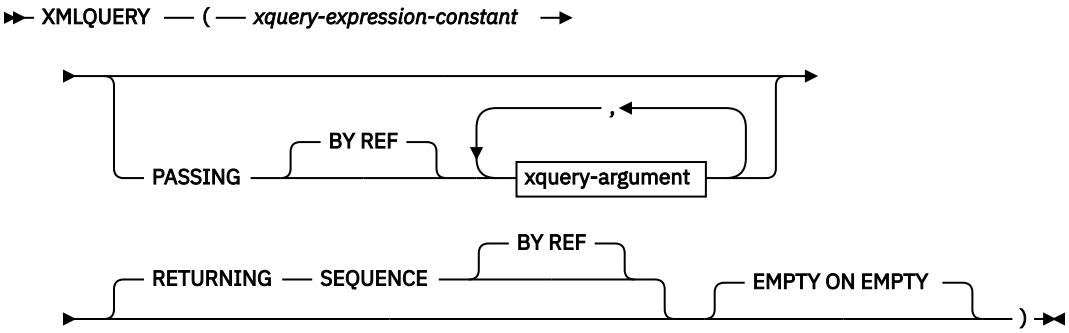
```
SELECT XMLPI(
  NAME "Warning"
)
FROM SYSIBM.SYSDUMMY1
```

This query produces the following result:

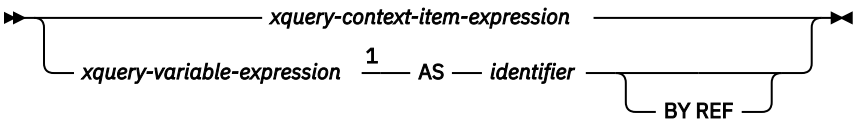
```
<?Warning ?>
```

XMLQUERY

The XMLQUERY function returns an XML value from the evaluation of an XQuery expression possibly using specified input arguments as XQuery variables.



xquery-argument



Notes:

[1] The data type of the expression cannot be DECFLOAT.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

xquery-expression-constant
Specifies an SQL character string constant that is interpreted as an XQuery expression using supported XQuery language syntax. The constant string is converted to UTF-8 before being parsed as an XQuery statement. The XQuery expression executes using an optional set of input XML values, and returns an output sequence that is also returned as the value of the XMLQUERY expression. The value for *xquery-expression-constant* must not be an empty string or a string of blank characters (SQLSTATE 10505).

PASSING

Specifies input values and the manner in which these values are passed to the XQuery expression specified by *xquery-expression-constant*. By default, every unique column name that is in the scope where the function is invoked is implicitly passed to the XQuery expression using the name of the column as the variable name. If an *identifier* in a specified xquery-argument matches an in-scope column name, then the explicit xquery-argument is passed to the XQuery expression overriding that implicit column.

BY REF

Specifies that the default passing mechanism is by reference for any *xquery-variable-expression* of data type XML and for the returned value. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any, directly from the specified input expressions, preserving all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons involving some nodes contained between the two input arguments might refer to nodes within the same XML node tree.

This clause has no impact on how non-XML values are passed. The non-XML values create a new copy of the value during the cast to XML.

xquery-argument

Specifies an argument that is to be passed to the XQuery expression specified by *xquery-expression-constant*. An argument specifies a value and the manner in which that value is to be passed. *xquery-argument* specifies both a value and the manner in which that value is to be passed. The method through which an argument, in the PASSING clause, is used in the XQuery expression depends on whether the argument is specified as *xquery-context-item-expression* or *xquery-variable-expression*. *xquery-argument* includes an SQL expression that is evaluated before it passes the result to the XQuery expression.

- If the resulting value is of type XML, it becomes an *input-xml-value*. A null XML value is converted to an XML empty sequence.
- If the resulting value is not of type XML, it must be castable to the XML data type. A null value is converted to an XML empty sequence. The converted value becomes an *input-xml-value*.

When the *xquery-expression-constant* is evaluated, an XQuery variable is presented with a value equal to *input-xml-value* and a name specified by the AS clause.

xquery-context-item-expression

xquery-context-item-expression specifies the initial context item in the XQuery expression that is specified by *xquery-expression-constant*. The value of the initial context item is the result of *xquery-context-item-expression* to XML. *xquery-context-item-expression* must not be specified more than one time. *xquery-context-item-expression* must not be a sequence of more than one item.

If *input-xml-value* is an empty XML string, the XQuery expression is evaluated with the initial context item set to an empty XML string. If the value of *input-xml-value* is null, the function returns a null value. If the *xquery-context-item-expression* is not specified or is an empty sequence, the initial context item in the XQuery expression is undefined and the XQuery expression must not reference the initial context item.

An XQuery variable is not created for the context item expression.

xquery-variable-expression

Specifies an SQL expression whose value is available to the XQuery expression specified by *xquery-expression-constant* during execution. The expression cannot contain a sequence reference (SQLSTATE 428F9) or an OLAP function (SQLSTATE 42903). The data type of the expression cannot be DECFLOAT.

AS *identifier*

Specifies that the value generated by *xquery-variable-expression* will be passed to *xquery-expression-constant* as an XQuery variable. The variable name will be *identifier*. The leading dollar sign ($) that precedes variable names in the XQuery language is not included in *identifier*. The identifier must be a valid XQuery variable name and is restricted to an XML

NCName (SQLSTATE 42634). The identifier must not be greater than 128 bytes in length. Two arguments within the same PASSING clause cannot use the same identifier (SQLSTATE 42711).

BY REF
Indicates that an XML input value is to be passed by reference. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any, directly from the specified input expressions, preserving all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons involving some nodes contained between the two input arguments might refer to nodes within the same XML node tree. If BY REF is not specified following an *xquery-variable-expression*, XML arguments are passed by way of the default passing mechanism that is provided through the syntax that follows the PASSING keyword. This option cannot be specified for non-XML values. When a non-XML value is passed, the value is converted to XML; this process creates a copy.

RETURNING SEQUENCE
Indicates that the XMLQUERY expression returns a sequence.

BY REF
Indicates that the result of the XQuery expression is returned by reference. If this value contains nodes, any expression using the return value of the XQuery expression will receive node references directly, preserving all node properties, including the original node identities and document order. Referenced nodes will remain connected within their node trees. If the BY REF clause is not specified and the PASSING is specified, the default passing mechanism is used. If BY REF is not specified and PASSING is not specified, the default returning mechanism is BY REF.

EMPTY ON EMPTY
Specifies that an empty sequence result from processing the XQuery expression is returned as an empty sequence.

The data type of the result is XML; it cannot be null.

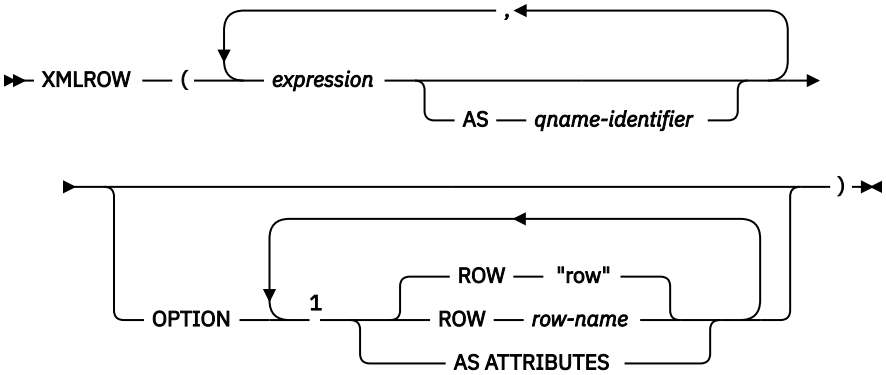
If the evaluation of the XQuery expression results in an error, then the XMLQUERY function returns the XQuery error (SQLSTATE class '10').

Notes

- **XMLQUERY usage restrictions**: The XMLQUERY function cannot be:
  - Part of the ON clause that is associated with a JOIN operator or a MERGE statement (SQLSTATE 42972)
  - Part of the GENERATE KEY USING or RANGE THROUGH clause in the CREATE INDEX EXTENSION statement (SQLSTATE 428E3)
  - Part of the FILTER USING clause in the CREATE FUNCTION (External Scalar) statement, or the FILTER USING clause in the CREATE INDEX EXTENSION statement (SQLSTATE 428E4)
  - Part of a check constraint or a column generation expression (SQLSTATE 42621)
  - Part of a group-by-clause (SQLSTATE 42822)
  - Part of an argument for an aggregate function (SQLSTATE 42607)
- **XMLQUERY as a subquery**: An XMLQUERY expression that acts as a subquery can be restricted by statements that restrict subqueries.

XMLROW

The XMLROW function returns an XML value with a single XQuery document node containing one top-level element node.



Notes:

[1] The same clause must not be specified more than once.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

expression
The content of each generated XML element node is specified by an expression. The data type of the expression cannot be a BINARY type, a VARBINARY type, or a structured type (SQLSTATE 42884). The expression can be any SQL expression. If the expression is not a simple column reference, an element name must be specified.

AS *qname-identifier*
Specifies the XML element name or attribute name as an SQL identifier. The *qname-identifier* must be of the form of an XML qualified name, or QName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope (SQLSTATE 42635). If *qname-identifier* is not specified, *expression* must be a column name (SQLSTATE 42703). The element name or attribute name is created from the column name using the fully escaped mapping from a column name to an QName.

OPTION
Specifies additional options for constructing the XML value. If no OPTION clause is specified, the default behavior applies.

AS ATTRIBUTES
Specifies that each expression is mapped to an attribute value with column name or *qname-identifier* serving as the attribute name.

ROW *row-name*
Specifies the name of the element to which each row is mapped. If this option is not specified, the default element name is "row".

Notes

By default, each row in the result set is mapped to an XML value as follows:

- Each row is transformed into an XML element named "row" and each column is transformed into a nested element with the column name as the element name.
- The null handling behavior is NULL ON NULL. A null value in a column maps to the absence of the subelement. If all column values are null, a null value is returned by the function.
- The binary encoding scheme for BLOB and FOR BIT DATA data types is base64Binary encoding.
- A document node will be added implicitly to the row element to make the XML result a well-formed single-rooted XML document.

Examples

Assume the following table T1 with columns C1 and C2 that contain numeric data stored in a relational format:

```
C1          C2
----------- -----------
          1           2
          -           2
          1           -
          -           -

  4 record(s) selected.
```

- *Example 1*: The following example shows an XMLRow query and output fragment with default behavior, using a sequence of row elements to represent the table:

```
SELECT XMLROW(C1, C2) FROM T1
```

```
<row><C1>1</C1><C2>2</C2></row>
<row><C2>2</C2></row>
<row><C1>1</C1></row>

  4 record(s) selected.
```

- *Example 2*: The following example shows an XMLRow query and output fragment with attribute centric mapping. Instead of appearing as nested elements as in the previous example, relational data is mapped to element attributes:

```
SELECT XMLROW(C1, C2 OPTION AS ATTRIBUTES) FROM T1
```

```
<row C1="1" C2="2"/>
<row C2="2"/>
<row C1="1"/>

  4 record(s) selected.
```

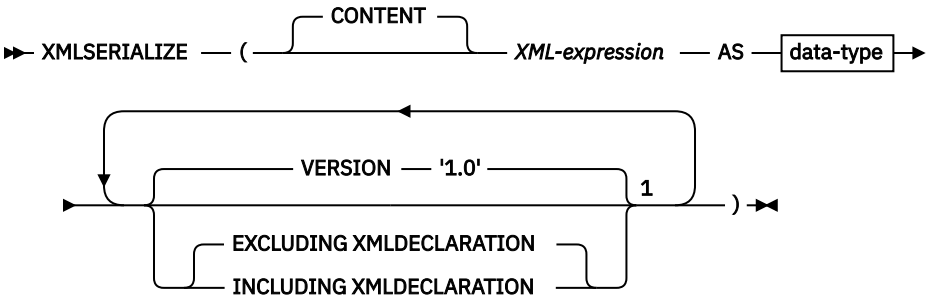
- *Example 3*: The following example shows an XMLRow query and output fragment with the default <row> element replaced by <entry>. Columns C1 and C2 are returned as <column1> and <column2> elements, and the total of C1 and C2 is returned inside a <total> element:

```
SELECT XMLROW(
   C1 AS "column1", C2 AS "column2",
   C1+C2 AS "total" OPTION ROW "entry")
FROM T1
```

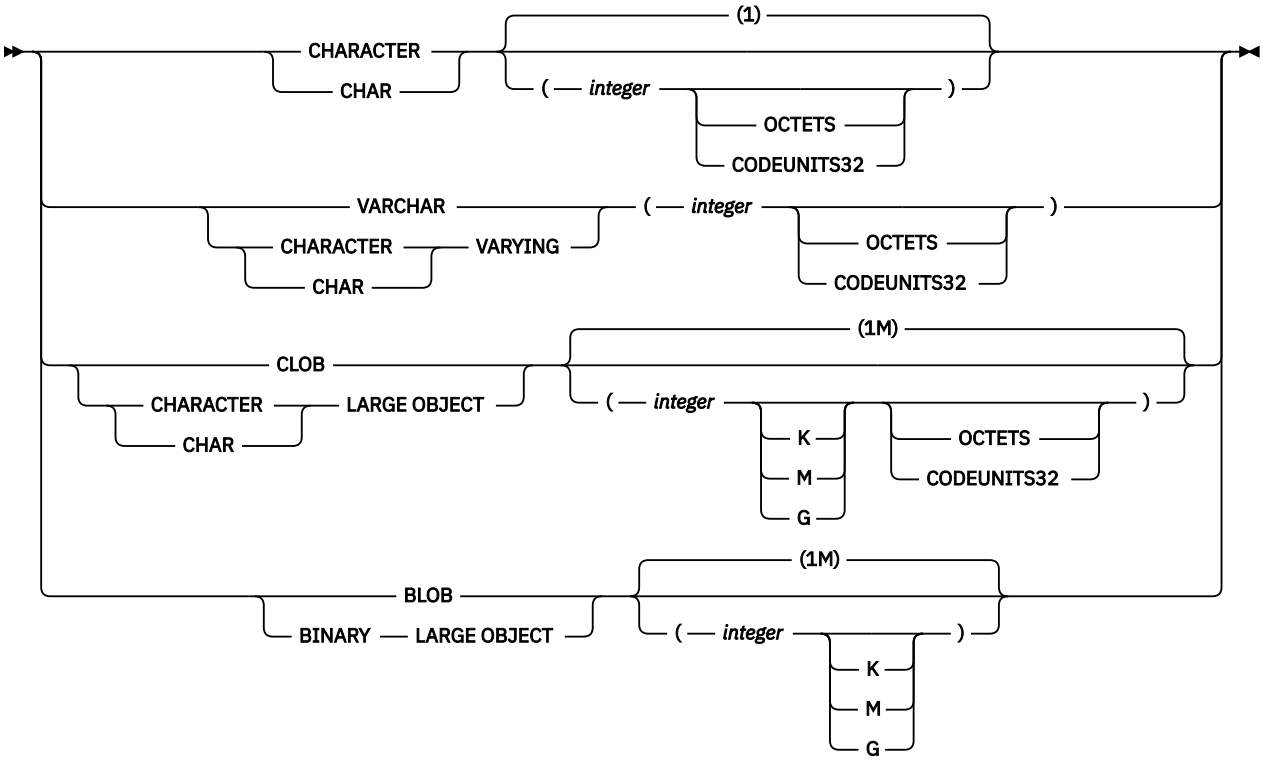
```
<entry><column1>1</column1><column2>2</column2><total>3</total></entry>
<entry><column2>2</column2></entry>
<entry><column1>1</column1></entry>

  4 record(s) selected.
```

XMLSERIALIZE

The XMLSERIALIZE function returns a serialized XML value of the specified data type generated from the *XML-expression* argument.



data-type



Notes:

[1] The same clause must not be specified more than once.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

CONTENT
Specifies that any XML value can be specified and the result of the serialization is based on this input value.

XML-expression
Specifies an expression that returns a value of data type XML. The XML sequence value must not contain an item that is an attribute node (SQLSTATE 2200W). This is the input to the serialization process.

AS *data-type*
Specifies the result type. The implicit or explicit length attribute of the specified result data type must be sufficient to contain the serialized output (SQLSTATE 22001).

VERSION '1.0'
Specifies the XML version of the serialized value. The only version supported is '1.0' which must be specified as a string constant (SQLSTATE 42815).

EXCLUDING XMLDECLARATION or INCLUDING XMLDECLARATION
Specifies whether an XML declaration is included in the result. The default is EXCLUDING XMLDECLARATION.

EXCLUDING XMLDECLARATION
Specifies that an XML declaration is not included in the result.

INCLUDING XMLDECLARATION
Specifies that an XML declaration is included in the result. The XML declaration is the string '<?xml version="1.0" encoding="UTF-8"?>'.

AS *data-type*
Specifies the result data type. The implicit or explicit length attribute of the specified result data type must be sufficient to contain the transformed output (SQLSTATE 22001).

If either the *xml-document* argument or the *xsl-stylesheet* argument is null, the result will be null.

Code page conversion might occur when storing any of the previously mentioned documents in a CHAR, VARCHAR, or CLOB column, which might result in a character loss.

The result has the data type specified by the user. An XML sequence is effectively converted to have a single document node by applying XMLDOCUMENT to *XML-expression* before serializing the resulting XML nodes. If the result of *XML-expression* can be null, the result can be null; if the result of *XML-expression* is null, the result is the null value.

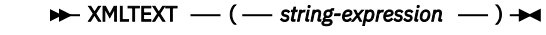
Notes

- **Encoding in the serialized result**: The serialized result is encoded with UTF-8. If XMLSERIALIZE is used with a character data type, and the INCLUDING XMLDECLARATION clause is specified, the resulting character string containing serialized XML might have an XML encoding declaration that does not match the code page of the character string. Following serialization, which uses UTF-8 encoding, the character string that is returned from the server to the client is converted to the code page of the client, and that code page might be different from UTF-8.

  Therefore, applications should avoid direct use of XMLSERIALIZE INCLUDING XMLDECLARATION that return character string types and should retrieve XML values directly into host variables to maintain the match between the external code page and the encoding in the XML declaration. If XMLSERIALIZE must be used in this situation, a BLOB type should be specified to avoid code page conversion.
- **Syntax alternative**: XML2CLOB(*XML-expression*) can be specified in place of XMLSERIALIZE(*XML-expression* AS CLOB(2G)). It is supported only for compatibility with previous Db2 releases.

XMLTEXT

The XMLTEXT function returns an XML value with a single XQuery text node having the input argument as the content.



The schema is SYSIBM. The function name cannot be specified as a qualified name.

string-expression
An expression whose value has a character string type: CHAR, VARCHAR or CLOB.

The data type of the result is XML. If the result of *string-expression* can be null, the result can be null; if the input value is null, the result is the null value. If the result of *string-expression* is an empty string, the result value is an empty text node.

Examples

- *Example 1*: Create a simple XMLTEXT query.

```
   VALUES(
     XMLTEXT(
       'The stock symbol for Johnson&Johnson is JNJ.'
     )
   )
```

This query produces the following serialized result:

```
1
--------------------------------------------------
The stock symbol for Johnson&amp;Johnson is JNJ.
```

Note that the '&' sign is mapped to '&' when a text node is serialized.

- *Example 2*: Use XMLTEXT with XMLAGG to construct mixed content. Suppose that the content of table T is as follows:

```
seqno  plaintext
                                                    emphtext
------ ------------------------------------------------------------
-------------
1      This query shows how to construct
 mixed content
2       using XMLAGG and XMLTEXT. Without
 XMLTEXT
3      XMLAGG will not have text nodes to group with other nodes,
 mixed content
       therefore, cannot generate
```

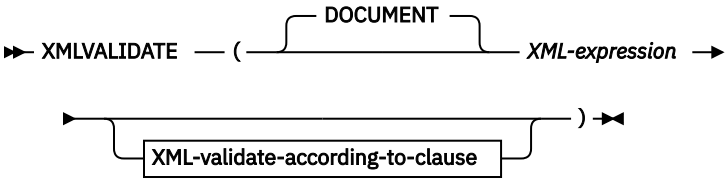
```
   SELECT
XMLELEMENT(
     NAME "para", XMLAGG(
       XMLCONCAT(
         XMLTEXT(
           PLAINTEXT
         ),
         XMLELEMENT(
           NAME "emphasis", EMPHTEXT
         )
       )
       ORDER BY SEQNO
     ), '.'
   ) AS "result"
   FROM T
```

This query produces the following result:

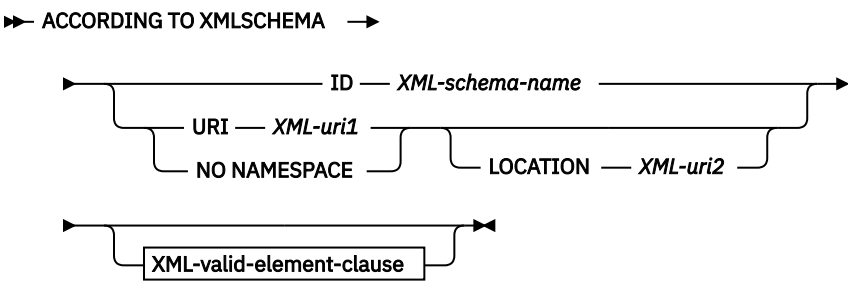
```
result
--------------------------------------------------------------------------------
<para>This query shows how to construct <emphasis>mixed content</emphasis>
using XMLAGG and XMLTEXT. Without <emphasis>XMLTEXT</emphasis>
, XMLAGG
will not have text nodes to group with other nodes, therefore, cannot
generate
<emphasis>mixed content</emphasis>.</para>
```

XMLVALIDATE

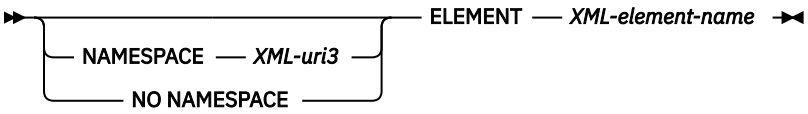
The XMLVALIDATE function returns a copy of the input XML value augmented with information obtained from XML schema validation, including default values.



XML-validate-according-to-clause



XML-valid-element-clause



The schema is SYSIBM. The function name cannot be specified as a qualified name.

DOCUMENT
Specifies that the XML value resulting from *XML-expression* must be a well-formed XML document that conforms to XML Version 1.0 (SQLSTATE 2200M).

XML-expression
An expression that returns a value of data type XML. If *XML-expression* is an XML host variable or an implicitly or explicitly typed parameter marker, the function performs a validating parse that strips ignorable whitespace and the CURRENT IMPLICIT XMLPARSE OPTION setting is not considered.

XML-validate-according-to-clause
Specifies the information that is to be used when validating the input XML value.

ACCORDING TO XMLSCHEMA
Indicates that the XML schema information for validation is explicitly specified. If this clause is not included, the XML schema information must be provided in the content of the *XML-expression* value.

ID *XML-schema-name*
Specifies an SQL identifier for the XML schema that is to be used for validation. The name, including the implicit or explicit SQL schema qualifier, must uniquely identify an existing XML schema in the XML schema repository at the current server. If no XML schema by this name exists in the implicitly or explicitly specified SQL schema, an error is returned (SQLSTATE 42704).

URI *XML-uri1*
Specifies the target namespace URI of the XML schema that is to be used for validation. The value of *XML-uri1* specifies a URI as a character string constant that is not empty. The URI must be the target namespace of a registered XML schema (SQLSTATE 4274A) and, if no LOCATION clause is specified, it must uniquely identify the registered XML schema (SQLSTATE 4274B).

NO NAMESPACE
Specifies that the XML schema for validation has no target namespace. The target namespace URI is equivalent to an empty character string that cannot be specified as an explicit target namespace URI.

LOCATION *XML-uri2*
Specifies the XML schema location URI of the XML schema that is to be used for validation. The value of *XML-uri2* specifies a URI as a character string constant that is not empty. The XML schema location URI, combined with the target namespace URI, must identify a registered XML schema (SQLSTATE 4274A), and there must be only one such XML schema registered (SQLSTATE 4274B).

XML-valid-element-clause
Specifies that the XML value in *XML-expression* must have the specified element name as the root element of the XML document.

NAMESPACE *XML-uri3* or NO NAMESPACE
Specifies the target namespace for the element that is to be validated. If neither clause is specified, the specified element is assumed to be in the same namespace as the target namespace of the registered XML schema that is to be used for validation.

NAMESPACE *XML-uri3*
Specifies the namespace URI for the element that is to be validated. The value of *XML-uri3* specifies a URI as a character string constant that is not empty. This can be used when the registered XML schema that is to be used for validation has more than one namespace.

NO NAMESPACE
Specifies that the element for validation has no target namespace. The target namespace URI is equivalent to an empty character string which cannot be specified as an explicit target namespace URI.

ELEMENT *xml-element-name*
Specifies the name of a global element in the XML schema that is to be used for validation. The specified element, with implicit or explicit namespace, must match the root element of the value of *XML-expression* (SQLSTATE 22535 or 22536).

The data type of the result is XML. If the value of *XML-expression* can be null, the result can be null; if the value of *XML-expression* is null, the result is the null value.

The XML validation process is performed on a serialized XML value. Because XMLVALIDATE is invoked with an argument of type XML, this value is automatically serialized before validation processing with the follow two exceptions.

- If the argument to XMLVALIDATE is an XML host variable or an implicitly or explicitly typed parameter marker, then a validating parse operation is performed on the input value (no implicit non-validating parse is performed and CURRENT IMPLICIT XMLPARSE OPTION setting is not considered).
- If the argument to XMLVALIDATE is an XMLPARSE invocation using the option PRESERVE WHITESPACE, then the XML parsing and XML validation of the document may be combined into a single validating parse operation.

If an XML value has previously been validated, the annotated type information from the previous validation is removed by the serialization process. However, any default values and entity expansions from the previous validation remain unchanged. If validation is successful, all ignorable whitespace characters are stripped from the result.

To validate a document whose root element does not have a namespace, an xsi:noNamespaceSchemaLocation attribute must be present on the root element.

Notes

- **Determining the XML schema**: The XML schema can be either specified explicitly with the `ACCORDING TO XMLSCHEMA` clause as part of the XMLVALIDATE invocation, or determined implicitly from the XML schema location information in the input XML value. The explicit or implicit XML schema information must identify an XML schema registered in the XML schema repository (SQLSTATE 42704, 4274A, or 22532), and there must be only one such registered XML schema (SQLSTATE 4274B or 22533).

  If an XML schema for validation is explicitly specified with the `ACCORDING TO XMLSCHEMA` clause, the schema location information specified in the input XML value is ignored.

  If the XML schema information is not specified with the `ACCORDING TO XMLSCHEMA` clause, the input XML value must contain XML schema location information (SQLSTATE 2200M). The schema location information in the input XML value, a namespace name, and a schema location specifies the XML schema document in the XML schema repository used for validation.

- **XML schema authorization**: The XML schema used for validation must be registered in the XML schema repository before use. The privileges held by the authorization ID of the statement must include at least one of the following authorities:
  - USAGE privilege on the XML schema that is to be used during validation
  - DBADM authority
- **Using a maxOccurs attribute value that is greater than 5000 in XML schemas:** In Db2 Version 9.7 Fix Pack 1 and later, if an XML schema that is registered in the XSR uses the maxOccurs attribute where the value is greater than 5000, the maxOccurs attribute value is treated as if you specified "unbounded". Because document elements that have a maxOccurs attribute value that is greater than 5000 are processed as if you specified "unbounded", an XML document might pass validation when you use the XMLVALIDATE function even if the number of occurrences of an element exceeds the maximum according to the XML schema that you used to validate the document.

  If you use an XML schema that defines an element that has a maxOccurs attribute value that is greater than 5000 and you want to reject XML documents that have a maxOccurs attribute value greater than 5000, you can define a trigger or procedure to check for that condition. In the trigger or procedure, use an XPath expression to count the number of occurrences of the element and return an error if the number of elements exceeds the maxOccurs attribute value.

  For example, the following trigger ensures that a document never has more than 6500 phone elements:

```
CREATE TRIGGER CUST_INSERT
  AFTER INSERT ON CUSTOMER
  REFERENCING NEW AS NEWROW
  FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  SELECT CASE WHEN X <= 6500 THEN 'OK - Do Nothing'
              ELSE RAISE_ERROR('75000', 'TooManyPhones')  END

  FROM (
    SELECT XMLCAST(XMLQUERY('$INFO/customerinfo/count(phone)') AS INTEGER) AS X
    FROM CUSTOMER
    WHERE CUSTOMER.CID = NEWROW.CID );
END
```

Examples

- *Example 1*: Validate using the XML schema identified by the XML schema hint in the XML instance document.

```
   INSERT INTO T1(XMLCOL)
     VALUES (XMLVALIDATE(?))
```

  Assume that the input parameter marker is bound to an XML value that contains the XML schema information.

```
   <po:order
      xmlns:po="http://my.world.com"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://my.world.com http://my.world.com/world.xsd" >
   ...
   </po:order>
```

  Further, assume that the XML schema that is associated with the target namespace "http://my.world.com" and by schemaLocation hint "http://my.world.com/world.xsd" is found in the XML schema repository.

  Based on these assumptions, the input XML value will be validated according to that XML schema.

- *Example 2*: Validate using the XML schema identified by the SQL name PODOCS.WORLDPO.

```
   INSERT INTO T1(XMLCOL)
     VALUES (
       XMLVALIDATE(
         ? ACCORDING TO XMLSCHEMA ID PODOCS.WORLDPO
```

```
    )
  )
```

Assuming that the XML schema that is associated with SQL name FOO.WORLDPO is found in the XML repository, the input XML value will be validated according to that XML schema.

- *Example 3*: Validate a specified element of the XML value.

```
INSERT INTO T1(XMLCOL)
  VALUES (
    XMLVALIDATE(
      ? ACCORDING TO XMLSCHEMA ID FOO.WORLDPO
      NAMESPACE 'http://my.world.com/Mary'
      ELEMENT "po"
    )
  )
```

Assuming that the XML schema that is associated with SQL name FOO.WORLDPO is found in the XML repository, the XML schema will be validated against the element "po", whose namespace is 'http://my.world.com/Mary'.

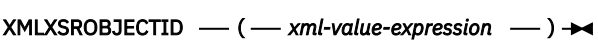
- *Example 4*: XML schema is identified by target namespace and schema location.

```
INSERT INTO T1(XMLCOL)
  VALUES (
    XMLVALIDATE(
      ? ACCORDING TO XMLSCHEMA URI 'http://my.world.com'
      LOCATION 'http://my.world.com/world.xsd'
    )
  )
```

Assuming that an XML schema associated with the target namespace "http://my.world.com" and by schemaLocation hint "http://my.world.com/world.xsd" is found in the XML schema repository, the input XML value will be validated according to that schema.

XMLXSROBJECTID

The XMLXSROBJECTID function returns the XSR object identifier of the XML schema used to validate the XML document specified in the argument. The XSR object identifier is returned as a BIGINT value and provides the key to a single row in SYSCAT.XSROBJECTS.



The schema is SYSIBM.

xml-value-expression
Specifies an expression that results in a value with a data type of XML. The resulting XML value must be an XML sequence with a single item that is an XML document or the null value (SQLSTATE 42815). If the argument is null, the function returns null. If *xml-value-expression* does not specify a validated XML document, the function returns 0.

Notes

- The XML schema corresponding to an XSR object ID returned by the function might no longer exist, because an XML schema can be dropped without affecting XML values that were validated using the XML schema. Therefore, queries that use the XSR object ID to fetch further XML schema information from the catalog views might return an empty result set.
- Applications can use the XSR object identifier to retrieve additional information about the XML schema. For example, the XSR object identifier can be used to return the individual XML schema documents that make up a registered XML schema from SYSCAT.SYSXSROBJECTCOMPONENTS, and the hierarchy of XML schema documents in the XML schema from SYSCAT.XSROBJECTHIERARCHIES.

Examples

- *Example 1*: Retrieve the XML schema identifier for the XML document XMLDOC stored in the table MYTABLE.

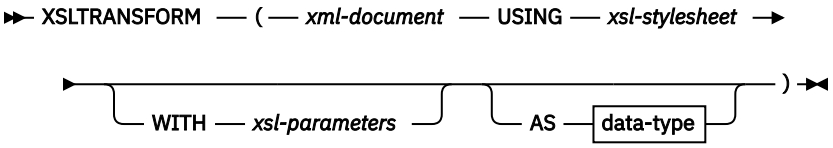
```
SELECT XMLXSROBJECTID(XMLDOC) FROM MYTABLE
```

- *Example 2*: Retrieve the XML schema documents associated with the XML document that has a specific ID (in this case where DOCKEY = 1) in the table MYTABLE, including the hierarchy of the XML schema documents that make up the XML schema.

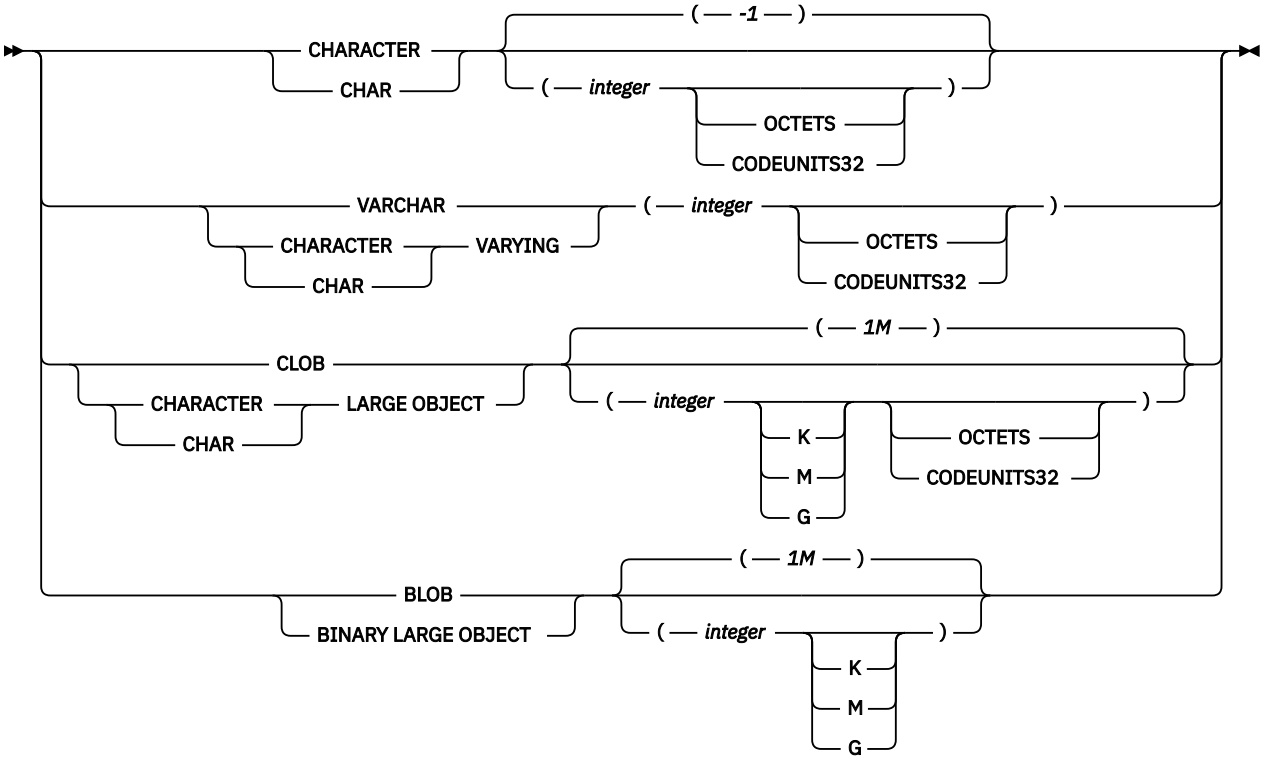
```
SELECT H.HTYPE, C.TARGETNAMESPACE, C.COMPONENT
FROM SYSCAT.XSROBJECTCOMPONENTS C, SYSCAT.XSROBJECTHIERARCHIES H
WHERE C.OBJECTID =
  (SELECT XMLXSROBJECTID(XMLDOC) FROM MYTABLE
   WHERE DOCKEY = 1)
AND C.OBJECTID = H.OBJECTID
```

XSLTRANSFORM

Use XSLTRANSFORM to convert XML data into other formats, including the conversion of XML documents that conform to one XML schema into documents that conform to another schema.



data-type



The schema is SYSIBM. This function cannot be specified as a qualified name.

The XSLTRANSFORM function transforms an XML document into a different data format. The data can be transformed into any form possible for the XSLT processor, including but not limited to XML, HTML, or plain text.

All paths used by XSLTRANSFORM are internal to the database system. This command cannot currently be used directly with files or stylesheets residing in an external file system.

xml-document
An expression that returns a well-formed XML document with a data type of XML, CHAR, VARCHAR, CLOB, or BLOB. This is the document that is transformed using the XSL style sheet specified in *xsl-stylesheet*.

The XML document must at minimum be single-rooted and well-formed.

xsl-stylesheet
An expression that returns a well-formed XML document with a data type of XML, CHAR, VARCHAR, CLOB, or BLOB. The document is an XSL style sheet that conforms to the W3C XSLT Version 1.0 Recommendation. Style sheets incorporating XQUERY statements or the `xsl:include` declaration are not supported. This stylesheet is applied to transform the value specified in *xml-document*.

xsl-parameters
An expression that returns a well-formed XML document or null with a data type of XML, CHAR, VARCHAR, CLOB, or BLOB. This is a document that provides parameter values to the XSL stylesheet specified in *xsl-stylesheet*. The value of the parameter can be specified as an attribute, or as a text node.

The syntax of the parameter document is as follows:

```
<params xmlns="http://www.ibm.com/XSLTransformParameters">
<param name="..." value="..."/>
<param name="...">enter value here</param>
        ...
</params>
```

The stylesheet document must have `xsl:param` element(s) in it with name attribute values that match the ones specified in the parameter document.

AS *data-type*
Specifies the result data type. The implicit or explicit length attribute of the specified result data type must be sufficient to contain the transformed output (SQLSTATE 22001). The default result data type is CLOB(2G OCTETS) except in a Unicode database where the string units of the environment is set to CODEUNITS32 when the default is CLOB(536870911 CODEUNITS32).

If either the *xml-document* argument or the *xsl-stylesheet* argument is null, the result will be null.

Code page conversion might occur when storing any of the previously mentioned documents in a CHAR, VARCHAR, or CLOB column, which might result in a character loss.

Example

This example illustrates how to use XSLT as a formatting engine. To get set up, first create the XML_TAB table and insert a row that includes an XML document and an XSL style sheet into the table.

```
CREATE TABLE XML_TAB (DOC_ID INTEGER, XML_DOC XML, XSL_DOC XML);

INSERT INTO XML_TAB VALUES
(1,
'<?xml version="1.0"?>
 <students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation = "/home/steffen/xsd/xslt.xsd">
  <student studentID="1" firstName="Steffen" lastName="Siegmund"
    age="23" university="Rostock"/>
 </students>',
'<?xml version="1.0" encoding="UTF-8"?>
 <xsl:stylesheet version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:param name="headline"/>
 <xsl:param name="showUniversity"/>
 <xsl:template match="students">
  <html>
  <head/>
  <body>
    <h1><xsl:value-of select="$headline"/></h1>
      <table border="1">
```

```
        <th>
         <tr>
          <td width="80">StudentID</td>
          <td width="200">First Name</td>
          <td width="200">Last Name</td>
          <td width="50">Age</td>
        <xsl:choose>
         <xsl:when test="$showUniversity =''true''">
          <td width="200">University</td>
         </xsl:when>
        </xsl:choose>
         </tr>
        </th>
       <xsl:apply-templates/>
      </table>
  </body>
  </html>
 </xsl:template>
 <xsl:template match="student">
         <tr>
          <td><xsl:value-of select="@studentID"/></td>
          <td><xsl:value-of select="@firstName"/></td>
          <td><xsl:value-of select="@lastName"/></td>
          <td><xsl:value-of select="@age"/></td>
        <xsl:choose>
         <xsl:when test="$showUniversity = ''true''">
          <td><xsl:value-of select="@university"/></td>
         </xsl:when>
        </xsl:choose>
        </tr>
 </xsl:template>
 </xsl:stylesheet>'
);
```

Next, call the XSLTRANSFORM function to convert the XML data into HTML and display it.

```
SELECT XSLTRANSFORM (XML_DOC USING XSL_DOC AS CLOB(1M)) FROM XML_TAB;
```

The result is this document:

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
<h1></h1>
<table border="1">
<th>
<tr>
<td width="80">StudentID</td>
<td width="200">First Name</td>
<td width="200">Last Name</td>
<td width="50">Age</td>
</tr>
</th>
 <tr>
<td>1</td>
<td>Steffen</td><td>Siegmund</td>
<td>23</td>
</tr>
              </table>
</body>
</html>
```

In this example, the output is HTML and the parameters influence only what HTML is produced and what data is brought over to it. As such it illustrates the use of XSLT as a formatting engine for end-user output.

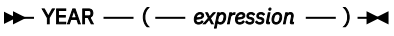
Usage note

There are many methods you can use to transform XML documents including the XSLTRANSFORM function, an XQuery update expression, and XSLT processing by an external application server. For documents stored in an XML column of a database table, many transformations can be performed more efficiently by using an XQuery update expression rather than with XSLT because XSLT always requires parsing of the XML documents that are being transformed. If you decide to transform XML documents

with XSLT, you should make careful decisions about whether to transform the document in the database or in an application server.

YEAR

The YEAR function returns the year part of a value.



The schema is SYSIBM.

expression
An expression that returns a value of one of the following built-in data types: DATE, TIMESTAMP, date duration, timestamp duration, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument specified:

- If the argument is a DATE, TIMESTAMP, or valid string representation of a date or timestamp:
  - The result is the year part of the value, which is an integer between 1 and 9999.
- If the argument is a date duration or timestamp duration:
  - The result is the year part of the value, which is an integer between -9999 and 9999. A nonzero result has the same sign as the argument.

Examples

- *Example 1*: Select all the projects in the PROJECT table that are scheduled to start (PRSTDATE) and end (PRENDATE) in the same calendar year.

```
SELECT * FROM PROJECT
  WHERE YEAR(PRSTDATE) = YEAR(PRENDATE)
```

- *Example 2*: Select all the projects in the PROJECT table that are scheduled to take less than one year to complete.

```
SELECT * FROM PROJECT
  WHERE YEAR(PRENDATE - PRSTDATE) < 1
```

YEARS_BETWEEN

The YEARS_BETWEEN function returns the number of full years between the specified arguments.

The schema is SYSIBM.

expression1
An expression that specifies the first datetime value to compute the number of full years between two datetime values. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression1* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

expression2
An expression that specifies the second datetime value to compute the number of full years between two datetime values. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or

VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression2* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

If there is less than a full year between *expression1* and *expression2*, the result is zero. If *expression1* is later than *expression2*, the result is positive. If *expression1* is earlier than *expression2*, the result is negative. If *expression1* or *expression2* contains time information, this information is also used to determine the number of full years. If *expression1* or *expression2* does not contain time information, a time of midnight (00.00.00) is used for the argument that is missing time information.

The result of the function is an INTEGER. If either argument can be null, the result can be null. If either argument is null, the result is the null value.

The YEARS_BETWEEN function is a synonym of the following expression:

```
YEAR( TIMESTAMP( expression1, 12 ) - TIMESTAMP( expression2, 12 ) )
```

Examples

1. Set the host variable NUM_YEARS with the number of full years between 2013-02-28 and 2012-02-29.

   ```
   SET :NUM_YEARS = YEARS_BETWEEN(DATE '2013-02-28', DATE '2012-02-29')
   ```

   The host variable NUM_YEARS is set to 0 because there is 1 day less than a full year between the arguments due to the existence of February 29, 2012.
2. Set the host variable NUM_YEARS with the number of full years between 2013-12-31 and 2001-01-01.

   ```
   SET :NUM_YEARS = YEARS_BETWEEN(DATE '2013-12-31', DATE '2001-01-01')
   ```

   The host variable NUM_YEARS is set to 12 because there is 1 day less than 13 full years between the arguments. It is positive because the first argument is later than the second argument.
3. Set the host variable NUM_YEARS with the number of full years between 2001-01-01-00.00.00 and 2013-12-31-23.59.59.

   ```
   SET :NUM_YEARS = YEARS_BETWEEN(TIMESTAMP '2001-01-01-00.00.00',
      TIMESTAMP '2013-12-31-23.59.59')
   ```

   The host variable NUM_YEARS is set to -12 because there is 1 day less than 13 full years between the arguments. It is negative because the first argument is earlier than the second argument.

YMD_BETWEEN

The YMD_BETWEEN function returns a numeric value that specifies the number of full years, full months, and full days between two datetime values.

The schema is SYSIBM.

expression1
An expression that specifies the first datetime value to compute the number of full years, full months, and full days between two datetime values. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression1* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

expression2
An expression that specifies the second datetime value to compute the number of full years, full months, and full days between two datetime values. The expression must return a value that is a DATE, TIMESTAMP, CHAR, or VARCHAR data type. In a Unicode database, the expression can also be a GRAPHIC or VARGRAPHIC data type. CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are supported by using implicit casting. If *expression2* is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it must be a valid string that is accepted by the TIMESTAMP scalar function.

If there is less than a full day between *expression1* and *expression2*, the result is zero. If *expression1* is later than *expression2*, the result is positive. If *expression1* is earlier than *expression2*, the result is negative. If *expression1* or *expression2* contains time information, this information is also used to determine the number of full years, full months, and full days. If *expression1* or *expression2* does not contain time information, a time of midnight (00.00.00) is used for the argument that is missing time information.

The result of the function is an INTEGER. If either argument can be null, the result can be null. If either argument is null, the result is the null value.

The YMD_BETWEEN function is a synonym of the following expression:

```
INTEGER( ( TIMESTAMP(expression1, 12) – TIMESTAMP(expression2, 12) ) / 1000000 )
```

The result is the integer representation of the extraction of the year, month, and day components of a timestamp duration.

Examples

1. Set the host variable YMD to the number of full years, full months, and full days between 2013-09-23-23.59.59.123456789012 and 2013-09-24-23.59.59.123456789011.

   ```
   SET :YMD = YMD_BETWEEN(TIMESTAMP '2013-09-24-23.59.59.123456789011',
      TIMESTAMP '2013-09-23-23.59.59.123456789012')
   ```

   The host variable YMD is set to 0 because there are 0.000000000001 seconds less than a full day between the arguments. It is positive because the first argument is later than the second argument.
2. Set the host variable YMD to the number of full years, full months, and full days between 2013-09-23-23.59.59.123456789012 and 2013-09-24-23.59.59.123456789012.

   ```
   SET :YMD = YMD_BETWEEN(TIMESTAMP '2013-09-24-23.59.59.123456789012',
      TIMESTAMP '2013-09-23-23.59.59.123456789012')
   ```

   The host variable YMD is set to 1 because there is exactly 1 day between the arguments. It is positive because the first argument is later than the second argument.
3. Set the host variable YMD to the number of full years, full months, and full days between 2013-09-23-23.59.59.123456789012 and 2016-03-01-23.59.59.123456789011.

   ```
   SET :YMD = YMD_BETWEEN(TIMESTAMP '2013-09-23-23.59.59.123456789012',
      TIMESTAMP '2016-03-01-23.59.59.123456789011')
   ```

   The host variable YMD is set to -20507 because there are 0.000000000001 seconds less than 2 full years, 5 full months, and 8 full days between the arguments. It is negative because the first argument is earlier than the second argument.

Table functions

Table functions return columns of a table, resembling a table created through a simple CREATE TABLE statement.

A table function can be used only in the FROM clause of a statement.

Table functions can be qualified with a schema name.

BASE_TABLE

The BASE_TABLE function returns both the object name and schema name of the object found after any alias chains have been resolved.

The schema is SYSPROC.

The specified objectname (and objectschema) are used as the starting point of the resolution. If the starting point does not refer to an alias, the schema name and the unqualified name of the starting point are returned. The function returns a single row table consisting of the following columns:

Table 107. Information returned by the BASE_TABLE function

Column name	Data type	Description
BASESCHEMA	VARCHAR(128)	Schema name of the object found after any alias chains have been resolved. Matches objectschema if no matching alias was found.
BASENAME	VARCHAR(128)	Unqualified name of the object found after any alias chains have been resolved. Matches objectname if no matching alias was found. The name may identify a table, a view, or an undefined object.

objectschema
A character expression representing the schema used to qualify the supplied *objectname* value before resolution. *objectschema* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.

objectname
A character expression representing the unqualified name to be resolved. *objectname* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.

Note: The BASE_TABLE table function improves performance in partitioned database configurations by avoiding the unnecessary communication that occurs between the coordinator partition and catalog partition when using the TABLE_SCHEMA and TABLE_NAME scalar functions.

Example

The following statement using the TABLE_SCHEMA and TABLE_NAME functions is written as:

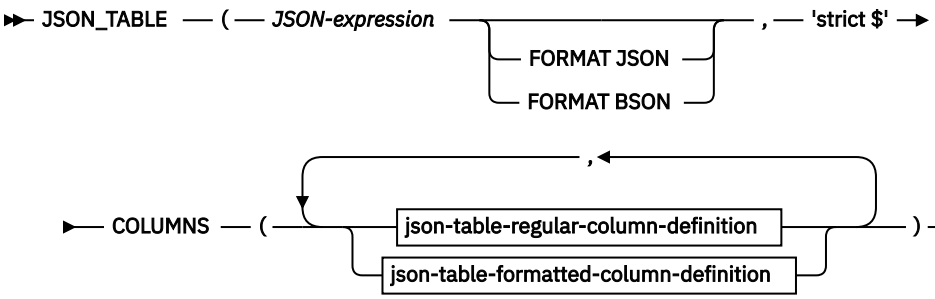
```
SELECT COLCOUNT INTO :H00030
FROM SYSCAT.TABLES
WHERE OWNER = TABLE_SCHEMA(:H00031 ,:H00032 )
AND TABNAME = TABLE_NAME(:H00031 ,:H00032 )
```

The equivalent statement using the BASE_TABLE function can be written as:

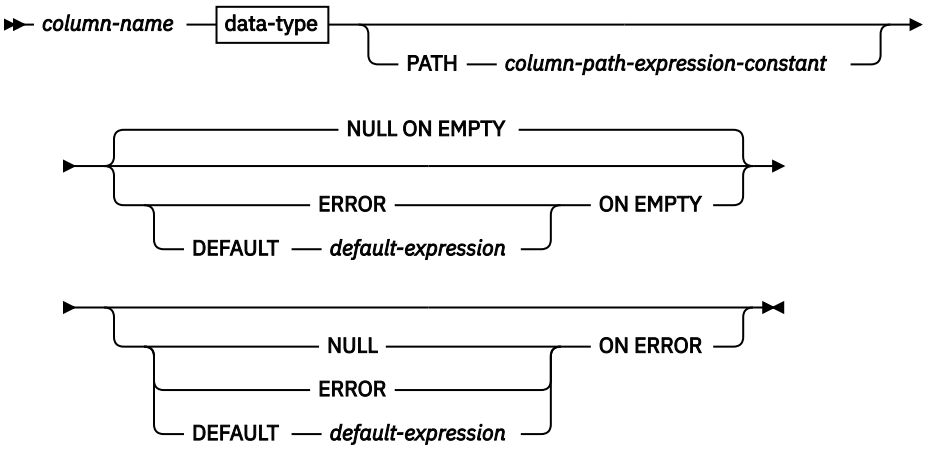
```
SELECT COLCOUNT INTO :H00030
FROM SYSCAT.TABLES A, TABLE(SYSPROC.BASE_TABLE(:H00032, :H00031)) AS B
WHERE A.OWNER = B.BASESCHEMA
AND A.TABNAME = B.BASENAME
```

JSON_TABLE

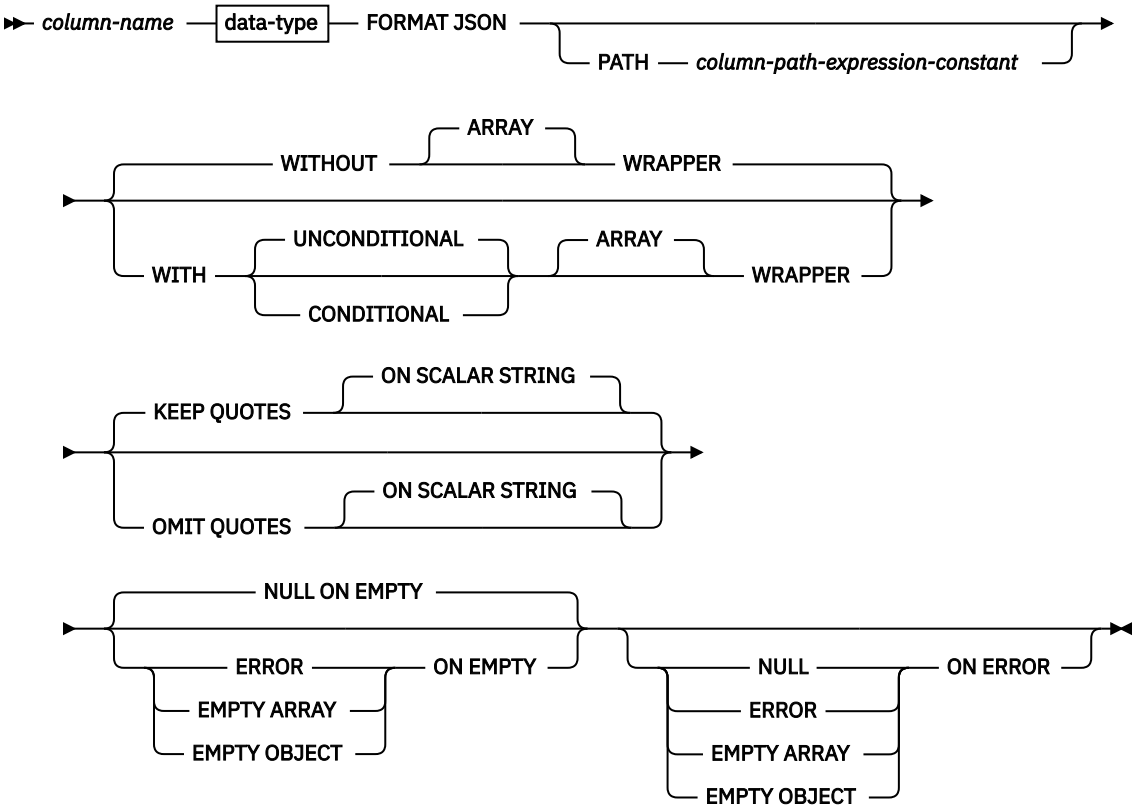
The JSON_TABLE table function returns a result table from the evaluation of SQL/JSON path expressions. Each item in the result sequence of the row SQL/JSON path expression represents one or more rows in the result table.



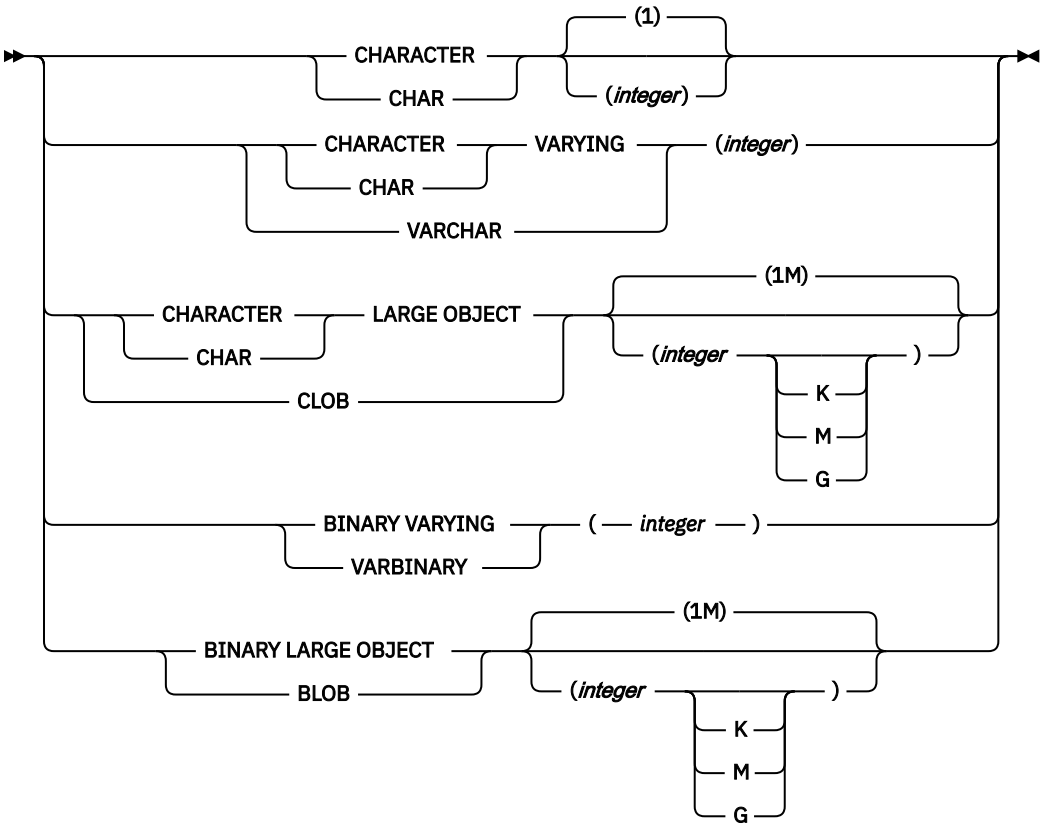
json-table-regular-column-definition



json-table-formatted-column-definition



data-type



The schema is SYSIBM. The function name cannot be specified as a qualified name.

JSON-expression
An expression that returns a value that is a built-in string data type, except the following data types (SQLSTATE 42815):

- GRAPHIC
- VARGRAPHIC
- DBCLOB
- BINARY
- CHAR FOR BIT DATA
- VARCHAR FOR BIT DATA
- A user-defined type that is sourced on any of the previously listed data types

If a character value is returned, it must contain correctly formatted JSON data (SQLSTATE 22032). If a binary data type is returned, it is interpreted according to the explicit or implicit FORMAT clause.

FORMAT JSON
JSON-expression is formatted as JSON data.

If *JSON-expression* is a character string data type, it is treated as JSON data.

If *JSON-expression* is a binary string data type, it is interpreted as UTF-8 data.

FORMAT BSON
Specifies that *JSON-expression* is formatted as the BSON representation of JSON data (SQLSTATE 22032). *JSON-expression* must be a binary string data type (SQLSTATE 42815).

'strict $'
Specifies that an error is reported when the specified path expression cannot be used to navigate the current JSON document from the start of the context item. The error is handled according to the current ON ERROR clause.

COLUMNS
Specifies the output columns of the result table, including the column name, data type, and how the column value is computed for each row. The sum of the result column lengths cannot exceed 64 KB.

json-table-regular-column-definition
Specifies an output column of the result table, including the column name, data type, and an SQL/JSON path expression to extract the value from the sequence item for the row.

column-name
Specifies the name of the column in the result table. The name cannot be qualified and the same name cannot be used for more than one column of the result table (SQLSTATE 42711).

data-type
Specifies the data type of the column.

See “CREATE TABLE ” on page 1351 for the description of built-in data types.

PATH *column-path-expression-constant*
Specifies a character string constant that is interpreted as an SQL/JSON path. The *column-path-expression-constant* specifies an SQL/JSON path expression that determines the column value regarding an item that is the result of evaluating the SQL/JSON path expression in *sql-json-path-expression*.

For more information about the content of an SQL/JSON path expression, see “sql-json-path-expression” on page 179.

Given an item from the result of processing the *sql-json-path-expression* as the externally provided context item, the *column-path-expression-constant* is evaluated and returns an output sequence. The column value is determined based on this output sequence as follows:

- If an empty sequence is returned, the ON EMPTY clause provides the value of the column.
- If ERROR ON EMPTY is specified, an error is returned.

- If an empty sequence is returned and the ON EMPTY clause is not specified, the null value is assigned to the column.
- If a single element sequence is returned and the type of the element is not a JSON array or a JSON object, the value is converted to the data type that was specified for the column.
- If a single element sequence is returned and the type of the element is a JSON array or a JSON object, an error is returned.
- If a sequence with more than one element is returned, an error is returned.
- If an error occurs, the ON ERROR clause specifies the value of the column.

The value of *column-path-expression-constant* must not be an empty string or a string of all blanks. If the PATH clause is not specified, the *column-path-expression-constant* is defined as `'$.'` prefixed to *column-name*.

ON EMPTY
Specifies the behavior when an empty sequence is returned for the column.

NULL ON EMPTY
An SQL null value is returned. This clause is the default.

ERROR ON EMPTY
An error is returned.

DEFAULT *default-expression* ON EMPTY
The value that is specified by *default-expression* is returned. The data type of *default-expression* must be the same as the return data type (SQLSTATE 42815).

ON ERROR
Specifies the behavior when an error is returned for the column. If this clause is not specified, the behavior specified for the table-level ON ERROR clause is followed.

NULL ON ERROR
A null value is returned.

ERROR ON ERROR
An error is returned.

DEFAULT *default-expression* ON ERROR
The value that is specified by *default-expression* is returned. The data type of *default-expression* must be the same as the return data type (SQLSTATE 42815).

json-table-formatted-column-definition
Specifies an output column of the result table. The definition includes the column name, data type, and an SQL/JSON path expression. This definition is used to extract the value from the sequence item for the row. The extracted value is formatted as a JSON string.

column-name
Specifies the name of the column in the result table. The name cannot be qualified and the same name cannot be used for more than one column of the result table (SQLSTATE 42711).

data-type
Specifies the data type of the column. The data type can be CHAR, VARCHAR, CLOB, VARBINARY, or BLOB (SQLSTATE 42815).

See “CREATE TABLE ” on page 1351 for the description of built-in data types.

FORMAT JSON
Indicates that the data that is retrieved is formatted as a JSON string.

PATH *column-path-expression-constant*
Specifies a character string constant that is interpreted as an SQL/JSON path.

The *column-path-expression-constant* specifies an SQL/JSON path expression that determines the column value regarding the result of evaluating the SQL/JSON path expression in *sql-json-path-expression*.

For more information about the content of an SQL/JSON path expression, see "sql-json-path-expression" on page 179.

Given an item from the result of processing the *sql-json-path-expression* as the externally provided context item, the *column-path-expression-constant* is evaluated and returns an output sequence. The column value is determined based on this output sequence as follows:

- If an empty sequence is returned, the ON EMPTY clause provides the value of the column.
- If ERROR ON EMPTY is specified, an error is returned.
- If an empty sequence is returned and the ON EMPTY clause is not specified, the null value is assigned to the column.
- If an error occurs, the ON ERROR clause specifies the value of the column.

The value for *column-path-expression-constant* must not be an empty string or a string of all blanks. If the PATH clause is not specified, the *column-path-expression-constant* is defined as `'$.'` prefixed to *column-name*.

WITHOUT ARRAY WRAPPER or WITH ARRAY WRAPPER
Specifies whether the output value is wrapped in a JSON array.

WITHOUT ARRAY WRAPPER
Indicates that the result is not wrapped. This clause is the default. Using a strict SQL/JSON path definition that resolves to a sequence of two or more SQL/JSON elements results in an error (SQLSTATE 2203A). Using a lax SQL/JSON path definition with the ON EMPTY that resolves to a sequence of two or more SQL/JSON elements results in an error (SQLSTATE 22035).

WITH UNCONDITIONAL ARRAY WRAPPER
Indicates that the result is enclosed in square brackets to create a JSON array.

WITH CONDITIONAL ARRAY WRAPPER
Indicates that the result is enclosed in square brackets to create a JSON array for either of the following scenarios:

- More than one SQL/JSON element is returned.
- A single SQL/JSON element that is not a JSON array or a JSON object is returned.

KEEP QUOTES or OMIT QUOTES
Specifies whether the surrounding quotation marks are removed when a scalar string is returned.

KEEP QUOTES
Quotation marks are not removed from scalar strings. This clause is the default.

OMIT QUOTES
Quotation marks are removed from scalar strings. When OMIT QUOTES is specified, the WITH ARRAY WRAPPER clause cannot be specified (SQLSTATE 42601).

ON EMPTY
Specifies the behavior when an empty sequence is returned for a column.

NULL ON EMPTY
An SQL null value is returned. This clause is the default.

ERROR ON EMPTY
An error is returned.

EMPTY ARRAY ON EMPTY
An empty JSON array is returned.

EMPTY OBJECT ON EMPTY
An empty JSON object is returned.

ON ERROR
Specifies the behavior when an error is returned for the column. If this clause is not specified, the behavior specified for the table-level ON ERROR clause is followed.

NULL ON ERROR
An SQL null value is returned. This clause is the default.

ERROR ON ERROR
An error is returned.

EMPTY ARRAY ON ERROR
An empty JSON array is returned.

EMPTY OBJECT ON ERROR
An empty JSON object is returned.

ERROR ON ERROR
An error is returned when a table level error is encountered.

Notes

- If parameter markers are not explicitly cast to a supported data type, an error is returned (SQLSTATE 42815)

Example

1. This example is based on the following JSON document:

```
{
  "id" : 901,
  "firstname" : "John",
  "lastname"  : "Doe",
  "phoneno"   : "555-3762"
}
```

List the employee ID, given name, surname, and phone number:

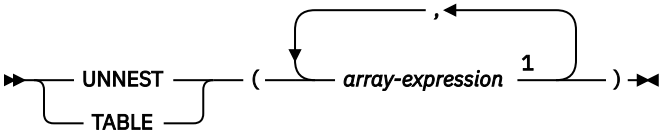
```
SELECT U."id", U."first name", U."last name", U."phone number"
  FROM EMPLOYEE_TABLE E
    JSON_TABLE(E.jsondoc, 'strict $'
                COLUMNS( "id" INTEGER,
                         "firstname"  VARCHAR(20),
                         "lastname"   VARCHAR(20),
                         "phoneno"    VARCHAR(20))
                        ERROR ON ERROR) AS U
```

This query returns the following table:

id	firstname	lastname	phoneno
901	John	Doe	555-3762

UNNEST

The UNNEST function returns a result table that includes a row for each element of the specified array. If there are multiple ordinary array arguments specified, the number of rows will match the array with the largest cardinality.



Notes:

[1] Only a single *array-expression* argument can be specified if the expression returns an associative array type or an array type with row type elements

The schema is SYSIBM.

array-expression
An expression that returns an array data type. The expression must be one of the following expressions:

- An SQL variable
- An SQL parameter
- A global variable
- A function invocation
- A CAST specification of a parameter marker
- A CASE expression

Names for the result columns produced by the UNNEST function can be provided as part of the *correlation-clause* of the *collection-derived-table* clause.

The UNNEST function can only be used in a *collection-derived-table* clause in a context where arrays are supported (SQLSTATE 42887).

The result table depends on the input arguments.

- If a single *array-expression* that returns an ordinary array is specified:
  - If the array element is not a row data type, the result is a single column table with a column data type that matches the array element data type.
  - If the array element is a row data type, the result is a table with one column for each row field in the element data type. The result table column data types match the corresponding array element row field data types.
- If more than one ordinary array argument is specified and none of the array elements have a row data type, the first array provides the first column in the result table, the second array provides the second column, and so on. The data type of each column matches the data type of the array elements of the corresponding array argument. If the cardinalities of the arrays are not identical, the cardinality of the resulting table is the same as the array with the largest cardinality. The column values in the table are set to the null value for all rows whose array index value is greater than the cardinality of the corresponding array. In other words, if each array is viewed as a table with two columns, one for the array indexes and one for the data, then UNNEST performs an OUTER JOIN among the arrays, using equality on the array indexes as a join predicate.
- If a single *array-expression* that returns an associative array is specified:
  - If the array element is not a row data type, the result is a table with 2 columns where the first column data type matches the array index data type and the second column data type matches the array element data type.
  - If the array element is a row data type, the result is a table with one more column than the number of fields in the row data type, where the first column data type matches the array index data type and the remaining column data types match the array element row field data types.
- An error is returned (SQLSTATE 42884):
  - If more than one associative array argument is specified.
  - If more than one array argument is specified and at least one of the arrays has a element data type that is a row type.
  - If both ordinary array arguments and associative array arguments are specified.

This special table function is only used in *collection-derived-table* of *table-reference* in a FROM clause.

If more than one array is provided and at least one of the arguments is an associative array, an error is returned (SQLSTATE 42884).

If the WITH ORDINALITY clause is used when unnesting an associative array, an error is returned (SQLSTATE 428HT).

Examples

1. Assume the ordinary array variable RECENT_CALLS of array type PHONENUMBERS contains only the three element values 9055553907, 4165554213, and 4085553678.The following query:

```
SELECT T.ID, T.NUM
FROM UNNEST(RECENT_CALLS) WITH ORDINALITY AS T(NUM, ID)
```

   returns a table formatted as follows:

```
ID    NUM
----------
1     9055553907
2     4165554213
3     4085553678
```

2. Return the list of personal phone numbers from the array variable PHONELIST of array type PERSONAL_PHONENUMBERS along with the index string values The following query:

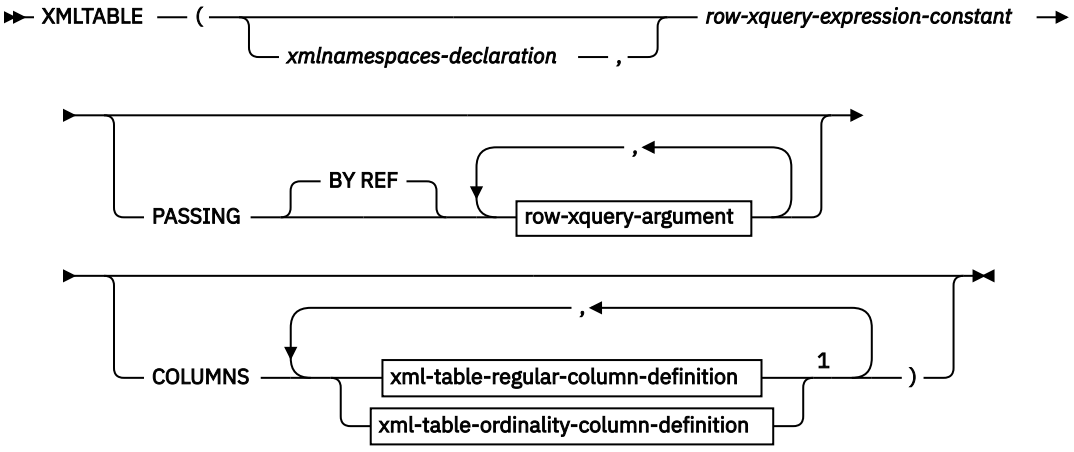
```
SELECT T.ID, T.PHONE
FROM UNNEST(PHONELIST) AS T(ID, PHONE)
```

   returns a table formatted as follows:

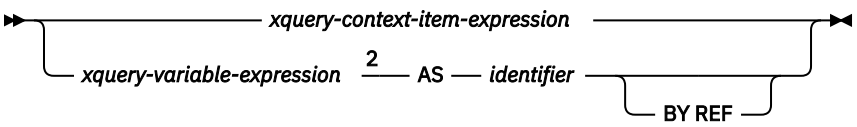
```
ID    PHONE
---------------
Home  4163053745
Work  4163053746
Mom   4164789683
```

XMLTABLE

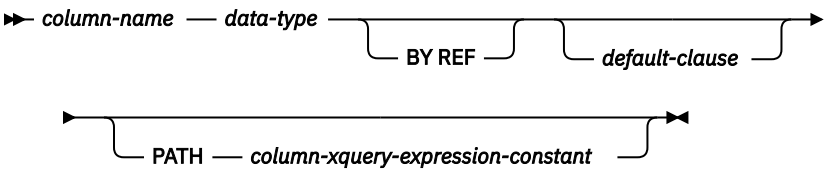
The XMLTABLE function returns a result table from the evaluation of XQuery expressions, possibly using specified input arguments as XQuery variables. Each sequence item in the result sequence of the row XQuery expression represents a row of the result table.



row-xquery-argument



xml-table-regular-column-definition



xml-table-ordinality-column-definition

Notes:

[1] The xml-table-ordinality-column-definition clause must not be specified more than once (SQLSTATE 42614).

[2] The data type of the expression cannot be DECFLOAT.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

xmlnamespaces-declaration
Specifies one or more XML namespace declarations that become part of the static context of the *row-xquery-expression-constant* and the *column-xquery-expression-constant*. The set of statically known namespaces for XQuery expressions which are arguments of XMLTABLE is the combination of the pre-established set of statically known namespaces and the namespace declarations specified in this clause. The XQuery prolog within an XQuery expression may override these namespaces.

If *xmlnamespaces-declaration* is not specified, only the pre-established set of statically known namespaces apply to the XQuery expressions.

row-xquery-expression-constant
Specifies an SQL character string constant that is interpreted as an XQuery expression using supported XQuery language syntax. The constant string is converted directly to UTF-8 without conversion to the database or section code page. The XQuery expression executes using an optional set of input XML values, and returns an output XQuery sequence where a row is generated for each item in the sequence. The value for *row-xquery-expression-constant* must not be an empty string or a string of all blanks (SQLSTATE 10505).

PASSING
Specifies input values and the manner in which these values are passed to the XQuery expression specified by *row-xquery-expression-constant*. By default, every unique column name that is in the scope where the function is invoked is implicitly passed to the XQuery expression using the name of the column as the variable name. If an *identifier* in a specified row-xquery-argument matches an in-scope column name, then the explicit row-xquery-argument is passed to the XQuery expression overriding that implicit column.

BY REF
Specifies that any XML input arguments are, by default, passed by reference. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any, directly from the specified input expressions, preserving all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons involving some nodes contained between the two input arguments might refer to nodes within the same XML node tree.

This clause has no impact on how non-XML values are passed. The non-XML values create a new copy of the value during the cast to XML.

row-xquery-argument
Specifies an argument that is to be passed to the XQuery expression that is specified by *row-xquery-expression-constant*. The method through which *row-xquery-argument* is used in the XQuery expression depends on whether the argument is specified as an *xquery-context-item-expression* or an *xquery-variable-expression*.

- If the resulting value is of type XML, it becomes an *input-xml-value*. A null XML value is converted to an XML empty sequence.

- If the resulting value is not of type XML, it must be castable to the XML data type. A null value is converted to an XML empty sequence. The converted value becomes an *input-xml-value*.

When the *row-xquery-expression-constant* is evaluated, an XQuery variable is presented with a value equal to *input-xml-value* and a name specified by the AS clause.

xquery-context-item-expression
An expression that returns a value that is XML, integer, decimal, or a character or graphic string that is not a LOB. *xquery-context-item-expression* must not be a character string that is bit data.

xquery-context-item-expression specifies the initial context item for the *row-xquery-expression*. The value of the initial context item is the result of *xquery-context-item-expression* cast to XML. *xquery-context-item-expression* must not be specified more than one time.

xquery-variable-expression
Specifies an SQL expression whose value is available to the XQuery expression specified by *row-xquery-expression-constant* during execution. The expression cannot contain a NEXT VALUE expression, PREVIOUS VALUE expression (SQLSTATE 428F9), or an OLAP function (SQLSTATE 42903). The data type of the expression cannot be DECFLOAT.

AS *identifier*
Specifies that the value generated by *xquery-variable-expression* will be passed to *row-xquery-expression-constant* as an XQuery variable. The variable name will be *identifier*. The leading dollar sign ($) that precedes variable names in the XQuery language is not included in *identifier*. The identifier must be a valid XQuery variable name and is restricted to an XML NCName. The identifier must not be greater than 128 bytes in length. Two arguments within the same PASSING clause cannot use the same identifier (SQLSTATE 42711).

BY REF
Indicates that an XML input value is to be passed by reference. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any, directly from the specified input expressions, preserving all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons involving some nodes contained between the two input arguments might refer to nodes within the same XML node tree. If BY REF is not specified following an *xquery-expression-variable*, XML arguments are passed by way of the default passing mechanism that is provided through the syntax that follows the PASSING keyword. This option cannot be specified for non-XML values (SQLSTATE 42636). When a non-XML value is passed, the value is converted to XML; this process creates a copy.

COLUMNS
Specifies the output columns of the result table. If this clause is not specified, a single unnamed column of data type XML is returned by reference, with the value based on the sequence item from evaluating the XQuery expression in the *row-xquery-expression-constant* (equivalent to specifying PATH '.'). To reference the result column, a *column-name* must be specified in the *correlation-clause* following the function.

xml-table-regular-column-definition
Specifies the output columns of the result table including the column name, data type, XML passing mechanism and an XQuery expression to extract the value from the sequence item for the row

column-name
Specifies the name of the column in the result table. The name cannot be qualified and the same name cannot be used for more than one column of the table (SQLSTATE 42711).

data-type
Specifies the data type of the column. See CREATE TABLE for the syntax and a description of types available. A *data-type* may be used in XMLTable if there is a supported XMLCAST from the XML data type to the specified *data-type*.

BY REF
Specifies that XML values are returned by reference for columns of data type XML. By default, XML values are returned BY REF. When XML values are returned by reference, the XML value includes the input node trees, if any, directly from the result values, and preserves all properties, including the original node identities and document order. This option cannot be specified for non-XML columns (SQLSTATE 42636). When a non-XML column is processed, the value is converted from XML; this process creates a copy.

default-clause
Specifies a default value for the column. See CREATE TABLE for the syntax and a description of the *default-clause*. For XMLTABLE result columns, the default is applied when the processing the XQuery expression contained in *column-xquery-expression-constant* returns an empty sequence.

PATH *column-xquery-expression-constant*
Specifies an SQL character string constant that is interpreted as an XQuery expression using supported XQuery language syntax. The constant string is converted directly to UTF-8 without conversion to the database or section code page. The *column-xquery-expression-constant* specifies an XQuery expression that determines the column value with respect to an item that is the result of evaluating the XQuery expression in *row-xquery-expression-constant*. Given an item from the result of processing the *row-xquery-expression-constant* as the externally provided context item, the *column-xquery-expression-constant* is evaluated, returning an output sequence. The column value is determined based on this output sequence as follows.

- If the output sequence contains zero items, the *default-clause* provides the value of the column.
- If an empty sequence is returned and no *default-clause* was specified, a null value is assigned to the column.
- If a non-empty sequence is returned, the value is XMLCAST to the *data-type* specified for the column. An error could be returned from processing this XMLCAST.

The value for *column-xquery-expression-constant* must not be an empty string or a string of all blanks (SQLSTATE 10505). If this clause is not specified, the default XQuery expression is simply the *column-name*.

xml-table-ordinality-column-definition
Specifies the ordinality column of the result table.

column-name
Specifies the name of the column in the result table. The name cannot be qualified and the same name cannot be used for more than one column of the table (SQLSTATE 42711).

FOR ORDINALITY
Specifies that *column-name* is the ordinality column of the result table. The data type of this column is BIGINT. The value of this column in the result table is the sequential number of the item for the row in the resulting sequence from evaluating the XQuery expression in *row-xquery-expression-constant*.

If the evaluation of any of the XQuery expressions results in an error, then the XMLTABLE function returns the XQuery error (SQLSTATE class '10').

Example

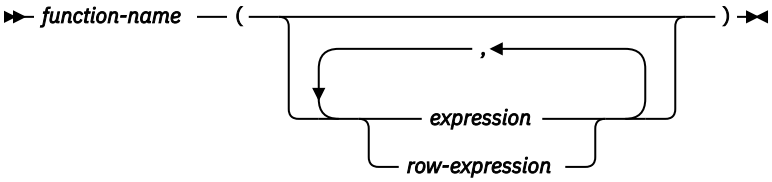
List as a table result the purchase order items for orders with a status of 'NEW'.

```
SELECT U."PO ID", U."Part #", U."Product Name",
     U."Quantity", U."Price", U."Order Date"
  FROM PURCHASEORDER P,
     XMLTABLE('$po/PurchaseOrder/item' PASSING P.PORDER AS "po"
              COLUMNS "PO ID"         INTEGER       PATH '../@PoNum',
                      "Part #"        CHAR(10)      PATH 'partid',
                      "Product Name"  VARCHAR(50)   PATH 'name',
                      "Quantity"      INTEGER       PATH 'quantity',
                      "Price"         DECIMAL(9,2)  PATH 'price',
```

```
                        "Order Date"    DATE          PATH '../@OrderDate'
                ) AS U
      WHERE P.STATUS = 'Unshipped'
```

User-defined functions

User-defined functions (UDFs) are extensions or additions to the existing built-in functions of the SQL language.



A user-defined function can be a scalar function, which returns a single value each time it is called; an aggregate function, which is passed a set of like values and returns a single value for the set; a row function, which returns one row; or a table function, which returns a table.

A number of user-defined functions are provided in the SYSFUN and SYSPROC schemas.

A UDF can be an aggregate function only if it is sourced on an existing aggregate function. A UDF is referenced by means of a qualified or unqualified function name, followed by parentheses enclosing the function arguments (if any). A user-defined column or scalar function registered with the database can be referenced in the same contexts in which any built-in function can appear. A user-defined row function can be referenced only implicitly when registered as a transform function for a user-defined type. A user-defined table function registered with the database can be referenced only in the FROM clause of a SELECT statement.

Function arguments must correspond in number and position to the parameters specified for the user-defined function when it was registered with the database. In addition, the arguments must be of data types that are promotable to the data types of the corresponding defined parameters.

The result of the function is specified in the RETURNS clause. The RETURNS clause, defined when the UDF was registered, determines whether or not a function is a table function. If the RETURNS NULL ON NULL INPUT clause is specified (or defaulted to) when the function is registered, the result is null if any argument is null. In the case of table functions, this is interpreted to mean a return table with no rows (that is, an empty table).

See "Row expressions" for more information about rules and row data types.

Following are some examples of user-defined functions:

- A scalar UDF called ADDRESS extracts the home address from resumes stored in script format. The ADDRESS function expects a CLOB argument and returns a VARCHAR(4000) value:

  ```
  SELECT EMPNO, ADDRESS(RESUME) FROM EMP_RESUME
    WHERE RESUME_FORMAT = 'SCRIPT'
  ```

- Table T2 has a numeric column A. Invoking the scalar UDF called ADDRESS from the previous example:

  ```
  SELECT ADDRESS(A) FROM T2
  ```

  raises an error (SQLSTATE 42884), because no function with a matching name and with a parameter that is promotable from the argument exists.

- A table UDF called WHO returns information about the sessions on the server machine that were active at the time that the statement is executed. The WHO function is invoked from within a FROM clause that includes the keyword TABLE and a mandatory correlation variable. The column names of the WHO() table were defined in the CREATE FUNCTION statement.

  ```
  SELECT ID, START_DATE, ORIG_MACHINE
    FROM TABLE( WHO() ) AS QQ
    WHERE START_DATE LIKE 'MAY%'
  ```

Built-in procedures

A procedure is an application program that can be started through the SQL CALL statement. The procedure is specified by a procedure name, which may be followed by arguments that are enclosed within parentheses. *Built-in procedures* are procedures provided with the database manager.

This topic lists the supported built-in procedures for managing XML schema repository (XSR) objects. See Table 108 on page 631.

There are additional built-in procedures documented under the following headings:

- ADMIN_CMD procedure and associated SQL routines
- Administrative task scheduler routines and views
- Audit routines and procedures
- Automatic maintenance SQL routines and views
- Common SQL API stored procedures
- Explain routines
- Monitor routines
- Snapshot SQL routines and views
- SQL procedures SQL routines
- Stepwise redistribute SQL routines
- Storage management tool SQL routines
- Text search SQL routines
- Workload management routines
- Miscellaneous SQL routines and views

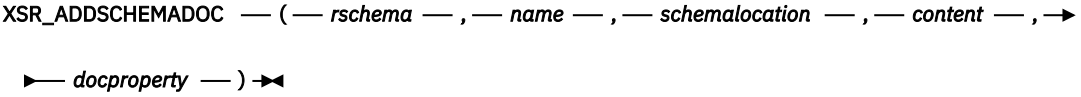
For details about these additional built-in procedures, see "Supported built-in SQL routines and views" in *Administrative Routines and Views* .

Table 108. XML schema repository procedures

Function	Description
“XSR_ADDSCHEMADOC” on page 631	Add an XML schema document to an XML schema.
“XSR_COMPLETE” on page 632	Complete the XML schema registration process for an XML schema.
“XSR_DTD” on page 633	Register a document type declaration.
“XSR_EXTENTITY” on page 634	Register an external entity.
“XSR_REGISTER” on page 635	Register an XML schema.
“XSR_UPDATE” on page 637	Update an existing XML schema.

XSR_ADDSCHEMADOC

Each XML schema in the XML schema repository (XSR) can consist of one or more XML schema documents. Where an XML schema consists of multiple documents, the XSR_ADDSCHEMADOC procedure is used to add every XML schema other than the primary XML schema document.



The schema is SYSPROC.

Authorization

The authorization ID of the caller of the procedure must be the owner of the XSR object as recorded in the catalog view SYSCAT.XSROBJECTS.

rschema
An input argument of type VARCHAR (128) that specifies the SQL schema for the XML schema. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR, which is to be moved to the complete state. (The other part of the SQL identifier is supplied by the name argument.) This argument can have a null value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

name
An input argument of type VARCHAR (128) that specifies the name of the XML schema. The complete SQL identifier for the XML schema is *rschema.name*. The XML schema name must already exist as a result of calling the XSR_REGISTER procedure, and XML schema registration cannot yet be completed. This argument cannot have a null value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

schemalocation
An input argument of type VARCHAR (1000), which can have a null value, that indicates the schema location of the primary XML schema document to which the XML schema document is being added. This argument is the external name of the XML schema, that is, the primary document can be identified in the XML instance documents with the xsi:schemaLocation attribute.

content
An input parameter of type BLOB (30M) that contains the content of the XML schema document being added. This argument cannot have a null value; an XML schema document must be supplied.

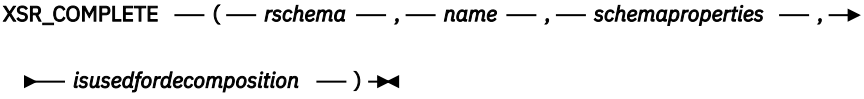
docproperty
An input parameter of type BLOB (5M) that indicates the properties for the XML schema document being added. This parameter can have a null value; otherwise, the value is an XML document.

Example

```
CALL SYSPROC.XSR_ADDSCHEMADOC(
  'user1',
  'POschema',
  'http://myPOschema/address.xsd',
  :content_host_var,
  0)
```

XSR_COMPLETE

The XSR_COMPLETE procedure is the final procedure to be called as part of the XML schema registration process, which registers XML schemas with the XML schema repository (XSR). An XML schema is not available for validation until the schema registration completes through a call to this procedure.



The schema is SYSPROC.

Authorization:

The authorization ID of the caller of the procedure must be the owner of the XSR object as recorded in the catalog view SYSCAT.XSROBJECTS.

rschema
An input argument of type VARCHAR (128) that specifies the SQL schema for the XML schema. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR, which is to be moved to the complete state. (The other part of the SQL identifier is supplied by the name argument.) This argument can have a null value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

name
An input argument of type VARCHAR (128) that specifies the name of the XML schema. The complete SQL identifier for the XML schema, for which a completion check is to be performed, is *rschema.name*. The XML schema name must already exist as a result of calling the XSR_REGISTER procedure, and XML schema registration cannot yet be completed. This argument cannot have a null value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

schemaproperties
An input argument of type BLOB (5M) that specifies properties, if any, associated with the XML schema. The value for this argument is either the null value, if there are no associated properties, or an XML document representing the properties for the XML schema.

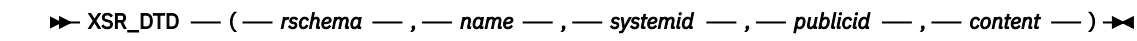
isusedfordecomposition
An input parameter of type integer that indicates if an XML schema is to be used for decomposition. If an XML schema is to be used for decomposition, this value should be set to 1; otherwise, it should be set to zero.

Example

```
CALL SYSPROC.XSR_COMPLETE(
  'user1',
  'POschema',
  :schemaproperty_host_var,
  0)
```

XSR_DTD

The XSR_DTD procedure registers a document type declaration (DTD) with the XML schema repository (XSR).



The schema is SYSPROC.

Authorization

The authorization ID of the caller of the procedure must have at least one of the following:

- DBADM authority.
- IMPLICIT_SCHEMA database authority if the SQL schema does not exist.
- CREATEIN privilege if the SQL schema exists.

rschema
An input and output argument of type VARCHAR (128) that specifies the SQL schema for the DTD. The SQL schema is one part of the SQL identifier used to identify this DTD in the XSR. (The other part of the SQL identifier is supplied by the *name* argument.) This argument can have a null value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. Relational schemas that begin with the string 'SYS' must not be used for this value. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

name
An input and output argument of type VARCHAR (128) that specifies the name of the DTD. The complete SQL identifier for the DTD is *rschema.name* and should be unique among all objects in the XSR. This argument accepts a null value. When a null value is provided for this argument, a unique value is generated and stored within the XSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

systemid
An input parameter of type VARCHAR (1000) that specifies the system identifier of the DTD. The system ID of the DTD should match the uniform resource identifier of the DTD in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration (as prefixed by the SYSTEM keyword, if used). This argument cannot have a null value. The system ID can be specified together with a public ID.

publicid
An input parameter of type VARCHAR (1000) that specifies the public identifier of the DTD. The public ID of a DTD should match the uniform resource identifier of the DTD in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration (as prefixed by the PUBLIC keyword, if used). This argument accepts a null value and should be used only if also specified in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration.

content
An input parameter of type BLOB (30M) that contains the content of the DTD document. This argument cannot have a null value.

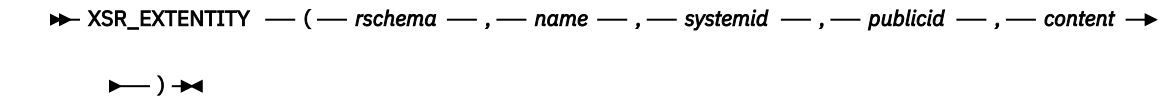
Example

Register the DTD identified by the system ID *http://www.test.com/person.dtd* and public ID *http://www.test.com/person*:

```
CALL SYSPROC.XSR_DTD ( 'MYDEPT' ,
   'PERSONDTD' ,
   'http://www.test.com/person.dtd' ,
   'http://www.test.com/person',
   :content_host_variable
   )
```

XSR_EXTENTITY

The XSR_EXTENTITY procedure registers an external entity with the XML schema repository (XSR).



The schema is SYSPROC.

Authorization

The authorization ID of the caller of the procedure must have at least one of the following:

- DBADM authority.
- IMPLICIT_SCHEMA database authority if the SQL schema does not exist.
- CREATEIN privilege if the SQL schema exists.

rschema
An input and output argument of type VARCHAR (128) that specifies the SQL schema for the external entity. The SQL schema is one part of the SQL identifier used to identify this external entity in the XSR. (The other part of the SQL identifier is supplied by the *name* argument.) This argument can have a null value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also

apply to this argument. Relational schemas that begin with the string 'SYS' must not be used for this value. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

name
An input and output argument of type VARCHAR (128) that specifies the name of the external entity. The complete SQL identifier for the external entity is *rschema.name* and should be unique among all objects in the XSR. This argument accepts a null value. When a null value is provided for this argument, a unique value is generated and stored within the XSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

systemid
An input parameter of type VARCHAR (1000) that specifies the system identifier of the external entity. The system ID of the external entity should match the uniform resource identifier of the external entity in the ENTITY declaration (as prefixed by the SYSTEM keyword, if used). This argument cannot have a null value. The system ID can be specified together with a public ID.

publicid
An input parameter of type VARCHAR (1000) that specifies the public identifier of the external entity. The public ID of a external entity should match the uniform resource identifier of the external entity in the ENTITY declaration (as prefixed by the PUBLIC keyword, if used). This argument accepts a null value and should be used only if also specified in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration.

content
An input parameter of type BLOB (30M) that contains the content of the external entity document. This argument cannot have a null value.

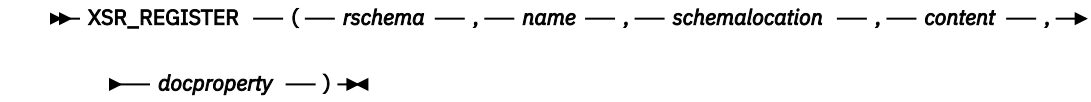
Example

Register the external entities identified by the system identifiers *http://www.test.com/food/chocolate.txt* and *http://www.test.com/food/cookie.txt*:

```
CALL SYSPROC.XSR_EXTENTITY ( 'FOOD' ,
   'CHOCLATE' ,
   'http://www.test.com/food/chocolate.txt' ,
   NULL ,
   :content_of_chocolate.txt_as_a_host_variable
   )

CALL SYSPROC.XSR_EXTENTITY ( 'FOOD' ,
   'COOKIE' ,
   'http://www.test.com/food/cookie.txt' ,
   NULL ,
   :content_of_cookie.txt_as_a_host_variable
   )
```

XSR_REGISTER

The XSR_REGISTER procedure is the first procedure to be called as part of the XML schema registration process, which registers XML schemas with the XML schema repository (XSR).



The schema is SYSPROC.

Authorization

The authorization ID of the caller of the procedure must have at least one of the following authorities:

- DBADM authority.

- IMPLICIT_SCHEMA database authority if the SQL schema does not exist.
- CREATEIN privilege if the SQL schema exists.

rschema
An input and output argument of type VARCHAR (128) that specifies the SQL schema for the XML schema. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR. (The other part of the SQL identifier is supplied by the name argument.) This argument can have a null value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. Relational schemas that begin with the string 'SYS' must not be used for this value. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

name
An input and output argument of type VARCHAR (128) that specifies the name of the XML schema. The complete SQL identifier for the XML schema is *rschema.name* and should be unique among all objects in the XSR. This argument accepts a null value. When a null value is provided for this argument, a unique value is generated and stored within the XSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

schemalocation
An input argument of type VARCHAR (1000), which can have a null value, that indicates the schema location of the primary XML schema document. This argument is the external name of the XML schema, that is, the primary document can be identified in the XML instance documents with the xsi:schemaLocation attribute.

content
An input parameter of type BLOB (30M) that contains the content of the primary XML schema document. This argument cannot have a null value; an XML schema document must be supplied.

docproperty
An input parameter of type BLOB (5M) that indicates the properties for the primary XML schema document. This parameter can have a null value; otherwise, the value is an XML document.

Examples

- *Example 1*: The following example shows how to call the XSR_REGISTER procedure from the command line:

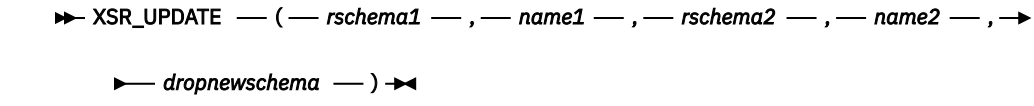
```
CALL SYSPROC.XSR_REGISTER(
  'user1',
  'POschema',
  'http://myPOschema/PO.xsd',
  :content_host_var,
  :docproperty_host_var)
```

- *Example 2*: The following example shows how to call the XSR_REGISTER procedure from a Java application program:

```
stmt = con.prepareCall("CALL SYSPROC.XSR_REGISTER (?, ?, ?, ?, ?)");
String xsrObjectName = "myschema1";
String xmlSchemaLocation = "po.xsd";
stmt.setNull(1, java.sql.Types.VARCHAR);
stmt.setString(2, xsrObjectName);
stmt.setString(3, xmlSchemaLocation);
stmt.setBinaryStream(4, buffer, (int)length);
stmt.setNull(5, java.sql.Types.BLOB);
stmt.registerOutParameter(1, java.sql.Types.VARCHAR);
stmt.registerOutParameter(2, java.sql.Types.VARCHAR);
stmt.execute();
```

XSR_UPDATE

The XSR_UPDATE procedure is used to evolve an existing XML schema in the XML schema repository (XSR). This enables you to modify or extend an existing XML schema so that it can be used to validate both already existing and newly inserted XML documents.



The schema is SYSPROC.

The original XML schema and the new XML schema specified as arguments to XSR_UPDATE must both be registered and completed in the XSR before the procedure is called. These XML schemas must also be compatible. For details about the compatibility requirements see *Compatibility requirements for evolving an XML schema*.

Authorization

The privileges held by the authorization ID of the caller of the procedure must include at least one of the following:

- DBADM authority.
- SELECT privilege on the catalog views SYSCAT.XSROBJECTS and SYSCAT.XSROBJECTCOMPONENTS and one of the following sets of privileges:
  - OWNER of the XML schema specified by the SQL schema *rschema1* and the object name *name1*
  - ALTERIN privilege on the SQL schema specified by the *rschema1* argument and, if the *dropnewschema* argument is not equal to zero, DROPIN privilege on the SQL schema specified by the *rschema2* argument.

rschema1
An input argument of type VARCHAR (128) that specifies the SQL schema for the original XML schema to be updated. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR. (The other part of the SQL identifier is supplied by the *name1* argument.) This argument cannot have a null value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

name1
An input argument of type VARCHAR (128) that specifies the name of the original XML schema to be updated. The complete SQL identifier for the XML schema is *rschema1.name1* . This XML schema must already be registered and completed in the XSR. This argument cannot have a null value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

rschema2
An input argument of type VARCHAR (128) that specifies the SQL schema for the new XML schema that will be used to update the original XML schema. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR. (The other part of the SQL identifier is supplied by the *name2* argument.) This argument cannot have a null value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

name2
An input argument of type VARCHAR (128) that specifies the name of the new XML schema that will be used to update the original XML schema. The complete SQL identifier for the XML schema is *rschema2.name2*. This XML schema must already be registered and completed in the XSR. This argument cannot have a null value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

dropnewschema
An input parameter of type integer that indicates whether the new XML schema should be dropped after it is used to update the original XML schema. Setting this parameter to any nonzero value will cause the new XML schema to be dropped. This argument cannot have a null value.

Example

```
CALL SYSPROC.XSR_UPDATE(
  'STORE',
  'PROD',
  'STORE',
  'NEWPROD',
  1)
```

The contents of the XML schema STORE.PROD is updated with the contents of STORE.NEWPROD, and the XML schema STORE.NEWPROD is dropped.

SQL queries

A *query* specifies a result table. A query is a component of certain SQL statements.

The three forms of a query are:

- subselect
- fullselect
- select-statement.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- For each table or view identified in the query, one of the following authorities:
  - SELECT privilege on the table or view
  - SELECTIN privilege on the schema of the table or view
  - DATAACCESS privilege on the schema of the table or view
  - CONTROL privilege on the table or view
- DATAACCESS authority

For each global variable used as an expression in the query, the privileges held by the authorization ID of the statement must include one of the following authorities:

- READ privilege on the global variable that is not defined in a module
- EXECUTE privilege on the module of the global variable that is defined in a module
- EXECUTEIN privilege on the schema containing the module of the global variable that is defined in a module
- DATAACCESS authority on the schema containing the module of the global variable that is defined in a module

If the query contains an SQL data change statement, the authorization requirements of that statement also apply to the query.

Group privileges, with the exception of PUBLIC, are not checked for queries that are contained in static SQL statements or DDL statements.

For nicknames, authorization requirements of the data source for the object referenced by the nickname are applied when the query is processed. The authorization ID of the statement may be mapped to a different authorization ID at the data source.

Queries and table expressions

A *query* is a component of certain SQL statements; it specifies a (temporary) result table.

A *table expression* creates a temporary result table from a simple query. Clauses further refine the result table. For example, you can use a table expression as a query to select all of the managers from several

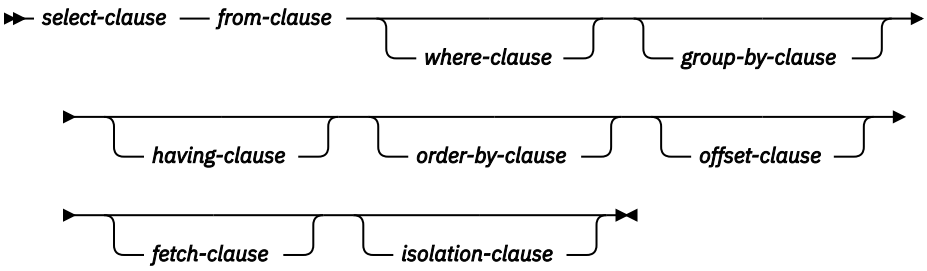
departments, specify that they must have over 15 years of working experience, and be located at the New York branch office.

A *common table expression* is like a temporary view within a complex query. It can be referenced in other places within the query, and can be used in place of a view. Each use of a specific common table expression within a complex query shares the same temporary view.

Recursive use of a common table expression within a query can be used to support applications such as airline reservation systems, bill of materials (BOM) generators, and network planning.

subselect

The *subselect* is a component of the fullselect.



A subselect specifies a result table derived from the tables, views or nicknames identified in the FROM clause. The derivation can be described as a sequence of operations in which the result of each operation is input for the next. (This is only a way of describing the subselect. The method used to perform the derivation can be quite different from this description. If portions of the subselect do not actually need to be executed for the correct result to be obtained, they might or might not be executed.)

The authorization for a *subselect* is described in the Authorization section in "SQL queries".

The clauses of the subselect are processed in the following sequence:

1. FROM clause
2. WHERE clause
3. GROUP BY clause
4. HAVING clause
5. SELECT clause
6. ORDER BY clause
7. OFFSET clause
8. FETCH clause

A subselect that contains an ORDER BY clause, OFFSET clause, or FETCH clause cannot be specified:

- In the outermost fullselect of a view.
- In the outer fullselect of a materialized query table.
- Unless the subselect is enclosed in parenthesis.

For example, the following is not valid (SQLSTATE 428FJ):

```
SELECT * FROM T1
   ORDER BY C1
UNION
SELECT * FROM T2
   ORDER BY C1
```

The following example *is* valid:

```
(SELECT * FROM T1
   ORDER BY C1)
UNION
```

```
(SELECT * FROM T2
   ORDER BY C1)
```

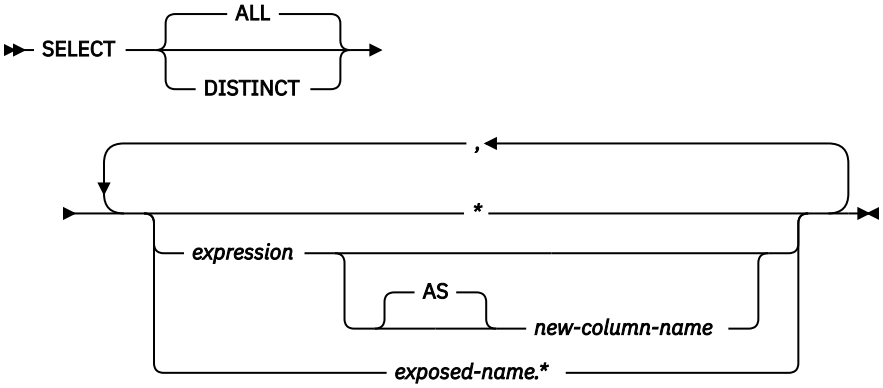
Note: An ORDER BY clause in a subselect does not affect the order of the rows returned by a query. An ORDER BY clause only affects the order of the rows returned if it is specified in the outermost fullselect.

For details about the clauses in the subselect query, refer to the following topics:

- “select-clause” on page 640
- “from-clause” on page 643
- “where-clause” on page 690
- “group-by-clause” on page 691
- “having-clause” on page 702
- “order-by-clause” on page 703
- “offset-clause” on page 706
- “fetch-clause” on page 705
- “isolation-clause (subselect query)” on page 707

select-clause

The SELECT clause specifies the columns of the final result table.



The column values are produced by the application of the *select list* to the final result table, *R*. The select list is the names or expressions specified in the SELECT clause, and *R* is the result of the previous operation of the subselect. For example, if the only clauses specified are SELECT, FROM, and WHERE, *R* is the result of that WHERE clause.

ALL
Retains all rows of the final result table, and does not eliminate redundant duplicates. This is the default.

DISTINCT
Eliminates all but one of each set of duplicate rows of the final result table. If DISTINCT is used, no string column of the result table can be a LOB type, distinct type based on LOB, or structured type. DISTINCT can be used more than once in a subselect. This includes SELECT DISTINCT, the use of DISTINCT in an aggregate function of the select list or HAVING clause, and subqueries of the subselect.

Two rows are duplicates of one another only if each value in the first is equal to the corresponding value in the second. For determining duplicates, two null values are considered equal, and two different decimal floating-point representations of the same number are considered equal. For example, -0 is equal to +0 and 2.0 is equal to 2.00. Each of the decimal floating-point special values are also considered equal: -NAN equals -NAN, -SNAN equals -SNAN, -INFINITY equals -INFINITY, INFINITY equals INFINITY, SNAN equals SNAN, and NAN equals NAN.

When the data type of a column is decimal floating-point, and multiple representations of the same number exist in the column, the particular value that is returned for a SELECT DISTINCT can be any one of the representations in the column. For more information, see "Numeric comparisons" on page 65.

For compatibility with other SQL implementations, UNIQUE can be specified as a synonym for DISTINCT.

Select list notation

***

Represents a list of names that identify the columns of table *R*, excluding any columns defined as IMPLICITLY HIDDEN. The first name in the list identifies the first column of *R*, the second name identifies the second column of *R*, and so on.

The list of names is established when the program containing the SELECT clause is bound. Hence the asterisk (*) does not identify any columns that have been added to a table after the statement containing the table reference has been bound.

expression

Specifies the values of a result column. Can be any expression that is a valid SQL language element, but commonly includes column names. Each column name used in the select list must unambiguously identify a column of R. The result type of the expression cannot be a row type (SQLSTATE 428H2).

new-column-name* or AS *new-column-name

Names or renames the result column. The name must not be qualified and does not have to be unique. Subsequent usage of column-name is limited as follows:

- A *new-column-name* specified in the AS clause can be used in the order-by-clause, provided the name is unique.
- A *new-column-name* specified in the AS clause of the select list cannot be used in any other clause within the subselect (where-clause, group-by-clause or having-clause).
- A *new-column-name* specified in the AS clause cannot be used in the update-clause.
- A *new-column-name* specified in the AS clause is known outside the fullselect of nested table expressions, common table expressions and CREATE VIEW.

exposed-name.*

Represents the list of names that identify the columns of the result table identified by *exposed-name*, excluding any columns defined as IMPLICITLY HIDDEN. The *exposed-name* can be a table name, view name, nickname, or correlation name, and must designate a table, view or nickname named in the FROM clause. The first name in the list identifies the first column of the table, view or nickname, the second name in the list identifies the second column of the table, view or nickname, and so on.

The list of names is established when the statement containing the SELECT clause is bound. Therefore, * does not identify any columns that have been added to a table after the statement has been bound.

The number of columns in the result of SELECT is the same as the number of expressions in the operational form of the select list (that is, the list established when the statement is prepared), and cannot exceed 500 for a 4K page size or 1012 for an 8K, 16K, or 32K page size.

Limitations on string columns

For restrictions using varying-length character strings on the select list, see "Character strings" on page 31.

Applying the select list

Some of the results of applying the select list to R depend on whether GROUP BY or HAVING is used. The results are described in two separate lists.

If GROUP BY or HAVING is used

- An expression *X* (not an aggregate function) used in the select list must have a GROUP BY clause with:
  - a *grouping-expression* in which each expression or column-name unambiguously identifies a column of R (see "group-by-clause" on page 691) or
  - each column of R referenced in *X* as a separate *grouping-expression*.
- The select list is applied to each group of R, and the result contains as many rows as there are groups in R. When the select list is applied to a group of R, that group is the source of the arguments of the aggregate functions in the select list.

If neither GROUP BY nor HAVING is used

- Either the select list must not include any aggregate functions, or each *column-name* in the select list must be specified within an aggregate function or must be a correlated column reference.
- If the select does not include aggregate functions, then the select list is applied to each row of R and the result contains as many rows as there are rows in R.
- If the select list is a list of aggregate functions, then R is the source of the arguments of the functions and the result of applying the select list is one row.

In either case the *n*th column of the result contains the values specified by applying the *n*th expression in the operational form of the select list.

Null attributes of result columns

Result columns do not allow null values if they are derived from:

- A column that does not allow null values
- A constant
- The COUNT or COUNT_BIG function
- A host variable that does not have an indicator variable
- A scalar function or expression that does not include an operand where nulls are allowed
- An expression with a result type that is a weakly typed distinct type defined with a NOT NULL data type constraint

Result columns allow null values if they are derived from:

- Any aggregate function except COUNT or COUNT_BIG
- A column where null values are allowed
- A scalar function or expression that includes an operand where nulls are allowed
- A NULLIF function with arguments containing equal values
- A host variable that has an indicator variable, an SQL parameter, an SQL variable, or a global variable
- A result of a set operation if at least one of the corresponding items in the select list is nullable
- An arithmetic expression or view column that is derived from an arithmetic expression and the database is configured with **dft_sqlmathwarn** set to Yes
- A scalar subselect
- A dereference operation
- A GROUPING SETS *grouping-expression*

Names of result columns

- If the AS clause is specified, the name of the result column is the name specified on the AS clause.
- If the AS clause is not specified and if a column list is specified in the correlation clause, the name of the result column is the corresponding name in the correlation column list.

- If neither an AS clause nor a column list in the correlation clause is specified and if the result column is derived only from a single column (without any functions or operators), the name of the result column is the unqualified name of that column.
- If neither an AS clause nor a column list in the correlation clause is specified and if the result column is derived only from a single SQL variable or SQL parameter (without any functions or operators), the name of the result column is the unqualified name of that SQL variable or SQL parameter.
- If neither an AS clause nor a column list in the correlation clause is specified and if the result column is derived using a dereference operation, the name of the result column is the unqualified name of the target column of the dereference operation.
- All other result columns are unnamed. The system assigns temporary numbers (as character strings) to these columns.

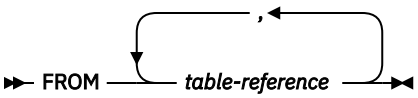
Data types of result columns

Each column of the result of SELECT acquires a data type from the expression from which it is derived.

When the expression is ...	The data type of the result column is ...
the name of any numeric column	the same as the data type of the column, with the same precision and scale for DECIMAL columns, or the same precision for DECFLOAT columns.
a constant	the same as the data type of the constant.
the name of any numeric variable	the same as the data type of the variable, with the same precision and scale for DECIMAL variables, or the same precision for DECFLOAT variables.
the name of any string column	the same as the data type of the column, with the same length attribute.
the name of any string variable	the same as the data type of the variable, with the same length attribute; if the data type of the variable is not identical to an SQL data type (for example, a NUL-terminated string in C), the result column is a varying-length string.
the name of a datetime column	the same as the data type of the column.
the name of a user-defined type column	the same as the data type of the column.
the name of a reference type column	the same as the data type of the column.

from-clause

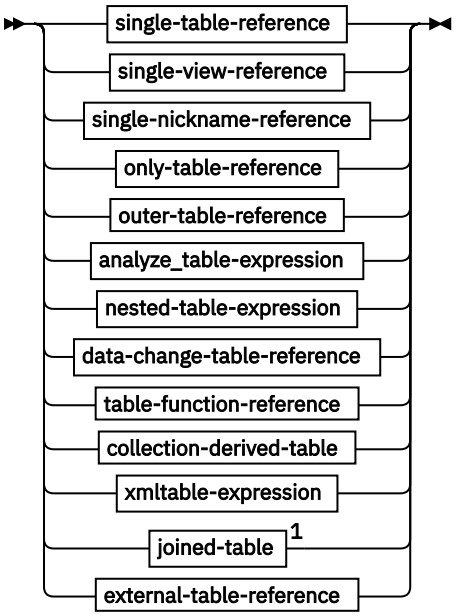
The FROM clause specifies an intermediate result table.



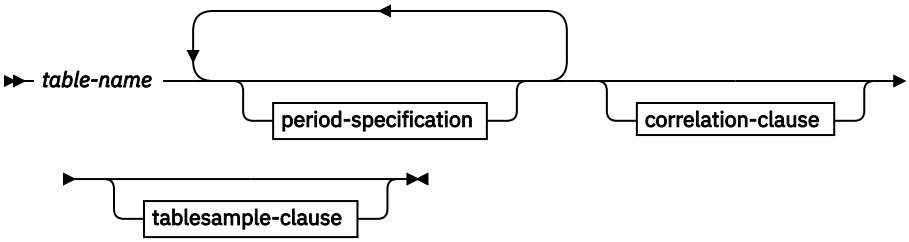
If only one *table-reference* is specified, the intermediate result table is the result of that *table-reference*. If more than one *table-reference* is specified, the intermediate result table consists of all possible combinations of the rows of the specified *table-reference* (the Cartesian product). Each row of the result is a row from the first *table-reference* concatenated with a row from the second *table-reference*, concatenated in turn with a row from the third, and so on. The number of rows in the result is the product of the number of rows in all the individual table references. For a description of *table-reference*, see "table-reference" on page 644.

table-reference

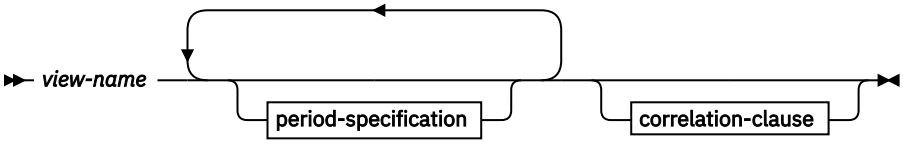
A *table-reference* specifies an intermediate result table.



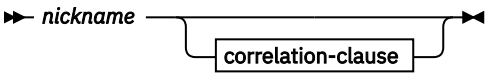
single-table-reference



single-view-reference



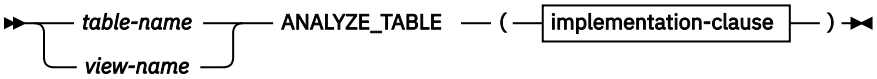
single-nickname-reference



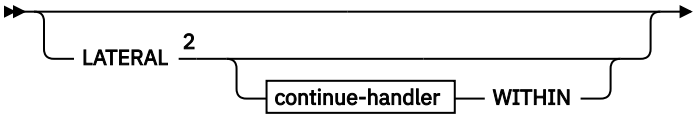
only-table-reference

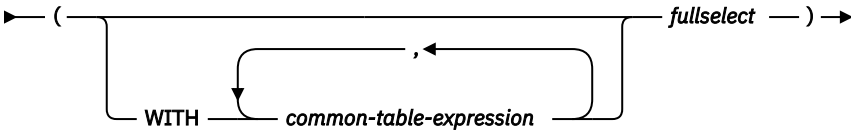
outer-table-reference

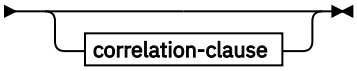
analyze_table-expression



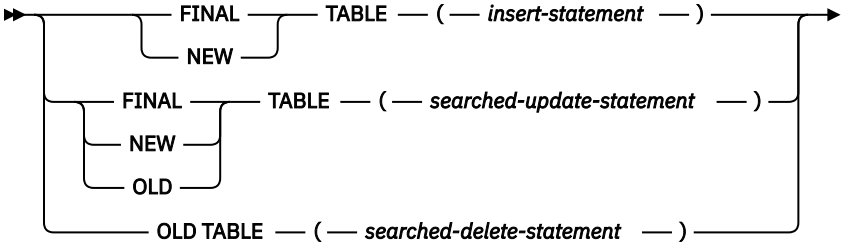
nested-table-expression







data-change-table-reference



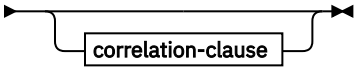
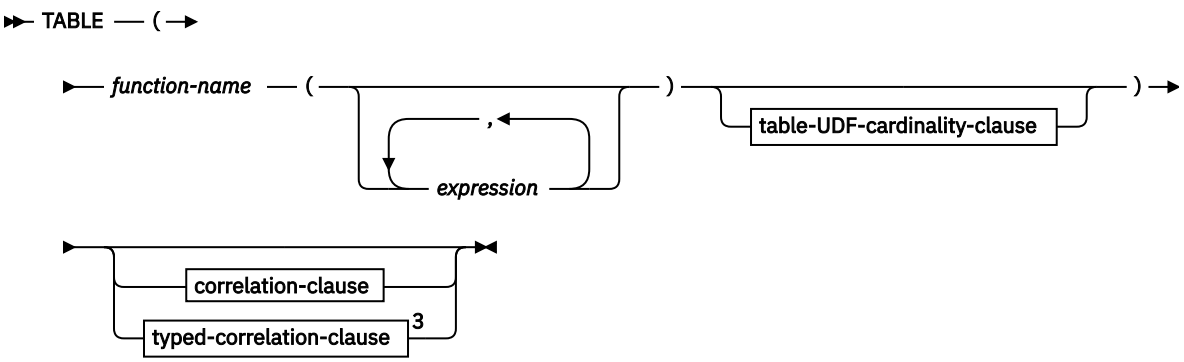
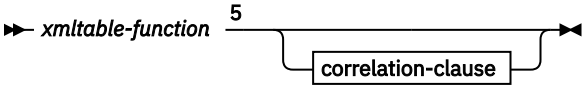


table-function-reference

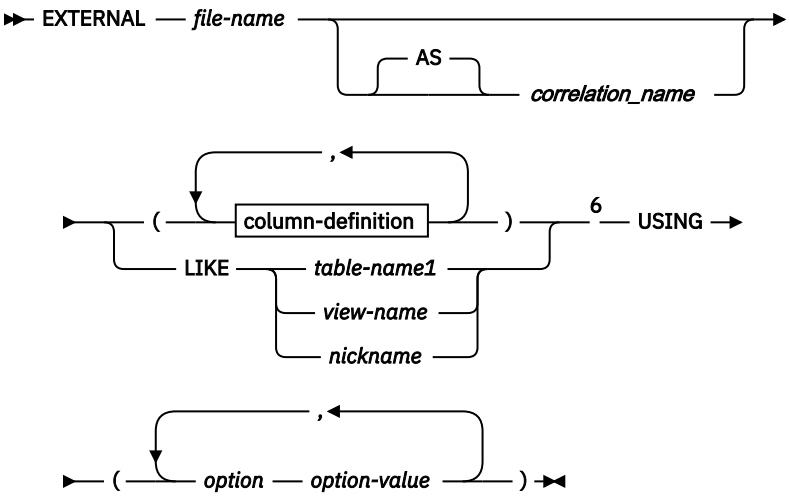


collection-derived-table

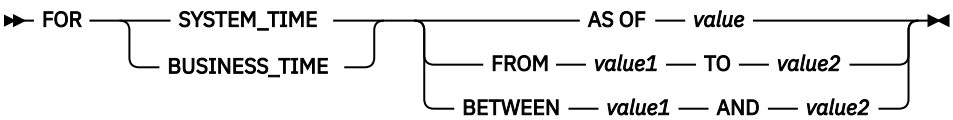
xmltable-expression



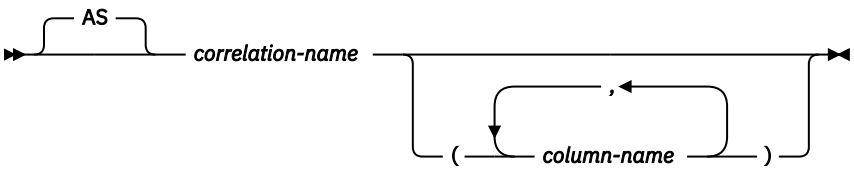
external-table-reference



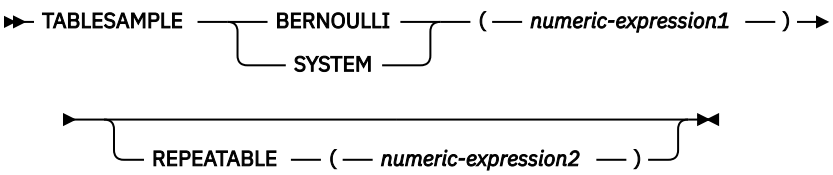
period-specification



correlation-clause



tablesample-clause



implementation-clause

continue-handler

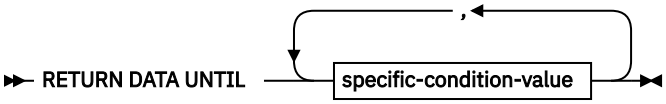
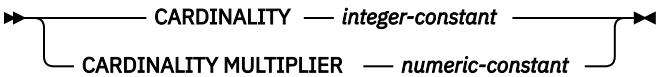
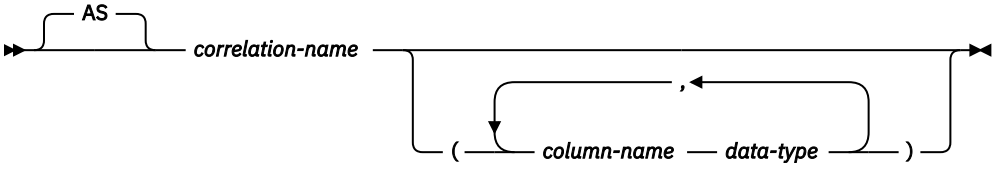


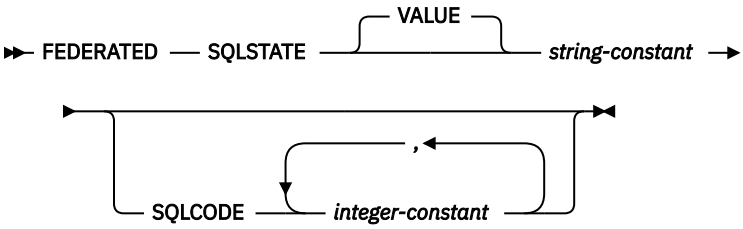
table-UDF-cardinality-clause



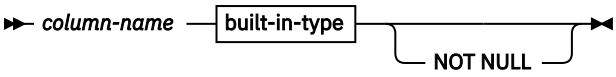
typed-correlation-clause



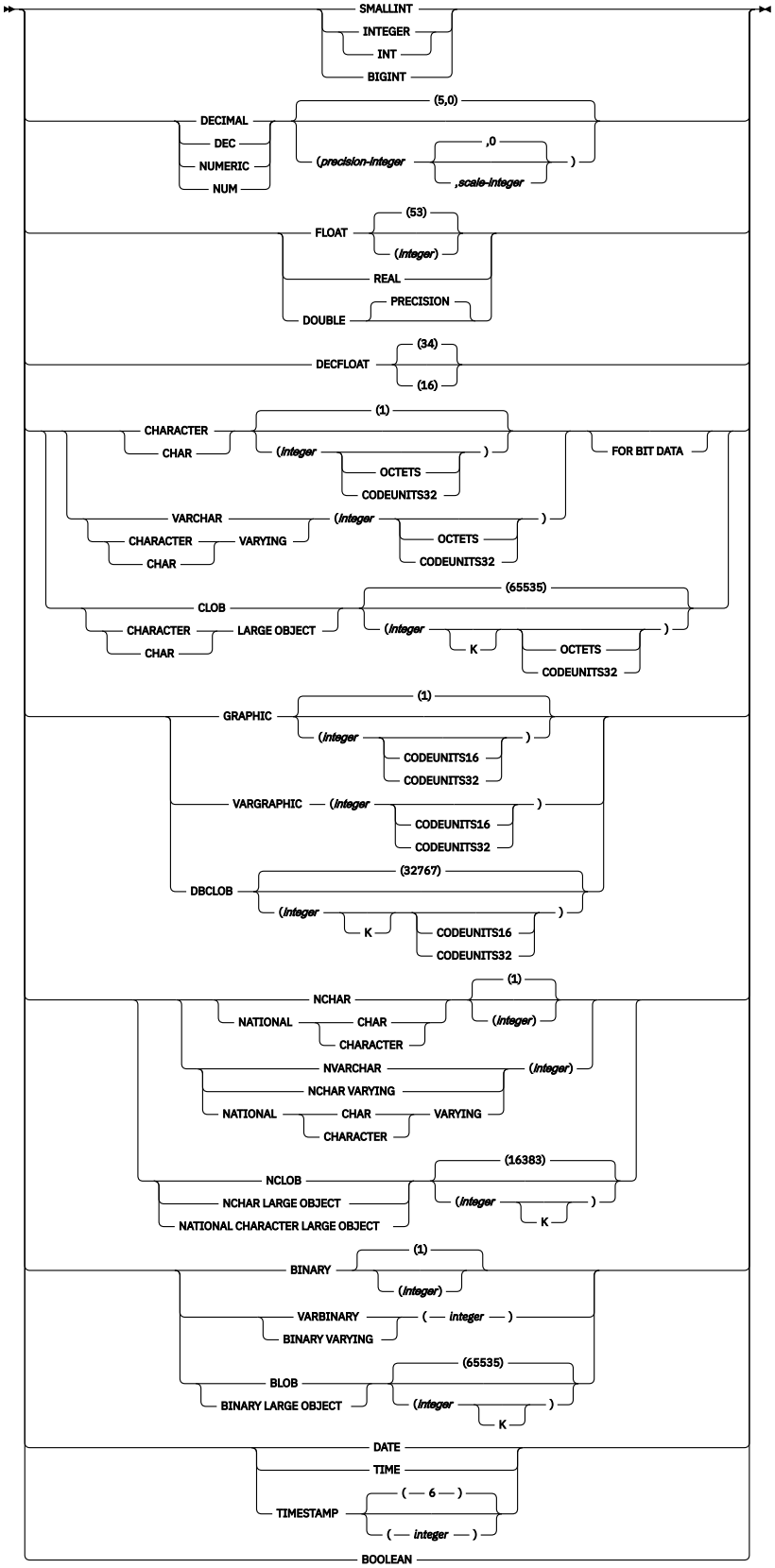
specific-condition-value



column-definition



built-in-type



Notes:

[1] The syntax for joined-table is covered in a separate topic; refer to "joined-table" on page 685.

[2] TABLE can be specified in place of LATERAL.

[3] The typed-correlation-clause is required for generic table functions. This clause cannot be specified for any other table functions.

[4] WITH ORDINALITY can be specified only if the argument to the UNNEST table function is one or more ordinary array variables or functions with ordinary array return types; an associative array variable or function with an associative array return type cannot be specified (SQLSTATE 428HT).

[5] An XMLTABLE function can be part of a table-reference. In this case, subexpressions within the XMLTABLE expression are in-scope of prior range variables in the FROM clause. For more information, see the description of "XMLTABLE".

[6] Specifying a LIKE clause or at least one column definition is not mandatory for an `INSERT INTO <table> SELECT FROM` statement, which acts as an implicit LIKE with respect to the INSERT target table.

A *table-reference* specifies an intermediate result table.

- If a single-table-reference is specified without a period-specification or a tablesample-clause, the intermediate result table is the rows of the table. If a period-specification is specified, the intermediate result table consists of the rows of the temporal table where the period matches the specification. If a tablesample-clause is specified, the intermediate result table consists of a sampled subset of the rows of the table.
- If a single-view-reference is specified without a period-specification, the intermediate result table is that view. If a period-specification is specified, temporal table references in the view consider only the rows where the period matches the specification.
- If a single-nickname-reference is specified, the intermediate result table is the data from the data source for that nickname.
- If an only-table-reference is specified, the intermediate result table consists of only the rows of the specified table or view without considering the applicable subtables or subviews.
- If an outer-table-reference is specified, the intermediate result table represents a virtual table based on all the subtables of a typed table or the subviews of a typed view.
- If an analyze_table-expression is specified, the result table contains the result of executing a specific data mining model by using an in-database analytics provider, a named model implementation, and input data.
- If a nested-table-expression is specified, the result table is the result of the specified fullselect.
- If a data-change-table-reference is specified, the intermediate result table is the set of rows that are directly changed by the searched UPDATE, searched DELETE, or INSERT statement that is included in the clause.
- If a table-function-reference is specified, the intermediate result table is the set of rows that are returned by the table function.
- If a collection-derived-table is specified, the intermediate result table is the set of rows that are returned by the UNNEST function.
- If an xmltable-expression is specified, the intermediate result table is the set of rows that are returned by the XMLTABLE function.
- If a joined-table is specified, the intermediate result table is the result of one or more join operations. For more information, see “joined-table” on page 685.
- If an external-table-reference is specified without a tablesample-clause, the intermediate result table is the rows of the external table that is represented by the specified file. If a tablesample-clause is specified, the intermediate result table consists of a sampled subset of the rows of the external table that is represented by the specified file.

single-table-reference

Each *table-name* specified as a table-reference must identify an existing table at the application server or an existing table at a remote server specified using a remote-object-name. The intermediate result table is the result of the table. If the *table-name* references a typed table, the intermediate result table is the UNION ALL of the table with all its subtables, with only the columns of the *table-name*. A period-specification can be used with a temporal table to specify the period from which

the rows are returned as the intermediate result table. A tablesample-clause can be used to specify that a sample of the rows be returned as the intermediate result table.

If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a non-null value *CTST* and *table-name* identifies a system-period temporal table, the table reference is executed as if it contained the following specification with the special register set to the null value:

```
table-name FOR SYSTEM_TIME AS OF CTST
```

If the CURRENT TEMPORAL BUSINESS_TIME special register is set to a non-null value *CTBT* and *table-name* identifies an application-period temporal table, the table reference is executed as if it contained the following specification with the special register set to the null value:

```
table-name FOR BUSINESS_TIME AS OF CTBT
```

single-view-reference

Each *view-name* specified as a table-reference must identify one of the following objects:

- An existing view at the application server
- A view at a remote server specified using a remote-object-name
- The *table-name* of a common table expression

The intermediate result table is the result of the view or common table expression. If the *view-name* references a typed view, the intermediate result table is the UNION ALL of the view with all its subviews, with only the columns of the *view-name*. A period-specification can be used with a view defined over a temporal table to specify the period from which the rows are returned as the intermediate result table.

If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a non-null value *CTST*, and *view-name* identifies a system-period temporal table, the table reference is executed as if it contained the following specification with the special register set to the null value:

```
view-name FOR SYSTEM_TIME AS OF CTST
```

If the CURRENT TEMPORAL BUSINESS_TIME special register is set to a non-null value *CTBT*, and *view-name* identifies an application-period temporal table, the table reference is executed as if it contained the following specification with the special register set to the null value:

```
view-name FOR BUSINESS_TIME AS OF CTBT
```

single-nickname-reference

Each *nickname* specified as a table-reference must identify an existing nickname at the application server. The intermediate result table is the result of the nickname.

only-table-reference

The use of ONLY(*table-name*) or ONLY(*view-name*) means that the rows of the applicable subtables or subviews are not included in the intermediate result table. If the *table-name* used with ONLY does not have subtables, then ONLY(*table-name*) is equivalent to specifying *table-name*. If the *view-name* used with ONLY does not have subviews, then ONLY(*view-name*) is equivalent to specifying *view-name*.

The use of ONLY requires the SELECT privilege on every subtable of *table-name* or subview of *view-name*.

outer-table-reference

The use of OUTER(*table-name*) or OUTER(*view-name*) represents a virtual table. If the *table-name* or *view-name* used with OUTER does not have subtables or subviews, then specifying OUTER is equivalent to not specifying OUTER. If the *table-name* does have subtables, the intermediate result table from OUTER(*table-name*) is derived from *table-name* as follows:

- The columns include the columns of *table-name* followed by the additional columns introduced by each of its subtables, if any. The additional columns are added on the right, traversing the subtable

hierarchy in depth-first order. Subtables that have a common parent are traversed in creation order of their types.

- The rows include all the rows of *table-name* and all the rows of its subtables. Null values are returned for columns that are not in the subtable for the row.

If the *view-name* does have subviews, the intermediate result table from OUTER(*view-name*) is derived from *view-name* as follows:

- The columns include the columns of *view-name* followed by the additional columns introduced by each of its subviews, if any. The additional columns are added on the right, traversing the subview hierarchy in depth-first order. Subviews that have a common parent are traversed in creation order of their types.
- The rows include all the rows of *view-name* and all the rows of its subviews. Null values are returned for columns that are not in the subview for the row.

The use of OUTER requires the SELECT privilege on every subtable of *table-name* or subview of *view-name*.

analyze_table-expression

table-name | view-name
The *table-name* or *view-name* variable must identify an existing table or view or identify the *table-name* of a common table expression that you define preceding the fullselect containing the table-reference. You can specify a nickname. However, in-database analytics are intended for local data, and retrieving the data for a nickname from another data source does not take advantage of the intended performance benefits.

ANALYZE_TABLE
Returns the result of executing a specific data mining model by using an in-database analytics provider, a named model implementation, and input data. A query referencing the ANALYZE_TABLE parameter cannot be a static SQL statement or a data definition language (DDL) statement. Input or output values cannot be of the following types:

- CHAR FOR BIT DATA or VARCHAR FOR BIT DATA
- BINARY or VARBINARY
- BLOB, CLOB, DBCLOB, or NCLOB
- BOOLEAN
- XML
- DB2SECURITYLABEL

IMPLEMENTATION '*string*'
Specifies how the expression is to be evaluated. The *string* parameter is a string constant whose maximum length is 1024 bytes. The specified value is used to establish a session with an in-database analytic provider. When you specify SAS as the provider, you must specify values for the following case-insensitive parameters:

PROVIDER
Currently, the only supported provider value is SAS.

ROUTINE_SOURCE_TABLE
Specifies a user table containing the DS2 code (and, optionally, any required format or metadata) to implement the algorithm that is specified by the ROUTINE_SOURCE_NAME parameter. DS2 is a procedural language processor for SAS, designed for data modeling, stored procedures, and data extraction, transformation, and load (ETL) processing.

The routine source table has a defined structure (see the examples at the end of the "*analyze_table-expression*" section) and, in a partitioned database environment, must be on the catalog database partition. The table cannot be a global temporary table. The MODELDS2 column for a particular row must not be empty or contain the null value. If the value of the MODELFORMATS or MODELMETADATA column is not null, the value must have a length greater

than 0. If you do not specify a table schema name, the value of the CURRENT SCHEMA special register is used.

ROUTINE_SOURCE_NAME
Specifies the name of the algorithm to use.

For example:

```
IMPLEMENTATION
  'PROVIDER=SAS;
  ROUTINE_SOURCE_TABLE=ETLIN.SOURCE_TABLE;
  ROUTINE_SOURCE_NAME=SCORING_FUN1;'
```

If the table name, schema name, or algorithm name contains lowercase or mixed-case letters, specify delimited identifiers, as shown in the following example:

```
IMPLEMENTATION
  'PROVIDER=SAS;
  ROUTINE_SOURCE_TABLE="ETLin"."Source_Table";
  ROUTINE_SOURCE_NAME="Scoring_Fun1";'
```

The following examples show you how to use the ANALYZE_TABLE expression.

SAS tooling helps you to define a table to store model implementations for scoring functions. A row in this table stores an algorithm that is written in DS2, with any required SAS format information and metadata. The MODELNAME column serves as the primary key. For a particular value of the ROUTINE_SOURCE_NAME parameter, at most one row is retrieved from the table that the ROUTINE_SOURCE_TABLE parameter specifies. For example:

```
CREATE TABLE ETLIN.SOURCE_TABLE (
  MODELNAME VARCHAR(128) NOT NULL PRIMARY KEY,
  MODELDS2 BLOB(4M) NOT NULL,
  MODELFORMATS BLOB(4M),
  MODELMETADATA BLOB(4M)
);
```

The MODELNAME column contains the name of the algorithm. The MODELDS2 column contains the DS2 source code that implements the algorithm. The MODELFORMATS column contains the aggregated SAS format definition that the algorithm requires. If the algorithm does not require a SAS format, this column contains the null value. The MODELMETADATA column contains any additional metadata that the algorithm requires. If the algorithm does not require any additional metadata, this column contains the null value. If the SAS EP installer creates the table, it might include additional columns.

- Use the data in columns C1 and C2 in table T1 as input data with the scoring model SCORING_FUN1, whose implementation is stored in ETLIN.SOURCE_TABLE:

```
WITH sas_score_in (c1,c2) AS
  (SELECT c1,c2 FROM t1)
  SELECT *
    FROM sas_score_in ANALYZE_TABLE(
      IMPLEMENTATION
        'PROVIDER=SAS;
         ROUTINE_SOURCE_TABLE=ETLIN.SOURCE_TABLE;
         ROUTINE_SOURCE_NAME=SCORING_FUN1;');
```

- Use all the data in the table T2 with the scoring model SCORING_FUN2, whose implementation is stored in ETLIN.SOURCE_TABLE:

```
SELECT *
  FROM t2 ANALYZE_TABLE(
    IMPLEMENTATION
      'PROVIDER=SAS;
       ROUTINE_SOURCE_TABLE=ETLIN.SOURCE_TABLE;
       ROUTINE_SOURCE_NAME=SCORING_FUN2;');
```

- Use all the data in view V1 with the scoring model SCORING_FUN3, whose implementation is stored in ETLIN.SOURCE_TABLE, and return the output in ascending order of the first output column:

```
SELECT *
  FROM v1 ANALYZE_TABLE(
    IMPLEMENTATION
      'PROVIDER=SAS;
       ROUTINE_SOURCE_TABLE=ETLIN.SOURCE_TABLE;
       ROUTINE_SOURCE_NAME=SCORING_FUN3;')
  ORDER BY 1;
```

nested-table-expression

A fullselect in parentheses is called a *nested table expression*. The intermediate result table is the result of that fullselect. The columns of the result do not need unique names, but a column with a non-unique name cannot be explicitly referenced. If LATERAL is specified, the fullselect can include correlated references to results columns of table references specified to the left of the nested table expression. If the nested table expression involves data from a federated data source, a continue-handler can be specified to tolerate certain error conditions from the data source.

An expression in the select list of a nested table expression that is referenced within, or is the target of, a data change statement within a fullselect is valid only when it does not include:

- A function that reads or modifies SQL data
- A function that is non-deterministic
- A function that has external action
- An OLAP function

If a view is referenced directly in, or as the target of a nested table expression in a data change statement within a FROM clause, the view must meet either of the following conditions:

- Be symmetric (have WITH CHECK OPTION specified)
- Satisfy the restriction for a WITH CHECK OPTION view

If the target of a data change statement within a FROM clause is a nested table expression, the following restrictions apply:

- Modified rows are not requalified
- WHERE clause predicates are not reevaluated
- ORDER BY or FETCH FIRST operations are not redone

A nested table expression can be used in the following situations:

- In place of a view to avoid creating the view (when general use of the view is not required)
- When the required intermediate result table is based on host variables

data-change-table-reference

A *data-change-table-reference* clause specifies an intermediate result table. This table is based on the rows that are directly changed by the searched UPDATE, searched DELETE, or INSERT statement that is included in the clause. A *data-change-table-reference* can be specified as the only *table-reference* in the FROM clause of the outer fullselect that is used in a *select-statement*, a SELECT INTO statement, or a common table expression. A *data-change-table-reference* can be specified as the only table reference in the only fullselect in a SET Variable statement (SQLSTATE 428FL). The target table or view of the data change statement is considered to be a table or view that is referenced in the query; therefore, the authorization ID of the query must have SELECT privilege on that target table or view. A *data-change-table-reference* clause cannot be specified in a view definition, materialized query table definition, or FOR statement (SQLSTATE 428FL).

The target of the UPDATE, DELETE, or INSERT statement cannot be a temporary view defined in a common table expression (SQLSTATE 42807) or a nickname (SQLSTATE 25000).

Expressions in the select list of a view or fullselect as target of a data change statement in a *table-reference* can be selected only if OLD TABLE is specified or the expression does not include the following elements (SQLSTATE 428G6):

- A subquery

- A function that reads or modifies SQL data
- A function is that is non-deterministic or has an external action
- An OLAP function
- A NEXT VALUE FOR *sequence* reference

FINAL TABLE
Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement as they exist at the completion of the data change statement. If there are AFTER triggers or referential constraints that result in further operations on the table that is the target of the SQL data change statement, an error is returned (SQLSTATE 560C6). If the target of the SQL data change statement is a view that is defined with an INSTEAD OF trigger for the type of data change, an error is returned (SQLSTATE 428G3).

NEW TABLE
Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement before the application of referential constraints and AFTER triggers. Data in the target table at the completion of the statement might not match the data in the intermediate result table because of additional processing for referential constraints and AFTER triggers.

OLD TABLE
Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement as they existed before the application of the data change statement.

(searched-update-statement)
Specifies a searched UPDATE statement. A WHERE clause or a SET clause in the UPDATE statement cannot contain correlated references to columns outside of the UPDATE statement.

(searched-delete-statement)
Specifies a searched DELETE statement. A WHERE clause in the DELETE statement cannot contain correlated references to columns outside of the DELETE statement.

(insert-statement)
Specifies an INSERT statement. A fullselect in the INSERT statement cannot contain correlated references to columns outside of the fullselect of the INSERT statement.

The content of the intermediate result table for a *data-change-table-reference* is determined when the cursor opens. The intermediate result table contains all manipulated rows, including all the columns in the specified target table or view. All the columns of the target table or view for an SQL data change statement are accessible using the column names from the target table or view. If an INCLUDE clause was specified within a data change statement, the intermediate result table will contain these additional columns.

table-function-reference

In general, a table function, together with its argument values, can be referenced in the FROM clause of a SELECT in exactly the same way as a table or view. Each *function-name* together with the types of its arguments, specified as a table reference must resolve to an existing table function at the application server. There are, however, some special considerations which apply.

- **Table function column names**: Unless alternative column names are provided following the *correlation-name*, the column names for the table function are those specified in the RETURNS or RETURNS GENERIC TABLE clause of the CREATE FUNCTION statement. This is analogous to the names of the columns of a table, which are defined in the CREATE TABLE statement.
- **Table function resolution**: The arguments specified in a table function reference, together with the function name, are used by an algorithm called *function resolution* to determine the exact function to be used. This is no different from what happens with other functions (such as scalar functions) that are used in a statement.

- **Table function arguments**: As with scalar function arguments, table function arguments can generally be any valid SQL expression. The following examples are valid syntax:

```
Example 1:  SELECT c1
            FROM TABLE( tf1('Zachary') ) AS z
            WHERE c2 = 'FLORIDA';

Example 2:  SELECT c1
            FROM TABLE( tf2 (:hostvar1, CURRENT DATE) ) AS z;

Example 3:  SELECT c1
            FROM t
            WHERE c2 IN
                     (SELECT c3 FROM
                      TABLE( tf5(t.c4) ) AS z  -- correlated reference
                     )                         -- to previous FROM clause
```

```
Example 4:  SELECT c1
            FROM TABLE( tf6('abcd') )                -- tf6 is a generic
                AS z (c1 int, c2 varchar(100)) -- java table function
```

- **Table functions that modify SQL data**: Table functions that are specified with the MODIFIES SQL DATA option can be used only as the last table reference in a *select-statement*, *common-table-expression*, or RETURN statement that is a subselect, a SELECT INTO, or a *row-fullselect* in a SET statement. Only one table function is allowed in one FROM clause, and the table function arguments must be correlated to all other table references in the subselect (SQLSTATE 429BL). The following examples have valid syntax for a table function with the MODIFIES SQL DATA property:

```
Example 1:  SELECT c1
            FROM TABLE( tfmod('Jones') ) AS z

Example 2:  SELECT c1
            FROM t1, t2, TABLE( tfmod(t1.c1, t2.c1) ) AS z

Example 3:  SET var =
            (SELECT c1
            FROM TABLE( tfmod('Jones') ) AS z

Example 4:  RETURN SELECT c1
            FROM TABLE( tfmod('Jones') ) AS z

Example 5:  WITH v1(c1) AS
            (SELECT c1
            FROM TABLE( tfmod(:hostvar1) ) AS z)
            SELECT c1
            FROM v1, t1 WHERE v1.c1 = t1.c1
```

```
Example 6: SELECT z.*
           FROM t1, t2, TABLE( tfmod(t1.c1, t2.c1) )
           AS z (col1 int)
```

collection-derived-table

A *collection-derived-table* can be used to convert the elements of an array into values of a column in separate rows. If WITH ORDINALITY is specified, an extra column of data type INTEGER is appended. This column contains the position of the element in the array. The columns can be referenced in the select list and the in rest of the subselect by using the names specified for the columns in the correlation-clause. The *collection-derived-table* clause can be used only in a context where arrays are supported (SQLSTATE 42887). See the "UNNEST table function" for details.

xmltable-expression

An *xmltable-expression* specifies an invocation of the built-in XMLTABLE function which determines the intermediate result table. See XMLTABLE for more information.

external-table-reference

An external table resides in a text-based, delimited or non-delimited file outside of a database. An *external-table-reference* specifies the name of the file that contains an external table.

column-definition

The attributes of a column.

column-name
Names a column of the table. The name cannot be qualified, and the same name cannot be used for more than one column of the table (SQLSTATE 42711).

built-in-type
One of the following built-in data types:

SMALLINT
A small integer.

[INTEGER | INT]
A large integer.

BIGINT
A big integer.

[DECIMAL | DEC | NUMERIC | NUM](*precision-integer, scale-integer*)
A decimal number.

- The precision integer specifies the total number of digits. It must be in the range 1 - 31. The default is 5.
- The scale integer specifies the number of digits to the right of the decimal point. It cannot be negative and cannot exceed the precision. The default is 0.

FLOAT(*integer*)
A single or double-precision floating-point number. If the specified length is in the range:

- 1 - 24, the number uses single precision
- 25 - 53, the number uses double-precision

Instead of FLOAT, you can specify:

REAL
For single precision floating-point.

DOUBLE
For double-precision floating-point.

DOUBLE PRECISION
For double-precision floating-point.

FLOAT
For double-precision floating-point.

DECFLOAT*(precision-integer)*
A decimal floating-point number. The precision integer specifies the total number of digits, which can be either 16 or 34. The default is 34.

[CHARACTER | CHAR](*integer* [OCTETS | CODEUNITS32])
A fixed-length character string of the specified number of code units. This number can range from 1 - 255 OCTETS or from 1 - 63 CODEUNITS32. The default is 1.

[VARCHAR | CHARACTER VARYING | CHAR VARYING](*integer* [OCTETS | CODEUNITS32])
A varying-length character string with a maximum length of the specified number of code units. This number can range from 1 - 32672 OCTETS or from 1 - 8168 CODEUNITS32.

FOR BIT DATA
Specifies that the contents of the column are to be treated as bit (binary) data. During data exchange with other systems, code page conversions are not performed. Comparisons are done in binary, irrespective of the database collating sequence.

The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

[CLOB | CHARACTER LARGE OBJECT | CHAR LARGE OBJECT](*integer* [K] [OCTETS | CODEUNITS32])
A character large object string with a maximum length of the specified number of code units. The default maximum length is 65,535 bytes.

If you want to multiply the length integer by 1024, specify a K (kilo) multiplier.

- Regardless of whether you use a K multiplier, the resulting length is limited by the maximum length of a CLOB column in an external table, which is 65,535 OCTETS, 32,767 CODEUNITS16, or 16,383 CODEUNITS32. Note that 64K OCTETS and 16K CODEUNITS32 each exceed the maximum length by one, and so are not allowed.
- Any number of spaces (including zero spaces) are allowed between the data type and the length specification or between the length integer and the K multiplier. For example, the following specifications are all equivalent and valid:

```
CLOB(50K)
CLOB(50 K)
CLOB (50   K)
```

- The K multiplier can be specified in either uppercase or lowercase.

In a Unicode database, the default string units for a character string data type are determined by the value of the NLS_STRING_UNITS global variable or *string_units* database configuration parameter. In a non-Unicode database, the default string units for character string data types are OCTETS.

OCTETS
Specifies that the units of the length attribute are bytes.

CODEUNITS32
Specifies that the units of the length attribute are Unicode UTF-32 code units, which approximates counting in characters. This does not affect the underlying code page of the data type. The actual length of a data value is determined by counting the UTF-32 code units as if the data were converted to UTF-32. CODEUNITS32 can be specified only in a Unicode database (SQLSTATE 560AA).

GRAPHIC(*integer* [CODEUNITS16 | CODEUNITS32])
A fixed-length graphic string of the specified length, which can range from 1 - 127 double bytes, 1 - 127 CODEUNITS16, or 1 - 63 CODEUNITS32. The default length is 1.

VARGRAPHIC(*integer* [CODEUNITS16 | CODEUNITS32])
A varying-length graphic string of the specified maximum length, which can range from 1 - 16336 double bytes, 1 - 16336 CODEUNITS16, or 1 - 8168 CODEUNITS32.

DBCLOB(*integer* [K] [CODEUNITS16 | CODEUNITS32])
A character large object string of the specified maximum length in double bytes, Unicode UTF-16 code units, or Unicode UTF-32 code units. The default maximum length is 32,767 double bytes.

If you want to multiply the length integer by 1024, specify a K (kilo) multiplier.

- Regardless of whether you use a K multiplier, the resulting length is limited by the maximum length of a DBCLOB column in an external table, which is 32,767 CODEUNITS16 or 16,383 CODEUNITS32. Note that 32K CODEUNITS16 and 16K CODEUNITS32 each exceed the maximum length by one, and so are not allowed.
- Any number of spaces (including zero spaces) are allowed between the data type and the length specification or between the length integer and the K multiplier. For example, the following specifications are all equivalent and valid:

```
DBCLOB(50K)
DBCLOB(50 K)
DBCLOB (50   K)
```

- The K multiplier can be specified in either uppercase or lowercase.

In a Unicode database, the default string units for a character string data type are determined by the value of the NLS_STRING_UNITS global variable or *string_units* database configuration parameter. In a non-Unicode database, the default string units for character string data types is CODEUNITS16.

CODEUNITS16
Specifies that the units of the length attribute are Unicode UTF-16 code units, which is the same as counting in double bytes. CODEUNITS16 can be specified only in a Unicode database (SQLSTATE 560AA).

CODEUNITS32
Specifies that the units of the length attribute are Unicode UTF-32 code units. This does not affect the underlying code page of the data type. The actual length of a data value is determined by counting the UTF-32 code units as if the data were converted to UTF-32. CODEUNITS32 can be specified only in a Unicode database (SQLSTATE 560AA).

[NATIONAL CHARACTER | NATIONAL CHAR | NCHAR](*integer*)
A fixed-length string of the specified length. The default length is 1.

The NATIONAL CHARACTER type maps to either a fixed-length character or a fixed-length graphic string, depending on the value of the *nchar_mapping* database configuration parameter, which also defines the string units.

[NATIONAL CHARACTER VARYING | NATIONAL CHAR VARYING | NCHAR VARYING | NVARCHAR](*integer*)
A varying-length string of the specified maximum length.

The NATIONAL CHARACTER VARYING type maps to either a varying-length character or a varying-length graphic string, depending on the value of the *nchar_mapping* database configuration parameter, which also defines the string units.

[NATIONAL CHARACTER LARGE OBJECT | NCHAR LARGE OBJECT | NCLOB](*integer* [K])
A large object string of the specified maximum length. The default maximum length is 16,383 double bytes.

This data type maps to either a character large object (CLOB) or a double-byte character large object (DBCLOB), depending on the current value of the *nchar_mapping* database configuration parameter, which also defines the string units. See the description of the CLOB or DBCLOB parameter (whichever applies) for information about possible values for the length integer and how to use a K (kilo) multiplier.

BINARY(*integer*)
A fixed-length binary string of the specified length, which must be in the range 1 - 255 bytes. The default length is 1.

[VARBINARY | BINARY VARYING](*integer*)
A varying-length binary string of the specified maximum length, which must be in the range 1 - 32672 bytes.

[BLOB | BINARY LARGE OBJECT](*integer* [K])
A binary large object string with a maximum length of the specified number of code units. The default maximum length is 65,535 bytes.

If you want to multiply the length integer by 1024, specify a K (kilo) multiplier.

- Regardless of whether you use a K multiplier, the resulting length is limited by the maximum length of a BLOB column in an external table, which is 65,535 bytes. Note that 64K exceeds the maximum length by one, and so is not allowed.
- Any number of spaces (including zero spaces) are allowed between the data type and the length specification or between the length integer and the K multiplier. For example, the following specifications are all equivalent and valid:

```
BLOB(50K)
BLOB(50 K)
BLOB (50   K)
```

- The K multiplier can be specified in either uppercase or lowercase.

DATE
A date.

TIME
A time.

TIMESTAMP(*integer*) or TIMESTAMP
A timestamp. The integer specifies the number of decimal places for fractions of seconds, from 0 (seconds) to 12 (picoseconds). The default is 6 (microseconds).

BOOLEAN
A Boolean value.

LIKE *table-name1* or *view-name* or *nickname*
Specifies that the columns of the table have the same name and description as the columns of the specified table (*table-name1*), view (*view-name*), or nickname (*nickname*). The specified table, view, or nickname must either exist in the catalog or must be a declared temporary table. A typed table or typed view cannot be specified (SQLSTATE 428EC).

The use of LIKE is an implicit definition of *n* columns, where *n* is the number of columns in the identified table (including implicitly hidden columns), view, or nickname. A column of the new table that corresponds to an implicitly hidden column in the existing table will also be defined as implicitly hidden. The implicit definition depends on what is specified after LIKE:

- If a table is specified, then the implicit definition includes the column name, data type, hidden attribute, and nullability characteristic of each of the columns of that table. If EXCLUDING COLUMN DEFAULTS is not specified, then the column default is also included.
- If a view is specified, then the implicit definition includes the column name, data type, and nullability characteristic of each of the result columns of the fullselect defined in that view. The data types of the view columns must be data types that are valid for columns of a table.
- If a nickname is specified, then the implicit definition includes the column name, data type, and nullability characteristic of each column of that nickname.
- If a protected table is specified, the new table inherits the same security policy and protected columns as the identified table.
- If a table is specified and if that table contains a row-begin column, row-end column, or transaction-start-ID column, the corresponding column of the new table inherits only the data type of the source column. The new column is not considered a generated column.
- If a table that includes a period is specified, the new table does not inherit the period definition.
- If a system-period temporal table is specified, the new table is not a system-period temporal table.
- If a random distribution table that uses the **random by generation** method is specified, and if the new table that is being created does not share the same table distribution, the **RANDOM_DISTRIBUTION_KEY** column that is used to generate the random distribution values is not included.

Column default and identity column attributes can be included or excluded, based on the copy-attributes clauses. The implicit definition does not include any other attributes of the identified table, view, or nickname. Consequently, the new table does not have any primary key, unique constraints, foreign key constraints, referential integrity constraints, triggers, indexes, ORGANIZE BY specification, or PARTITIONING KEY specification. The table is created in the table space implicitly or explicitly specified by the IN clause, and the table has any other optional clause only if the optional clause is specified.

When a table is identified in the LIKE clause and that table contains a ROW CHANGE TIMESTAMP column, the corresponding column of the new table inherits only the data type of the ROW CHANGE TIMESTAMP column. The new column is not considered to be a generated column.

If a table is specified, and if row or column level access control is activated for that table, it is not inherited by the new table.

option
The following options control the loading of data to or retrieval of data from an external-table file. The value of each option is a text string and is not case-sensitive.

BOOLSTYLE or BOOLEAN_STYLE
During a load operation, all Boolean values must use the same style. This option specifies the Boolean style that is to be used:

- 1_0 (this is the default)
- T_F
- Y_N
- YES_NO
- TRUE_FALSE

CARDINALITY
Non-zero positive integer value to override the estimation of the expected number of returned rows.

CCSID
The coded character set identifier (CCSID) of the input data file. The value can be any valid integer value from the CCSID specification. There is no default value. The CCSID and ENCODING options are mutually exclusive when the value of the ENCODING option is UTF8, LATIN9, or INTERNAL.

Which styles are used for dates and times depends on whether a CCSID is specified:

- When a CCSID is specified, and when DATESTYLE, TIMESTYLE, DATEDELIM, or TIMEDELIM are not specified, the values or defaults for DATE_FORMAT, TIME_FORMAT, and TIMESTAMP_FORMAT are used.
- When a CCSID is not specified, and when TIMESTAMP_FORMAT, DATE_FORMAT or TIME_FORMAT are not specified, the values or defaults for DATESTYLE, TIMESTYLE, DATEDELIM, and TIMEDELIM are used.

COMPRESS
For a load operation or an unload operation, whether the data file data is compressed:

GZIP
The data file data is compressed by using the GZIP compression algorithm.

NO
The data file data is not compressed. This is the default.

LZ4
The data file data is compressed by using the LZ4 compression algorithm.

The COMPRESS option cannot be specified if the value of the REMOTESOURCE option is GZIP or LZ4.

CRINSTRING
How to interpret an unescaped carriage-return (CR) or carriage-return line-feed (CRLF) character:

TRUE or ON
An unescaped CR character is interpreted as data, not as a record delimiter.

FALSE or OFF
An unescaped CR is interpreted as a record delimiter. This is the default.

Use fixed-length format for **CRINSTRING** only if the value of the **CtrlChars** option is set to OFF.

CTRLCHARS
Whether to allow an ASCII value 1 - 31 in a CHAR or VARCHAR field. Any NULL, CR, or LF characters must be escaped. Allowed values are:

TRUE or ON
An ASCII value 1 - 31 in a CHAR or VARCHAR field is allowed.

If fixed-length format is enabled, all unescaped characters are allowed.

FALSE or OFF
An ASCII value 1 - 31 in a CHAR or VARCHAR field is not allowed. This is the default.

If fixed-length format is enabled, unescaped characters cause an error.

Exceptions for fixed-length format:

- \t, \n
- \r if the CRinString option is set to ON

DATAOBJECT or FILE_NAME

The fully-qualified name of the file (or any medium that can be treated as a file) that is to contain the external table to be created. This option is mandatory when the name of the file is not specified immediately after the table name; otherwise, it is not allowed.

When both the REMOTESOURCE option is set to LOCAL (this is its default value) and the **extbl_strict_io** configuration parameter is set to NO, the path to the external table file is an absolute path and must be one of the paths specified by the **extbl_location** configuration parameter. Otherwise, the path to the external table file is relative to the path that is specified by the **extbl_location** configuration parameter followed by the authorization ID of the table definer. For example, if **extbl_location** is set to `/home/xyz` and the authorization ID of the table definer is `user1`, the path to the external table file is relative to `/home/xyz/user1/`.

The file name must be a valid UTF-8 string.
For a load operation, the following conditions apply:

- The file must already exist.
- Required permissions:
  - If the external table is a named external table, the owner must have read permission for the file and write permission for the LOGDIR directory.
  - If the external table is a transient external table, the authorization ID of the statement must have read permission for the file and write permission for the LOGDIR directory.

For an unload operation, the following conditions apply:

- If the file exists, it is overwritten.
- Required permissions:
  - If the external table is a named external table, the owner must have read and write permission for the directory of this file.
  - If the external table is transient, the authorization ID of the statement must have read and write permission for the directory of this file.

DATEDELIM

The delimiter character that separates the components of a date, according to the format specified by the DATESTYLE option. If you specify an empty string, there is no delimiter between the date components, and days and months must be specified as two-digit numbers. When DATESTYLE is set to MONDY or MONDY2, the default DATEDELIM value is a space. The TIMESTAMP_FORMAT and DATEDELIM options are mutually exclusive.

DATESTYLE

How to interpret the date format. For days or months in the range 1 - 9, use 1 digit, 2 digits, or a space followed by a single digit. When the DATEDELIM option is a space, you can specify a comma after the day. An error occurs if you:

- Specify zero for a day, month, or year
- Specify a nonexistent date (for example, 32 August or 30 February)

The DATESTYLE option and the DATE_FORMAT or TIMESTAMP_FORMAT option are mutually exclusive.

Table 109. Possible values for the DateStyle option. The example shows how the date 21 March 2014 would be represented when DATEDELIM is set to '-'.

Value	Description	Example
YMD	4-digit year, 2-digit month, 2-digit day. This is the default.	2014-03-21
DMY	2-digit day, 2-digit month, 4-digit year.	21-03-2014
MDY	2-digit month, 2-digit day, 4-digit year.	03-21-2014
MONDY	3-character month, 2-digit day, 4-digit year.	Mar 21 2014
DMONY	2-digit day, 3-character month, 4-digit year.	21-Mar-2014
Y2MD	2-digit year, 2-digit month, 2-digit day. Not supported for unloads.	14-03-21
DMY2	2-digit day, 2-digit month, 2-digit year. Not supported for unloads.	21-03-14
MDY2	2-digit month, 2-digit day, 2-digit year. Not supported for unloads.	03-21-14
MONDY2	3-character month, 2-digit day, 2-digit year. Not supported for unloads.	Mar 21 14
DMONY2	2-digit day, 3-character month, 2-digit year. Not supported for unloads.	21-Mar-14

DATETIMEDELIM
A single-byte character that separates the date component and time component of the timestamp data type.

The default delimiter is a space (' ').

Between the date component and the time component, a delimiter is not required. For example, both of the following values are valid:

```
2010-10-10 10:10:10
2010-10-1010:10:10
```

DATE_FORMAT
The format of the date field in the data file. The value can be any of the date format strings that are accepted by the "TIMESTAMP_FORMAT " on page 527. The default is YYYY-MM-DD. The DATE_FORMAT option and the DATEDELIM or DATESTYLE option are mutually exclusive.

DECIMALDELIM or DECIMAL_CHARACTER
The decimal delimiter for the data types FLOAT, DOUBLE, TIME, and TIMESTAMP. Allowed values are ',' and '.'.

DECPLUSBLANK
Specifies how the positive decimal value is represented during the unload operation.

You can specify one of the following values for this option:

NONE
This is the default.

This value represents a positive decimal value without a sign.

PLUS
Specifies that a positive decimal value is represented by a '+' sign.

BLANK
Specifies that a positive decimal value is represented by a blank sign instead of a '+' sign.

If you specify the DECPLUSBLANK option for the load operation, the output is not affected.

Examples for a table test with ddl (decimal (6,2)) and all the available values for the DECPLUSBLANK option:

```
1234
-4563
```

- Create external table '/tmp/unload.txt' using (DECPLUSBLANK NONE) as select * from test:

```
unload.txt
1234.00
-4563.00
```

- Create external table '/tmp/unload.txt' using (DECPLUSBLANK PLUS) as select * from test:

```
unload.txt
+1234.00
-4563.00
```

- Create external table '/tmp/unload.txt' using (DECPLUSBLANK BLANK) as select * from test:

```
unload.txt
 1234.00
-4563.00
```

DELIMITER or COLUMN_DELIMITER

The character that is used to delimit the fields of an input or output record. The default is a vertical bar (`'|'`).

You can specify a character in the 7-bit ASCII range (decimal 1 - 127) in any of the following ways:

- As a single character (for example `DELIMITER ';'`)
- By specifying its corresponding ASCII decimal value (for example, `DELIMITER 59` or `DELIMITER '59'`)
- By specifying its corresponding ASCII hex value (for example, `DELIMITER x'3B'`)

The decimal range 128 - 255 is supported only with the ISO character set input file by specifying its corresponding ASCII decimal value or hex value. If the input file is in the UTF8 character set, this delimiter value range is not supported.

ENCODING

The type of data in the file:

UTF8

The file uses UTF8 encoding for all character data.

LATIN9

The file uses LATIN9 encoding for all character data.

INTERNAL

This is the default option.

The file uses a mixture of both UTF8 and LATIN9 encoding.

Files are encoded in Netezza internal format and therefore should be used only for files that are extracted from Netezza by using ENCODING (INTERNAL).

When the target column is CODEUINTS32 (NCHAR/VARCHAR), the input data is validated to be valid UTF-8 characters.

This option is supported only in a Unicode database.

DBCS_GRAPHIC

This value is allowed only for a load operation, not an unload operation. If this value is specified, the CCSID option must also be specified. During the load operation, fields of type GRAPHIC or VARGRAPHIC are encoded using the double-byte character set of the specified CCSID; fields of all other types are encoded using the mixed-byte character set of the specified CCSID.

Note: ENCODING cannot be set to DBCS_GRAPHIC for a DEL file that was created by the EXPORT utility, because such DEL files are encoded using a single character set.

The CCSID and ENCODING options are mutually exclusive when the value of the ENCODING option is UTF8, LATIN9, or INTERNAL.

ESCAPECHAR or ESCAPE_CHARACTER
Which character is to be regarded as an escape character. An escape character indicates that the character that follows it, which would otherwise be treated as a field-delimiter character or end-of-row sequence character, is instead treated as part of the value in the field. The escape character is ignored for graphic-string data. There is no default.

FILLRECORD
For a load operation, the field of a record are loaded into the columns of a target table from left to right. This option specifies whether an input record can contain fewer fields than there are columns defined for the target table:

TRUE or ON
An input line can contain fewer fields, provided that all columns for which a value is missing are nullable. Missing values are set to NULL. If one or more columns for which a value is missing is not nullable, the record is rejected.

FALSE or OFF
An input line that contains fewer columns is rejected. This is the default.

FORMAT or FILE_FORMAT
The data format of the source file:

TEXT
The data to be loaded or unloaded is in text-delimited format. This is the default.

INTERNAL
The data is in an internal format used by Netezza Platform Software (NPS). This value is valid only when loading data from a file to the database, not when unloading data to a file. If this value is specified for the FORMAT option, the following options, and only these options, must also be specified:

- DATAOBJECT or FILE_NAME.
- REMOTESOURCE, SWIFT or S3. If the REMOTESOURCE option is specified, it must have the value LOCAL or YES.

  Attention: In the container-only release of Db2 Version 11.5 Mod Pack 1 or later releases, AZURE is compatible, in addition to SWIFT and S3.
- COMPRESS. This must be set to GZIP.

BINARY
The data is in an internal format that is used by Db2.

FIXED
The data is in fixed-length format.

Fixed-length format is supported only for load operations.

Files in fixed-length format use ordinal positions, which are offsets, to identify where fields are within the record.

Note:

- The following external table options are not supported for the fixed-length format:
  - Delimiter
  - Encoding
  - EscapeChar
  - FillRecord
  - IgnoreZero

  - IncludeZeroSeconds
  - Lfinstring
  - QuotedValue
  - RequireQuotes
  - TimeExtraZeros
  - TruncString
- There are no field delimiters.
- An end-of-record delimiter is required even for the last record.
- Usually, data in fixed-length format files does not have decimal delimiters or time delimiters because delimiters are not necessary and use space.
- The locations of delimiters are fixed and specified in the layout definition because the fields are fixed in size. This definition comes with the fixed-length format data file.
- To load fixed-format data into the database, you must define the target data type for the fields and the locations within the record.
- You do not have to load all fields in a fixed-length format file. You can skip them by using the *filler* specification.
- The order of fields in the data file must match the order in the target table. Alternatively, you must create an external table definition that specifies the order of the fields as database columns.
- You can change the field order by using an external table definition in combination with an insert-select statement.
- Typically, unknown values or null values are represented by known data patterns that are classified as representing null.

The following parameters apply when the FORMAT option of the external table is set to FIXED:

LAYOUT
Mandatory.

A layout is an ordered collection of zone or field definitions. It defines the location of the fields of the input record.

Specify comma-separated zone definitions within braces { }.

Each zone definition is made up of mutually exclusive, non-overlapping clauses.

No default value.

The clauses must be in the following order, even if some of them are optional and can be empty:

USE TYPE
Optional.

Indicates whether a zone is a normal data zone, a reference zone, or a filler zone.

For data zones, this value is omitted.

A reference zone is specified as **REF**. This specification implies that the zone is referred by another zone for zone length or null values.

A filler zone is specified as **FILLER**. Filler zones specify that the bytes or characters are treated as fillers in a data file.

NAME
Optional.

The name of the zone.

Currently, this definition is not used. Typically, it is provided to identify the field.

TYPE
Optional.

Defines the type of the zone.

If you do not specify the type, it gets the default value of the corresponding type of a table column.

Valid values are as follows:

- CHAR
- VARCHAR
- NCHAR
- NVARCHAR
- SMALLINT
- BIGINT
- BINARY
- VARBINARY
- GRAPHIC
- VARGRAPHIC
- FLOAT
- DOUBLE
- DEC, NUM, or NUMERIC
- DECFLOAT
- BOOLEAN
- DATE
- TIME
- TIMESTAMP

STYLE

Optional.

Defines the zone representation.

The default representation is based on zone type and format option.

All other styles are valid only for their corresponding non-textual zone types.

Valid values are as follows:

- INTERNAL

  Valid only for textual zones, that is, char, varchar, nchar, and nvarchar.
- DECIMAL

  Valid for integer and numeric zone types.
- DECIMALDELIM <'decimal-delim'>

  Valid for numeric, float, double, and time style (time and timestamp) zone types.
- FLOATING

  Valid for float or double zone types.
- EXPONENTIAL

  Valid for float or double zone types.
- YMD <'date-delim'>

  Valid for date zones, including other date styles that are supported for the DateStyle and DateDelim external table options.
- 12Hour <'time-delim'>

Valid for time zones, including other time styles that are supported for the TimeStyle and TimeDelim external table options.

- 24Hour <'time-delim'>

  Valid for time zones, including other time styles that are supported for the TimeStyle and TimeDelim external table options.

- YMD <'date-delim'> 24Hour <'time-delim'>

  Valid for timestamp zones, including other combinations of date and time styles that are supported for the DateStyle, DateDelim, TimeStyle, and TimeDelim external table options.

- TRUE_FALSE, Y_N, 1_0

  Valid for boolean zones, including other boolean styles that are supported for the BoolStyle external table option. The style must be in accordance with the format.

LENGTH

Optional.

Specified as bytes or characters followed by the number or the internal reference to the reference zone.

Number of bytes or characters as provided or as referenced by the reference zone.

For reference zones or filler zones, you cannot use internal references. For reference zones, the number of bytes specifies how the data is read from the data file to get the referred value.

You can use plus signs and minus signs as follows:

```
BYTES @2 + 10
BYTES @2 - 10
```

NULLIF

Optional.

Definition of the zone NULLESS attribute.

Specifies a known data pattern within the field that, when it is present, signifies that the field is null.

The length is equal to or less than the column width. The maximum length is 39 bytes.

You can use the following types of references:

@

Internal reference to numeric zones.

Exact match of the numeric value.

&

External reference.

Exact match of the specified value.

&&

Isolated reference.

Leading spaces and trailing spaces are to be skipped with the exact string match.

Nulls are detailed in the following examples:

Table 110. Layout example

Use type	Name	Type	Style	Length	Nullif
NA	f1	int4	DECIMAL	Bytes 10	Nullif & = 0

Table 110. Layout example (continued)					
Use type	**Name**	**Type**	**Style**	**Length**	**Nullif**
NA	f2	date	YMD	Bytes 10	Nullif &= '2000-10-10'
NA	f3	char(20)	INTERNAL	Chars 10	Nullif &&='ab'
Filler	f4	char(10)	NA	Bytes 10	NA

Remember:

- The referred zone in a length clause must be of type integer.
- You must not specify the NULLIF option for reference zones or filler zones.
- Reference zones and filler zones cannot have variable lengths.
- Variable length zones cannot refer themselves.
- Define the referred zone in a length clause as REF.
- Length-clause references can use only the INTERNAL (@) reference. External or isolated references are not supported.
- Between the referred zone of a length clause and the zone itself, reference zones are not allowed.
- If the reference type is INTERNAL (@), the NULLIF clause cannot refer to itself.
- If the column is non-nullable, it may not have the NULLIF clause.
- Variable length is allowed only for the string type of zones.
- The NULLIF clause can refer only to REF zones or the zones themselves.
- Between the zone that is referred by the NULLIF clause and the zone itself, other referred zones are not allowed, except for the zone that is referred in the length clause.
- The record length can point to zone 1 only for reference.
- A REF must have a zone that refers it.
- The NULLIF clause can have external references only if the REF zone is non-integer.

Recordlength
Specifies the length of the entire record, where null-indicator bytes are included if they exist, and the record delimiter is excluded if it exists.

The value is a constant integer.

The value can also be an internal reference to the reference zone in the layout definition.

There is no default value.

You can use plus signs and minus signs for an internal reference as follows:

```
RECORDLENGTH @1 + 10
RECORDLENGTH @1 - 10
```

IGNOREZERO or TRIM_NULLS
Specifies whether the binary value zero in CHAR fields and VARCHAR fields is to be discarded.

TRUE or ON
The byte value zero is ignored.

FALSE or OFF
The byte value zero is not ignored. This is the default.

KEEP
The binary value zero is accepted and allowed as part of the input field.

INCLUDEHEADER or COLUMN_NAMES
For an unload operation, whether the table column names are to be included as headers in the external-table file:

TRUE or ON
The table column names are to be included as headers.

FALSE or OFF
The table column names are not to be included as headers. This is the default.

INCLUDEZEROSECONDS
For an unload operation, whether to specify **00** as the value for seconds when no value for seconds is available:

TRUE or ON
Specify **00** as the value for seconds.

FALSE or OFF
Do not specify a value for seconds. This is the default.

INCLUDEHIDDEN
For a load operation, specify whether hidden column values are present in a data file.

The `INCLUDEHIDDEN` option works when you are creating an external table by using the LIKE or SAMEAS clause, and base table has hidden columns.

TRUE
A data file contains values against hidden column.

FALSE
A data file does not contain values against hidden column. This is the default. You can change the default value by using the registry variable `DB2_EXTBL_INCLUDE_HIDDEN_COLS`.

LFINSTRING
Specifies how to interpret an unescaped line-feed (sometimes called an LF or newline) character within string data:

TRUE or ON
An unescaped LF character is interpreted as a record delimiter only if it is in the last field of a record; otherwise, it is treated as data. To cause an LF character that is in the last field of a record to be treated as data, enclose the value of that field in single or double quotation marks.

FALSE or OFF
An unescaped LF character is interpreted as a record delimiter regardless of its position. This is the default.

This option is not supported for unload operations.

Attention: This SQL compatibility enhancement is only available in Db2 Version 11.5 Mod Pack 2 and later versions.

LOGDIR or ERROR_LOG
The directory to which the following files are written:

<database>.<schema>.<external-table-name>.<file-name>.<application-handle>.<id>.bad
A file containing rejected records (that is, records that could not be processed).

<database>.<schema>.<external-table-name>.<file-name>.<application-handle>.<id>.log
A log file.

The default is the directory to which the external-table file is written. If the length of the name that is constructed for a .bad or .log file would exceed the allowed maximum, the name of the file that contains the external table (indicated by `<file-name>`) is truncated so that the maximum is not exceeded.

If a .log or .bad file is generated while carrying out an operation on a partition, the name of the generated file is suffixed with a period followed by the 3-digit partition number.

MAXERRORS or MAX_ERRORS
For a load operation, the threshold for the number of rejected records at which the system stops processing and immediately rolls back the load. The default is 1 (that is, a single rejected record results in a rollback).

For fixed-length format, the following conditions apply:

- The parser reports errors for each field or zone rather than one error for the row.
- Multiple errors can be reported for the same row.
- When the parser detects an error in a field or zone, it recovers by using the field length or zone length. It then continues from the next field or zone until the end of record is reached, or an unrecoverable error occurs, or the MaxErrors limit is reached.
- Unrecoverable errors include the following errors:
  - RecordLength mismatch.
  - RecordDelimiter is not found.
  - The RecordLength value is not valid, that is, the value is a negative value or zero.
  - The zone length is not valid, that is, the value is a negative value.
  - The UTF-8 initial byte is not valid.
  - The UTF-8 continuation bytes are not valid.

MULTIPARTSIZEMB

When the DB2_ENABLE_COS_SDK registry variable is set to ON, Db2 remote storage communication with cloud object storage is facilitated through an embedded vendor COS SDK which allows Db2 to stream objects/files to cloud object storage in multiple parts (aka 'multipart upload'). This parameter specifies the part size for multipart upload, in megabytes (MB), for the file being unloaded, and overrides the value specified in the MULTIPARTSIZEMB dbm config parameter. This option is available starting in Version 11.5 Modification Pack 7, in Linux (x86) environments only.

MAXROWS or MAX_ROWS
If set to a positive integer, this specifies the maximum number of records (rows) in the external table that are to be processed. If set to 0 (the default), there is no limit and all rows are processed. During a load operation, if MAXROWS is set to a positive value, after that number of rows are processed, regardless of whether some of the rows were rejected or skipped, the system ends the load operation and commits all inserted records.

MERIDIANDELIM
A single-byte character that separates the seconds component from the AM token or PM token in the 12-hour delimited and undelimited formats of a time value.

The default delimiter is a space (`' '`).

Between the seconds component and the AM token or PM token, a delimiter is not required. For example, both of the following values are valid:

```
1:02:46.12345 AM
1:02:46.12345AM
```

NOLOG
Specifies whether the `.log` file for the external table is created.

This option does not apply to `.bad` files.

Possible values are:

TRUE
No `.log` file is created.

FALSE
The `.log` file is created.

This is the default.

NULLVALUE or NULL_VALUE
The UTF-8 string of at most 4 bytes that is to be used to indicate a null value. The default is `'NULL'`.

PARTITION
If the Database Partitioning Feature (DPF) is enabled for the database, an external table can be partitioned into several files. The name of each of the data files that comprise an external table are suffixed with a period followed by a 3-digit number from 000 to 999 that indicates the number of the partition. For example, if an external table with the name **dataFile.txt** is divided into three partitions, the files that comprise it have the names **dataFile.txt.000**, **dataFile.txt.001**, and **dataFile.txt.002**. These files must be accessible from all members.

For a partitioned external table, the PARTITION option specifies to which partition or partitions the statement applies:

PARTITION ALL
The statement applies to all of the partitions that comprise the external table. For an unload operation, this is the only value that is allowed.

PARTITION (*n* TO *n*)
The statement applies to all of the partitions in the specified range, for example, **PARTITION (54 TO 62)**.

PARTITION (*n*,*n*,...)
The statement applies only to the specified partition or partitions, for example, **PARTITION (53)** or **PARTITION (51,57,58)**. If more than one partition number is specified, they must be in ascending order (sqlcode SQL0263N with SQLSTATE=42615) and there can be no duplicates (sqlcode SQL0265N with SQLSTATE=42615).

If a .log or .bad file is generated while carrying out an operation on a partitioned external table, the name of the generated file is suffixed with a period followed by the 3-digit partition number.

If the DPF is enabled and the PARTITION option is not specified, the external table is treated as single-partitioned table on the coordinator member. The names of the external table file and the .log and .bad files are not suffixed with a partition number.

If the DPF is not enabled, the PARTITION option can be specified, but only with the value ALL, (0 to 0), or (0) (SQL0644N). It will have no effect.

The REMOTESOURCE and PARTITION options are mutually exclusive.

QUOTEDNULL
For a load operation, how to interpret a value that is enclosed in single or double quotation marks and that matches the null value specified by the NULLVALUE or NULL_VALUE option (for example, "NULL" or 'NULL'):

TRUE or ON
The value is interpreted as a null value. This is the default.

FALSE or OFF
The value is interpreted as a character string.

QUOTEDVALUE or STRING_DELIMITER
Whether data values are enclosed in quotation marks:

SINGLE or YES
Data values are enclosed in single quotation marks (').

DOUBLE
Data values are enclosed in double quotation marks (").

NO
Data values are not enclosed in quotation marks. This is the default.

RECORDDELIM or RECORD_DELIMITER
The string literal that is to be interpreted as a row (record) delimiter. The default is `'\n'`.

When CRINSTRING is set to TRUE, RECORDDELIM cannot contain a CR ('\r') character - with the sole exception of a CRLF ('\r\n') delimiter allowed with CRINSTRING for text format only.

REMOTESOURCE
Where the external-table file resides and, if it resides on a remote system, whether the file data is to be compressed:

LOCAL
The file resides on the local server, that is, the system that hosts the database. This is the default.

YES
The file resides on a system other than the local server. For example, specify YES if a client system is connected to the database and the file resides on that system. File data is not compressed before it is transferred.

GZIP
Similar to YES, except that the file data is compressed using the GZIP compression algorithm before the data is transferred, and is decompressed after it is received. This improves overall performance when a large amount of compressible data is being transferred.

LZ4
Similar to YES, except that the file data is compressed using the LZ4 compression algorithm before the data is transferred, and is decompressed after it is received. This improves overall performance when a large amount of compressible data is being transferred.

The REMOTESOURCE, SWIFT, and S3 options are mutually exclusive.

Attention: In the container-only release of Db2 Version 11.5 Mod Pack 1 or later versions, AZURE is compatible and its options are mutually exclusive with REMOTESOURCE, SWIFT, and S3.

The REMOTESOURCE and PARTITION options are mutually exclusive. The COMPRESS option cannot be specified if the value of the REMOTESOURCE option is GZIP or LZ4.

REQUIREQUOTES
Whether quotation marks are mandatory:

TRUE or ON
Quotation marks are mandatory. The QUOTEDVALUE option must be set to YES, SINGLE, or DOUBLE.

FALSE or OFF
Quotation marks are not mandatory. This is the default.

SKIPROWS or SKIP_ROWS
For a load operation, the number of rows to skip before beginning to load the data. The default is 0. Because skipped rows are processed before they are skipped, a skipped row is still capable of causing a processing error.

SOCKETBUFSIZE
The size, in bytes, of the chunks of data that are read from the source file. Valid values range from 64 KB - 800 MB. If you specify a value outside this range, the value is set to the nearest valid value. The default is 8 MB.

STRICTNUMERIC
For a load operation, how to treat a value that is to be inserted into a DECIMAL field when its scale exceeds that defined for the field:

TRUE or ON
The row containing the value to be inserted is rejected. For example, if any of the following values were to be loaded into a DECIMAL(5,3) field, the row containing that value would be rejected:

```
12.666666666
-98.34496862785
0.00089
```

FALSE or OFF
The row containing the value to be inserted is accepted, and the portion of the decimal fraction that exceeds the scale defined for the field is truncated. This is the default. For example, the values in the previous example would be converted to:

```
12.666
-98.344
0.000
```

SWIFT
Specifies that the source data file is located in a Swift object store. The REMOTESOURCE, SWIFT, and S3 options are mutually exclusive. Use the DATAOBJECT option to specify the file name.

Attention: In the container-only release of Db2 Version 11.5 Mod Pack 1 or later versions, AZURE is compatible and its options are mutually exclusive with REMOTESOURCE, SWIFT, and S3.

Syntax:

```
SWIFT (endpoint, authKey1, authKey2, bucket)
```

where:

endpoint
A character string that specifies the URL of the SWIFT web service.

authKey1
A character string that specifies the access ID or username of the Swift open stack account used to validate the user.

authKey2
A character string that specifies the password of the Swift open stack account used to validate the user.

bucket
The name of the Swift open stack container (bucket) in which the file resides.

Example:

```
CREATE EXTERNAL TABLE exttab1(a int) using
  (dataobject 'datafile1.dat'
   swift('https://dal05.objectstorage.softlayer.net/auth/v1.0/',
    'XXXOS123456-2:xxx123456',
    'b207c6e974020737d92174esdf6d5be9382aa4c335945a14eaa9172c70f8df16',
    'my_dev'
   )
  )
```

S3
Specifies that the source data file is located in an S3 compatible object store. The REMOTESOURCE, SWIFT, and S3 options are mutually exclusive. Use the DATAOBJECT option to specify the file name.

Attention: In the container-only release of Db2 Version 11.5 Mod Pack 1 or later versions, AZURE is compatible and its options are mutually exclusive with REMOTESOURCE, SWIFT, and S3.

Syntax:

```
S3 (endpoint, authKey1, authKey2, bucket)
```

where:

endpoint
A character string that specifies the URL of the S3 compatible web service.

authKey1
A character string that specifies the S3 access key ID of the access keys used to validate the user and all user actions. For IBM Cloud Object Storage, this is the access key ID from the HMAC credentials.

authKey2
A character string that specifies the S3 secret key of the access keys that are used to validate the user and all user actions. For IBM Cloud Object Storage, this is the secret access key from the HMAC credentials.

bucket
The name of the S3 bucket in which the file resides.

Note: For IBM Cloud Object Storage, to create HMAC credentials, when creating new service credentials, specify `{"HMAC:true}` in the **Add Inline Configuration Parameters** field.

Example using AWS S3:

```
CREATE EXTERNAL TABLE exttab2(a int) using
  (dataobject 'datafile2.dat'
   s3('s3.amazonaws.com',
    'XXXOS123456-2:xxx123456',
    'bs07c6e974040737d92174e5e96d5be9382aa4c33xxx5a14eaa9172c70f8df16',
    'my_dev'
   )
  )
```

Example using IBM Cloud Object Storage:

```
CREATE EXTERNAL TABLE exttab2(a int) using
  (dataobject 'datafile2.dat'
   s3('s3-api.us-geo.objectstorage.softlayer.net',
    '1a2bkXXXsaddntLo0xX0',
    'XXxxiEPjJ7T7WBUz74E6abcdABCDE8Q7RgU4gYY9',
    'my_dev'
   )
  )
```

AZURE

Attention: This feature is available in the container-only release of Db2 Version 11.5 Mod Pack 1 or later versions.

Specifies that the source data file is located in Microsoft Azure Blob Storage. The REMOTESOURCE, SWIFT, S3, and AZURE options are mutually exclusive. Use the DATAOBJECT option to specify the file name. Syntax:

Syntax:

```
AZURE (endpoint, authKey1, authKey2, bucket)
```

where:

endpoint
A character string that specifies the URL of the AZURE web service.

authKey1
A character string that specifies the access ID or username of the Azure Blob Storage account used to validate the user.

authKey2
A character string that specifies the access key of the Azure Blob Storage account used to validate the user.

bucket
The name of the Azure Blob Storage container (bucket) in which the file resides.

Example:

```
CREATE EXTERNAL TABLE exttab1(a int) using
  (dataobject 'datafile1.dat'
   azure('https://my_account.blob.core.windows.net',
    'my_account',
    'lW+oHjmZecPS++IKgThAHlMUOaFUA5C6Z2RlFmc9JPpK34RO/ZIOywzILxJnzGPHz6d/
yDrcQDAwH5wySbOZMQ==',
    'my_bucket'
   )
  )
```

Example using IBM Cloud Object Storage:

```
CREATE EXTERNAL TABLE exttab2(a int) using
  (dataobject 'datafile2.dat'
   s3('s3-api.us-geo.objectstorage.softlayer.net',
    '1a2bkXXXsaddntLo0xX0',
    'XXxxiEPjJ7T7WBUz74E6abcdABCDE8Q7RgU4gYY9',
    'my_dev'
   )
  )
```

TIMEDELIM
The single-byte character that is to separate time components (hours, minutes, and seconds). The default is ' : '. If TIMEDELIM is set to an empty string, hours, minutes, and seconds must all be specified as two-digit numbers. The TIMESTAMP_FORMAT and TIMEDELIM options are mutually exclusive.

TIMEROUNDNANOS or TIMEEXTRAZEROS

Note: This option applies only to TIMESTAMP columns.

Specifies whether records that contain time values whose non-zero precision exceeds six decimal places are to be accepted (and rounded to the nearest microsecond) or rejected:

TRUE
All records are accepted. Their time values are rounded to the nearest microsecond.

FALSE
Only those records that can be stored without a loss of precision (for example, '08.15.32.123' or '08.15.32.12345600000', but not '08.15.32.1234567') are accepted. All other records are rejected. This is the default.

TIMESTYLE
The time format that is to be used in the data file:

24HOUR
24-hour format, for example 23:55. This is the default.

12HOUR
12-hour format, for example 11:55 PM. An AM or PM token can be preceded by a single space and is not case-sensitive.

The TIMESTYLE option and the TIME_FORMAT or TIMESTAMP_FORMAT option are mutually exclusive.

TIMESTAMP_FORMAT
The format of the timestamp field in the data file. The value can be any of the format strings that are accepted by the “TIMESTAMP_FORMAT ” on page 527. The default is **'YYYY-MM-DD HH.MI.SS'**. The TIMESTAMP_FORMAT option and the TIMEDELIM, DATEDELIM, TIMESTYLE, or DATESTYLE option are mutually exclusive.

TIME_FORMAT
The format of the time field in the data file. The value can be any of the time format strings that are accepted by the “TIMESTAMP_FORMAT ” on page 527. The default is **HH.MI.SS**. The TIME_FORMAT option and a TIMEDELIM or TIMESTYLE option are mutually exclusive.

TRIMBLANKS
How an external table is to treat leading or trailing blanks (that is, leading or trailing space characters) in a string:

LEADING
All leading blanks (that is, blanks that precede the first non-blank character) are removed.

TRAILING
All trailing blanks (that is, blanks that follow the last non-blank character) are removed.

BOTH
All leading and trailing blanks are removed.

NONE
No blanks are removed. This is the default.

When reading data from a file and loading it into an external table:

- If QUOTEDVALUE or STRING_DELIMITER is specified with the values SINGLE, YES, or DOUBLE, leading and trailing blanks within quotation marks are not removed.
- For CHAR and NCHAR data, the values TRAILING or BOTH will not have any effect on trailing blanks, because the string will automatically be re-padded with trailing blanks.

TRUNCSTRING or TRUNCATE_STRING
How the system processes a CHAR or VARCHAR string that exceeds its declared storage size:

TRUE
The system truncates a string value that exceeds its declared storage size.

FALSE
The system returns an error when a string value exceeds its declared storage size. This is the default.

Y2BASE
The year that is the beginning of the 100-year range. Years that are specified as 2 digits are counted from this year. The default is 2000. This option must be specified when DATESTYLE is set to Y2MD, MDY2, DMY2, MONDY2 or DMONY2.

Table 111. Options

Option	Default	Applies to Load	Applies to Unload
Azure **Attention:** This option only applies to the container-only release of Db2 Version 11.5 Mod Pack 1 or later versions.	(no default)	Y	Y
BOOLSTYLE or BOOLEAN_STYLE	1_0	Y	Y
CARDINALITY	(no default)	Y	Y
CCSID	(no default)	Y	Y
COMPRESS	NO	Y	Y
CRINSTRING	FALSE	Y	Y
CTRLCHARS	FALSE	Y	N
DATAOBJECT or FILE_NAME	(no default)	Y	Y
DATEDELIM	'-'	Y	Y
DATETIMEDELIM	A space (' ')	Y	Y
DATESTYLE	YMD	Y	Y
DATE_FORMAT	YYYY-MM-DD	Y	Y

Table 111. Options (continued)

Option	Default	Applies to Load	Applies to Unload
DECIMALDELIM or DECIMAL_CHARACTER	'.'	Y	Y
DELIMITER	'|'	Y	Y
ENCODING	INTERNAL	Y	Y[1]
ESCAPECHAR or ESCAPE_CHARACTER	(no default)	Y	Y
FILLRECORD	FALSE	Y	N
FORMAT or FILE_FORMAT	TEXT	Y	Y
IGNOREZERO or TRIM_NULLS	FALSE	Y	N
INCLUDEHEADER or COLUMN_NAMES	FALSE	N	Y
INCLUDEZEROSECONDS	FALSE	Y	Y
INCLUDEHIDDEN	FALSE	Y	N
LFINSTRING	FALSE	Y	N
LOGDIR or ERROR_LOG	target directory of external-table file	Y	N
MULTIPARTSIZEMB	value specified by the MULTIPARTSIZEMB dbm config parameter.	Y	N
MAXERRORS or MAX_ERRORS	1	Y	N
MAXROWS or MAX_ROWS	0	Y	N
MERIDIANDELIM	A space (' ')	Y	Y
NOLOG	FALSE	Y	Y
NULLVALUE or NULL_VALUE	'NULL'	Y	Y
PARTITION	(no default)	Y	Y
QUOTEDNULL	TRUE	Y	N
QUOTEDVALUE	NO	Y	N
RECORDDELIM or RECORD_DELIMITER	'\n'	Y	N
REMOTESOURCE	LOCAL	Y	Y
REQUIREQUOTES	FALSE	Y	N
SKIPROWS or SKIP_ROWS	0	Y	N
SOCKETBUFSIZE	8 MB	Y	Y
STRICTNUMERIC	FALSE	Y	N
SWIFT	(no default)	Y	Y
S3	(no default)	Y	Y
TIMEDELIM	':'	Y	Y

Table 111. Options (continued)			
Option	**Default**	**Applies to Load**	**Applies to Unload**
TIMEROUNDNANOS or TIMEEXTRAZEROS	FALSE	Y	N
TIMESTAMP_FORMAT	'YYYY-MM-DD HH.MI.SS'	Y	Y
TIMESTYLE	24HOUR	Y	Y
TIME_FORMAT	HH.MI.SS	Y	Y
TRIMBLANKS	NONE	Y	Y
TRUNCSTRING or TRUNCATE_STRING	FALSE	Y	N
Y2BASE	2000	Y	N
[1] Only for the values INTERNAL, UTF8, and LATIN9.			

joined-table

A *joined-table* specifies an intermediate result set that is the result of one or more join operations. For more information, see "joined-table" on page 685.

period-specification

A *period-specification* identifies an intermediate result table consisting of the rows of the referenced table where the period matches the specification. A *period-specification* can be specified following the name of a temporal table or the name of a view. The same period name must not be specified more than once for the same table reference (SQLSTATE 428HY). The rows of the table reference are derived by application of the period specifications.

If the table is a system-period temporal table and a *period-specification* for the SYSTEM_TIME period is not specified, the table reference includes all current rows and does not include any historical rows of the table. If the table is an application-period temporal table and a *period-specification* for the BUSINESS_TIME period is not specified, the table reference includes all rows of the table. If the table is a bitemporal table and a *period-specification* is not specified for both SYSTEM_TIME and BUSINESS_TIME, the table reference includes all current rows of the table and does not include any historical rows of the table.

If the table reference is a single-view-reference, the rows of the view reference are derived by application of the period specifications to all of the temporal tables accessed when computing the result table of the view. If the view does not access any temporal table, then the *period-specification* has no effect on the result table of the view. If *period-specification* is used, the view definition or any view definitions referenced when computing the result table of the view must not include any references to compiled SQL functions or external functions with a data access indication other than NO SQL (SQLSTATE 428HY).

If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a value other than the null value, then a *period-specification* that references SYSTEM_TIME must not be specified for the table reference or view reference, unless the value in effect for the SYSTIMESENSITIVE bind option is NO (SQLSTATE 428HY).

If the CURRENT TEMPORAL BUSINESS_TIME special register is set to a value other than the null value, then a period specification that references BUSINESS_TIME must not be specified for the table reference or view reference, unless the value in effect for the BUSTIMESENSITIVE bind option is NO (SQLSTATE 428HY).

FOR SYSTEM_TIME

Specifies that the SYSTEM_TIME period is used for the *period-specification*. If the clause is specified following a *table-name*, the table must be a system-period temporal table (SQLSTATE 428HY). FOR SYSTEM_TIME must not be specified if the value of the CURRENT TEMPORAL SYSTEM_TIME special register is not the null value and the SYSTIMESENSITIVE bind option is set to YES (SQLSTATE 428HY).

FOR BUSINESS_TIME

Specifies that the BUSINESS_TIME period is used for the *period-specification*. If the clause is specified following a *table-name*, BUSINESS_TIME must be a period defined in the table (SQLSTATE 4274M). FOR BUSINESS_TIME must not be specified if the value of the CURRENT TEMPORAL BUSINESS_TIME special register is not the null value and the BUSTIMESENSITIVE bind option is set to YES (SQLSTATE 428HY).

value*, *value1*, and *value2

The *value*, *value1*, and *value2* expressions return the null value or a value of one of the following built-in data types (SQLSTATE 428HY): a DATE, a TIMESTAMP, or a character string that is not a CLOB or DBCLOB. If the argument is a character string, it must be a valid character string representation of a timestamp or a date (SQLSTATE 22007). For the valid formats of string representations of timestamp values, see the section "String representations of datetime values" in the topic Datetime values.

Each expression can contain any of the following supported operands (SQLSTATE 428HY):

- Constant
- Special register
- Variable. For details, refer to “References to variables” on page 19.
- Parameter marker
- Scalar function whose arguments are supported operands (user-defined functions and non-deterministic functions cannot be used)
- CAST specification where the cast operand is a supported operand
- Expression using arithmetic operators and operands

AS OF *value*

Specifies that the table reference includes each row for which the value of the begin column for the specified period is less than or equal to *value*, and the value of the end column for the period is greater than *value*. If *value* is the null value, the table reference is an empty table.

Example: The following query returns the insurance coverage information for insurance policy number 100 on August 31, 2010.

```
SELECT coverage FROM policy_info FOR BUSINESS_TIME
        AS OF '2010-08-31' WHERE policy_id = '100'
```

FROM *value1* TO *value2*

Specifies that the table reference includes rows that exist for the period specified from *value1* to *value2*. A row is included in the table reference if the value of the begin column for the specified period in the row is less than *value2*, and the value of the end column for the specified period in the row is greater than *value1*. The table reference contains zero rows if *value1* is greater than or equal to *value2*. If *value1* or *value2* is the null value, the table reference is an empty table.

Example: The following query returns the insurance coverage information for insurance policy 100, during the year 2009 (from January 1, 2009 at 12:00 AM until before January 1, 2010).

```
SELECT coverage FROM policy_info FOR BUSINESS_TIME
        FROM '2009-01-01' TO '2010-01-01' WHERE policy_id = '100'
```

BETWEEN *value1* AND *value2*

Specifies that the table reference includes rows in which the specified period overlaps at any point in time between *value1* and *value2*. A row is included in the table reference if the value of the begin column for the specified period in the row is less than or equal to *value2* and the value of the end column for the specified period in the row is greater than *value1*. The table reference contains zero rows if *value1* is greater than *value2*. If *value1* is equal to *value2*, the expression is equivalent to AS OF *value1*. If *value1* or *value2* is the null value, the table reference is an empty table.

Example: The following query returns the insurance coverage information for insurance policy number 100, during the year 2008 (between January 1, 2008 and December 31, 2008 inclusive).

```
SELECT coverage FROM policy_info FOR BUSINESS_TIME
        BETWEEN '2008-01-01' AND '2008-12-31' WHERE policy_id = '100'
```

Following are syntax alternatives for *period-specification* clauses:

- AS OF TIMESTAMP can be specified in place of FOR SYSTEM_TIME AS OF
- VERSIONS BETWEEN TIMESTAMP can be specified in place of FOR SYSTEM_TIME BETWEEN

correlation-clause

The exposed names of all table references must be unique. An exposed name is:

- A *correlation-name*
- A *table-name* that is not followed by a *correlation-name*
- A *view-name* that is not followed by a *correlation-name*
- A *nickname* that is not followed by a *correlation-name*
- An *alias-name* that is not followed by a *correlation-name*

If a *correlation-clause* clause does not follow a *function-name* reference, *xmltable-expression* expression, nested table expression, or *data-change-table-reference* reference, or if a *typed-correlation-clause* clause does not follow a *function-name* reference, then there is no exposed name for that table reference.

Each *correlation-name* is defined as a designator of the immediately preceding *table-name*, *view-name*, *nickname*, *function-name* reference, *xmltable-expression*, nested table expression, or *data-change-table-reference*. Any qualified reference to a column must use the exposed name. If the same table name, view, or nickname is specified twice, at least one specification must be followed by a *correlation-name*. The *correlation-name* is used to qualify references to the columns of the table, view or nickname. When a *correlation-name* is specified, *column-name*s can also be specified to give names to the columns of the table reference. If the *correlation-clause* does not include *column-name*s, the exposed column names are determined as follows:

- Column names of the referenced table, view, or nickname when the *table-reference* is a *table-name*, *view-name*, *nickname*, or *alias-name*
- Column names specified in the RETURNS clause of the CREATE FUNCTION statement when the *table-reference* is a *function-name* reference
- Column names specified in the COLUMNS clause of the *xmltable-expression* when the *table-reference* is an *xmltable-expression*
- Column names exposed by the fullselect when the *table-reference* is a *nested-table-expression*
- Column names from the target table of the data change statement, along with any defined INCLUDE columns when the *table-reference* is a *data-change-table-reference*

tablesample-clause

The optional *tablesample-clause* can be used to obtain a random subset (a sample) of the rows from the specified *table-name*, rather than the entire contents of that *table-name*, for this query. This sampling is in addition to any predicates that are specified in the *where-clause*. Unless the optional REPEATABLE clause is specified, each execution of the query will usually yield a different sample, except in degenerate cases where the table is so small relative to the sample size that any

sample must return the same rows. The size of the sample is controlled by the *numeric-expression1* in parentheses, representing an approximate percentage (P) of the table to be returned.

TABLESAMPLE

The method by which the sample is obtained is specified after the TABLESAMPLE keyword, and can be either BERNOULLI or SYSTEM. The exact number of rows in the sample might be different for each execution of the query, but on average is approximately P percent of the table, before any predicates further reduce the number of rows.

The *table-name* must be a stored table. It can be a materialized query table (MQT) name, but not a subselect or table expression for which an MQT has been defined, because there is no guarantee that the database manager will route to the MQT for that subselect.

Semantically, sampling of a table occurs before any other query processing, such as applying predicates or performing joins. Repeated accesses of a sampled table within a single execution of a query (such as in a nested-loop join or a correlated subquery) will return the same sample. More than one table can be sampled in a query.

BERNOULLI

BERNOULLI sampling considers each row individually. It includes each row in the sample with probability P/100 (where P is the value of *numeric-expression1*), and excludes each row with probability 1 - P/100, independently of the other rows. So if the *numeric-expression1* evaluated to the value 10, representing a ten percent sample, each row would be included with probability 0.1, and excluded with probability 0.9.

SYSTEM

SYSTEM sampling permits the database manager to determine the most efficient manner in which to perform the sampling. In most cases, SYSTEM sampling applied to a *table-name* means that each page of *table-name* is included in the sample with probability P/100, and excluded with probability 1 - P/100. All rows on each page that is included qualify for the sample. SYSTEM sampling of a *table-name* generally executes much faster than BERNOULLI sampling, because fewer data pages are retrieved. However, SYSTEM sampling can often yield less accurate estimates for aggregate functions, such as SUM(SALES), especially if the rows of *table-name* are clustered on any columns referenced in that query. The optimizer might in certain circumstances decide that it is more efficient to perform SYSTEM sampling as if it were BERNOULLI sampling. An example is when a predicate on *table-name* can be applied by an index and is much more selective than the sampling rate P.

numeric-expression1

The *numeric-expression1* specifies the size of the sample to be obtained from *table-name*, expressed as a percentage. It must be a constant numeric expression that cannot contain columns. The expression must evaluate to a positive number that is less than or equal to 100, but can be between 1 and 0. For example, a value of 0.01 represents one one-hundredth of a percent, meaning that 1 row in 10 000 is sampled, on average. A *numeric-expression1* that evaluates to 100 is handled as if the tablesample-clause were not specified. If *numeric-expression1* evaluates to the null value, or to a value that is greater than 100 or less than 0, an error is returned (SQLSTATE 2202H).

REPEATABLE (*numeric-expression2*)

It is sometimes desirable for sampling to be repeatable from one execution of the query to the next; for example, during regression testing or query debugging. This can be accomplished by specifying the REPEATABLE clause. The REPEATABLE clause requires the specification of a *numeric-expression2* in parentheses, which serves the same role as the seed in a random number generator. Adding the REPEATABLE clause to the tablesample-clause of any *table-name* ensures that repeated executions of that query (using the same value for *numeric-expression2*) return the same sample, assuming that the data itself has not been updated, reorganized, or repartitioned. To guarantee that the same sample of *table-name* is used across multiple queries, use of a global temporary table is recommended. Alternatively, the multiple queries can be combined into one query, with multiple references to a sample that is defined using the WITH clause.

Examples:

1. Request a 10% Bernoulli sample of the Sales table for auditing purposes.

```
SELECT * FROM Sales
  TABLESAMPLE BERNOULLI(10)
```

2. Compute the total sales revenue in the Northeast region for each product category, using a random 1% SYSTEM sample of the Sales table. The semantics of SUM are for the sample itself, so to extrapolate the sales to the entire Sales table, the query must divide that SUM by the sampling rate (0.01).

```
SELECT SUM(Sales.Revenue) / (0.01)
  FROM Sales TABLESAMPLE SYSTEM(1)
  WHERE Sales.RegionName = 'Northeast'
  GROUP BY Sales.ProductCategory
```

3. Using the REPEATABLE clause, modify the previous query to ensure that the same (yet random) result is obtained each time the query is executed. The value of the constant enclosed by parentheses is arbitrary.

```
SELECT SUM(Sales.Revenue) / (0.01)
  FROM Sales TABLESAMPLE SYSTEM(1) REPEATABLE(3578231)
  WHERE Sales.RegionName = 'Northeast'
  GROUP BY Sales.ProductCategory
```

table-UDF-cardinality-clause

The *table-UDF-cardinality* clause can be specified for each user-defined table function reference within the FROM clause. This option indicates the expected number of rows to be returned only for the SELECT statement that contains it. The CARDINALITY and CARDINALITY MULTIPLIER clauses are not allowed if the table function is an inlined SQL table function (SQLSTATE 42887).

CARDINALITY *integer-constant*

Specifies an estimate of the expected number of rows that are returned by the reference to the user-defined function. The value range of *integer-constant* is from 0 to 9 223 372 036 854 775 807 inclusive.

CARDINALITY MULTIPLIER *numeric-constant*

The product of the specified CARDINALITY MULTIPLIER *numeric-constant* and the reference cardinality value are used by the database server as the expected number of rows that are returned by the table function reference.

In this case, *numeric-constant* can be in the integer, decimal, or floating-point format. The value must be greater than or equal to zero. If the decimal number notation is used, the number of digits can be up to 31. An integer value is treated as a decimal number with no fraction. If zero is specified or the computed cardinality is less than 1, the cardinality of the reference to the user-defined table function is assumed to be 1.

The value in the CARDINALITY column of SYSSTAT.ROUTINES for the table function name is used as the reference cardinality value. If no value is set in the CARDINALITY column of SYSSTAT.ROUTINES, a finite value is assumed as its default value for the reference cardinality value. This finite value is the same value that is assumed for tables for which the RUNSTATS utility has not gathered statistics.

Only a numeric constant can follow the keyword CARDINALITY or CARDINALITY MULTIPLIER. A host variable or parameter marker is not supported. Specifying a cardinality value in a table function reference does not change the CARDINALITY column value for the function in the SYSSTAT.ROUTINES catalog view.

The CARDINALITY value for external and compiled SQL user-defined table functions can be changed by updating the CARDINALITY column in the SYSSTAT.ROUTINES catalog view. The CARDINALITY value for an external table function can also be initialized by specifying the CARDINALITY option in the CREATE FUNCTION (external table) statement when a user-defined table function is created.

typed-correlation-clause

A *typed-correlation-clause* clause defines the appearance and contents of the table generated by a generic table function. This clause must be specified when the table-function-references is a generic table function and cannot be specified for any other table reference. The following *data-type* values are supported in generic table functions:

Table 112. Data types supported in generic table functions

SQL column data type	Equivalent Java data type
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
DOUBLE	double
DECIMAL(p,s)	java.math.BigDecimal
NUMERIC(p,s)	java.math.BigDecimal
CHAR(n)	java.lang.String
CHAR(n) FOR BIT DATA	COM.ibm.db2.app.Blob
VARCHAR(n)	java.lang.String
VARCHAR(n) FOR BIT DATA	COM.ibm.db2.app.Blob
GRAPHIC(n)	java.lang.String
VARGRAPHIC(n)	String
BLOB(n)	COM.ibm.db2.app.Blob
CLOB(n)	COM.ibm.db2.app.Clob
DBCLOB(n)	COM.ibm.db2.app.Clob
DATE	String
TIME	String
TIMESTAMP	String
XML AS CLOB(n)	COM.ibm.db2.jcc.DB2Xml

continue-handler

Certain errors that occur within a *nested-table-expression* can be tolerated, and instead of returning an error, the query can continue and return a result. This is referred to as an *error tolerant nested-table-expression*.

Specifying the RETURN DATA UNTIL clause will cause any rows that are returned from the fullselect before the indicated condition is encountered to make up the result set from the fullselect. This means that a partial result set (which can also be an empty result set) from the fullselect is acceptable as the result for the *nested-table-expression*.

The FEDERATED keyword restricts the condition to handle only errors that occur at a remote data source.

The condition can be specified as an SQLSTATE value, with a *string-constant* length of 5. You can optionally specify an SQLCODE value for each specified SQLSTATE value. For portable applications, specify SQLSTATE values as much as possible, because SQLCODE values are generally not portable across platforms and are not part of the SQL standard.

Only certain conditions can be tolerated. Errors that do not allow the rest of the query to be executed cannot be tolerated, and an error is returned for the whole query. The *specific-condition-value* might specify conditions that cannot actually be tolerated by the database manager, even if a specific SQLSTATE or SQLCODE value is specified, and for these cases, an error is returned.

A query or view containing an error tolerant *nested-table-expression* is read-only.

The fullselect of an error tolerant *nested-table-expression* is not optimized using materialized query tables.

specific-condition-value

The following SQLSTATE values and SQLCODE values have the potential, when specified, to be tolerated by the database manager:

- SQLSTATE 08001; SQLCODEs -1336, -30080, -30081, -30082
- SQLSTATE 08004
- SQLSTATE 42501
- SQLSTATE 42704; SQLCODE -204
- SQLSTATE 42720
- SQLSTATE 28000

Correlated references in table-references

Correlated references can be used in nested table expressions or as arguments to table functions. The basic rule that applies for both of these cases is that the correlated reference must be from a *table-reference* at a higher level in the hierarchy of subqueries. This hierarchy includes the table-references that have already been resolved in the left-to-right processing of the FROM clause. For nested table expressions, the LATERAL keyword must exist before the fullselect. The following examples have valid syntax:

```
Example 1:  SELECT t.c1, z.c5
            FROM t, TABLE( tf3(t.c2) ) AS z      -- t precedes tf3
            WHERE t.c3 = z.c4;                   -- in FROM, so t.c2
                                                 -- is known

Example 2:  SELECT t.c1, z.c5
            FROM t, TABLE( tf4(2 * t.c2) ) AS z  -- t precedes tf4
            WHERE t.c3 = z.c4;                   -- in FROM, so t.c2
                                                 -- is known

Example 3:  SELECT d.deptno, d.deptname,
                   empinfo.avgsal, empinfo.empcount
            FROM department d,
                 LATERAL (SELECT AVG(e.salary) AS avgsal,
                                 COUNT(*) AS empcount
                            FROM employee e           -- department precedes nested
                            WHERE e.workdept=d.deptno -- table expression and
                          ) AS empinfo;               -- LATERAL is specified,
                                                      -- so d.deptno is known
```

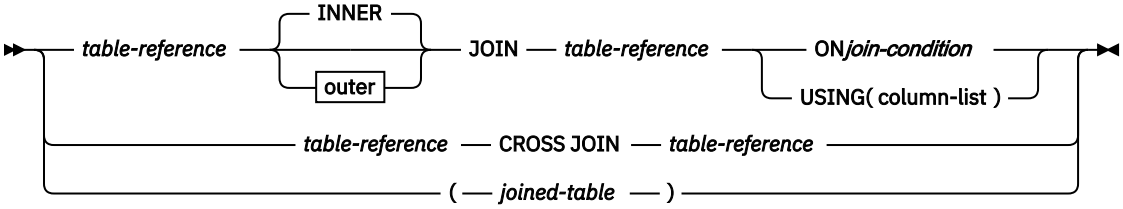
But the following examples are not valid:

```
Example 4:  SELECT t.c1, z.c5
            FROM TABLE( tf6(t.c2) ) AS z, t  -- cannot resolve t in t.c2!
            WHERE t.c3 = z.c4;               -- compare to Example 1 above.

Example 5:  SELECT a.c1, b.c5
            FROM TABLE( tf7a(b.c2) ) AS a, TABLE( tf7b(a.c6) ) AS b
            WHERE a.c3 = b.c4;               -- cannot resolve b in b.c2!

Example 6:  SELECT d.deptno, d.deptname,
                   empinfo.avgsal, empinfo.empcount
            FROM department d,
                 (SELECT AVG(e.salary) AS avgsal,
                                 COUNT(*) AS empcount
                            FROM employee e           -- department precedes nested
                            WHERE e.workdept=d.deptno -- table expression but
```

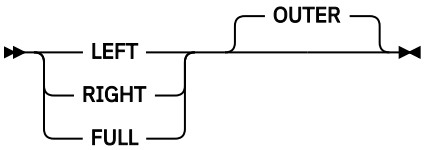
```
                      ) AS empinfo;              -- LATERAL is not specified,
                                                 -- so d.deptno is unknown
```

joined-table

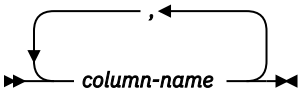
A *joined table* specifies an intermediate result table that is the result of either an inner join or an outer join. The table is derived by applying one of the join operators: CROSS, INNER, LEFT OUTER, RIGHT OUTER, or FULL OUTER to its operands.



outer



column-list



There are several types of joins:

Cross join
Represents the cross product of the tables, where each row of the left table is combined with every row of the right table.

Inner join
Keeps only the rows for which the join condition is true. Rows from either of the joined tables for which the join condition is false are excluded from the result table.

Outer join
Contains the rows for which the join condition is true, plus additional rows:

Left outer join
Also includes rows from the left table for which the join condition is false.

Right outer join
Also includes rows from the right table for which the join condition is false.

Full outer join
Also includes rows from both the left and right tables for which the join condition is false.

The order in which multiple joins are performed can affect the result. Joins can be nested within other joins. The order of processing for joins is generally from left to right, but based on the position of the required join-condition. You can use parentheses to clarify the order of nested joins. For example, the following two statements are equivalent:

```
T1 LEFT JOIN T2 ON T1.C1=T2.C1
  RIGHT JOIN T3 LEFT JOIN T4 ON T3.C1=T4.C1
    ON T1.C1=T3.C1
```

```
(T1 LEFT JOIN T2 ON T1.C1=T2.C1)
  RIGHT JOIN (T3 LEFT JOIN T4 ON T3.C1=T4.C1)
    ON T1.C1=T3.C1
```

A joined table can be used in any context in which any form of the SELECT statement is used. A view or a cursor is read-only if its SELECT statement includes a joined table.

Column references are resolved using the rules for resolution of column name qualifiers. The same rules that apply to predicates apply to join conditions.

Join operations

When a row of T1 is joined with a row of T2, a row in the result consists of the values of that row of T1 concatenated with the values of that row of T2. A join operation might result in the generation of a null row. A null row consists of a null value for each column of a table, regardless of whether null values are allowed in the columns.

The following list summarizes the result of the join operations:

- The result of T1 CROSS JOIN T2 consists of all possible pairings of their rows.
- The result of T1 INNER JOIN T2 consists of their paired rows where the join-condition is true.
- The result of T1 LEFT OUTER JOIN T2 consists of their paired rows where the join-condition is true and, for each unpaired row of T1, the concatenation of that row with the null row of T2. Null values are allowed in all columns derived from T2.
- The result of T1 RIGHT OUTER JOIN T2 consists of their paired rows where the join-condition is true and, for each unpaired row of T2, the concatenation of that row with the null row of T1. Null values are allowed in all columns derived from T1.
- The result of T1 FULL OUTER JOIN T2 consists of their paired rows and, for each unpaired row of T2, the concatenation of that row with the null row of T1 and, for each unpaired row of T1, the concatenation of that row with the null row of T2. Null values are allowed in all columns derived from T1 and T2.

Join conditions

A *join condition* is a qualification expression that involves the two tables to be joined. It specifies pairings of t1 and t2, where t1 and t2 represent the names of the left (t1) and right (t2) operand tables of the JOIN operator. For all possible combinations of rows of t1 and t2, a row of t1 is paired with a row of t2 if the join-condition is true.

A join condition is similar to a search condition, except that:

- It cannot include any dereference operations or the DEREF function, where the reference value is other than the object identifier column
- Any column referenced in an expression of the *join-condition* must be a column of one of the operand tables of the associated join (in the scope of the same joined-table clause)
- Any function referenced in an expression of the *join-condition* of a full outer join must be deterministic and have no external action
- It cannot include an XMLQUERY or XMLEXISTS expression

A join condition must comply with these rules (SQLSTATE 42972).

Join result

You influence which columns are included in the join result by specifying one of the following clauses:

ON <join-condition>
This clause can specify any qualification expression that involves the two tables that are to be joined. For example:

```
SELECT * FROM t1 JOIN t2 ON t1.c1 = t2.c1 AND t1.c2 = t2.c2;
```

The join result contains all columns of t1 followed by all columns of t2.

USING <column-list>

This clause joins the tables on the specified columns. Each column exists in both of the tables to be joined. The tables are joined where the value in a column is the same in both tables. For example:

```
SELECT * FROM t1 JOIN t2 USING (c1, c2);
```

The columns of the join result depend on the join type:

For an inner join or left outer join

The join result contains the join columns from the t1, followed by the non-join columns from t1, followed by the non-join columns from t2.

For a right outer join

The join result contains the join columns from t2, followed by the non-join columns from t1, followed by the non-join columns from t2.

For a full outer join

The join result has a non-null value from the join columns followed by the non-join columns from t1, followed by the non-join columns from t2.

Note that the join columns appear in the join result only once.

Unless a projection column list is explicitly specified, the order of the columns in the output is the same as in the join result.

Any subsequent, unqualified reference to a join column of a USING clause by another clause (such as a WHERE, ON, ORDER BY, GROUP BY, or HAVING clause) resolves to the column from the join result. For example:

```
CREATE TABLE t1 (c1 int, c2 varchar(10), c3 numeric(4,2));
CREATE TABLE t2 (c1 bigint, c2 char(8), c4 numeric(6,3));
CREATE TABLE t3 (c3 bigint, c5 int, c6 numeric(6,3));
```

```
SELECT * FROM t1 FULL JOIN t2 USING (c1, c2) JOIN t3 ON (c1 = t3.c3);
Column projections:
  CASE WHEN (t1.c1 IS NOT NULL) THEN t1.c1 ELSE t2.c1 END AS c1
  CASE WHEN (t1.c2 IS NOT NULL) THEN t1.c2 ELSE t2.c2 END AS c2
  t1.c3
  t2.c4
  t3.c3
  t3.c5
  t3.c6
```

The reference to column c1 in the ON clause resolves to the CASE expression that represents the join column c1 from the full outer join. So, the ON clause is transformed to:

```
ON ((CASE WHEN (t1.c1 IS NOT NULL) THEN t1.c1 ELSE t2.c1 END) = t3.c3)
```

The following restrictions apply:

- The ON and USING clauses are mutually exclusive for a particular join operation, that is, only one of these clauses can be specified when joining two tables. However, a single SQL statement can contain several join operations, and each of these can use either clause.
- An ON or USING clause cannot be used if a plus symbol (+) is used as the outer join operator.

Examples

Assume that the following three tables have been created:

```
CREATE TABLE t1 (c1 int, c2 varchar(10), c3 numeric(4,2));
CREATE TABLE t2 (c1 bigint, c2 char(8), c4 numeric(6,3));
CREATE TABLE t3 (c3 bigint, c5 int, c6 numeric(6,3));
```

- For a join with an ON clause:

```
SELECT * FROM t1 INNER JOIN t2 ON t1.c1 = t2.c1 AND t1.c2 = t2.c2;
Column projections: t1.c1, t1.c2, t1.c3, t2.c1, t2.c2, t2.c4
```

```
SELECT * FROM t1 FULL JOIN t2 ON t1.c1 = t2.c1 AND t1.c2 = t2.c2;
Column projections: t1.c1, t1.c2, t1.c3, t2.c1, t2.c2, t2.c4
```

- For an inner join or left outer join with a USING clause:

```
SELECT * FROM t1 INNER JOIN t2 USING (c1, c2);
Column projections: t1.c1, t1.c2, t1.c3, t2.c4
```

```
SELECT * FROM t1 LEFT JOIN t2 USING (c1, c2);
Column projections: t1.c1, t1.c2, t1.c3, t2.c4
```

- For a right outer join with a USING clause:

```
SELECT * FROM t1 RIGHT JOIN t2 USING (c1, c2);
Column projections: t2.c1, t2.c2, t1.c3, t2.c4
```

- For a full outer join with a USING clause:

```
SELECT * FROM t1 FULL JOIN t2 USING (c1, c2);
Column projections:
  CASE WHEN (t1.c1 IS NOT NULL) THEN t1.c1 ELSE t2.c1 END AS c1
  CASE WHEN (t1.c2 IS NOT NULL) THEN t1.c2 ELSE t2.c2 END AS c2
  t1.c3
  t2.c4
```

Examples of subselect queries with joins
The following examples illustrate the use of joins in a subselect query.

- *Example 1:* This example illustrates the results of the various joins using tables J1 and J2. These tables contain rows as shown.

```
SELECT * FROM J1

W   X
--- ------
A       11
B       12
C       13

SELECT * FROM J2

Y   Z
--- ------
A       21
C       22
D       23
```

The following query does an inner join of J1 and J2 matching the first column of both tables.

```
SELECT * FROM J1 INNER JOIN J2 ON W=Y

W   X      Y   Z
--- ------ --- ------
A       11 A       21
C       13 C       22
```

In this inner join example the row with column W='C' from J1 and the row with column Y='D' from J2 are not included in the result because they do not have a match in the other table. Note that the following alternative form of an inner join query produces the same result.

```
SELECT * FROM J1, J2 WHERE W=Y
```

The following left outer join will get back the missing row from J1 with nulls for the columns of J2. Every row from J1 is included.

```
SELECT * FROM J1 LEFT OUTER JOIN J2 ON W=Y

W   X      Y   Z
--- ------ --- ------
A       11 A       21
```

```
B       12 -        -
C       13 C       22
```

The following right outer join will get back the missing row from J2 with nulls for the columns of J1. Every row from J2 is included.

```
SELECT * FROM J1 RIGHT OUTER JOIN J2 ON W=Y

W   X      Y   Z
--- ------ --- ------
A       11 A       21
C       13 C       22
-        - D       23
```

The following full outer join will get back the missing rows from both J1 and J2 with nulls where appropriate. Every row from both J1 and J2 is included.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y

W   X      Y   Z
--- ------ --- ------
A       11 A       21
C       13 C       22
-        - D       23
B       12 -        -
```

- *Example 2:* Using the tables J1 and J2 from the previous example, examine what happens when and additional predicate is added to the search condition.

```
SELECT * FROM J1 INNER JOIN J2 ON W=Y AND X=13

W   X      Y   Z
--- ------ --- ------
C       13 C       22
```

The additional condition caused the inner join to select only 1 row compared to the inner join in Example 1.

Notice what the affect of this is on the full outer join.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y AND X=13

W   X      Y   Z
--- ------ --- ------
-        - A       21
C       13 C       22
-        - D       23
A       11 -        -
B       12 -        -
```

The result now has 5 rows (compared to 4 without the additional predicate) because there was only 1 row in the inner join and all rows of both tables must be returned.

The following query illustrates that placing the same additional predicate in WHERE clause has completely different results.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y
   WHERE X=13

W   X      Y   Z
--- ------ --- ------
C       13 C       22
```

The WHERE clause is applied after the intermediate result of the full outer join. This intermediate result is the same as the result of the full outer join query in Example 1. The WHERE clause is applied to this intermediate result and eliminates all but the row that has X=13. Choosing the location of a predicate

when performing outer joins can have a significant affect on the results. Consider what happens if the predicate was X=12 instead of X=13. The following inner join returns no rows.

```
SELECT * FROM J1 INNER JOIN J2 ON W=Y AND X=12
```

Hence, the full outer join returns 6 rows, 3 from J1 with nulls for the columns of J2 and 3 from J2 with nulls for the columns of J1.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y AND X=12

W   X      Y   Z
--- ------ --- ------
-        - A       21
-        - C       22
-        - D       23
A       11 -        -
B       12 -        -
C       13 -        -
```

If the additional predicate is in the WHERE clause instead, 1 row is returned.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y
   WHERE X=12

W   X      Y   Z
--- ------ --- ------
B       12 -        -
```

- *Example 3:* List every department with the employee number and last name of the manager, including departments without a manager.

```
SELECT DEPTNO, DEPTNAME, EMPNO, LASTNAME
  FROM DEPARTMENT LEFT OUTER JOIN EMPLOYEE
       ON MGRNO = EMPNO
```

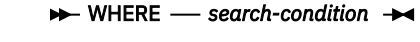
- *Example 4:* List every employee number and last name with the employee number and last name of their manager, including employees without a manager.

```
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
  FROM EMPLOYEE E LEFT OUTER JOIN
                                        DEPARTMENT INNER JOIN EMPLOYEE M
       ON MGRNO = M.EMPNO
       ON E.WORKDEPT = DEPTNO
```

The inner join determines the last name for any manager identified in the DEPARTMENT table and the left outer join guarantees that each employee is listed even if a corresponding department is not found in DEPARTMENT.

where-clause

The WHERE clause specifies an intermediate result table that consists of those rows of R for which the *search-condition* is true. R is the result of the FROM clause of the subselect.



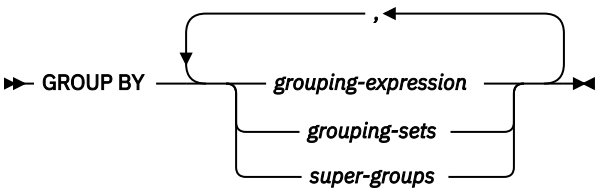
The *search-condition* must conform to the following rules:

- Each *column-name* must unambiguously identify a column of R or be a correlated reference. A *column-name* is a correlated reference if it identifies a column of a *table-reference* in an outer subselect.
- An aggregate function must not be specified unless the WHERE clause is specified in a subquery of a HAVING clause and the argument of the function is a correlated reference to a group.

Any subquery in the *search-condition* is effectively executed for each row of R, and the results are used in the application of the *search-condition* to the given row of R. A subquery is actually executed for each row of R only if it includes a correlated reference. In fact, a subquery with no correlated references might be executed just once, whereas a subquery with a correlated reference might be executed once for each row.

group-by-clause

The GROUP BY clause specifies an intermediate result table that consists of a grouping of the rows of R. R is the result of the previous clause of the subselect.



In its simplest form, a GROUP BY clause contains a *grouping expression*. A grouping expression is an *expression* used in defining the grouping of R. Each expression or *column name* included in grouping-expression must unambiguously identify a column of R (SQLSTATE 42702 or 42703). A grouping expression cannot include a scalar fullselect or an XMLQUERY or XMLEXISTS expression (SQLSTATE 42822), or any expression or function that is not deterministic or has an external action (SQLSTATE 42845).

Note: The following expressions, which do not contain an explicit column reference, can be used in a *grouping-expression* to identify a column of R:

- ROW CHANGE TIMESTAMP FOR *table-designator*
- ROW CHANGE TOKEN FOR *table-designator*
- RID_BIT or RID scalar function

More complex forms of the GROUP BY clause include *grouping-sets* and *super-groups*. For a description of *grouping-sets*, see "grouping-sets" on page 692. For a description of *super-groups*, see "super-groups" on page 692.

The result of GROUP BY is a set of groups of rows. Each row in this result represents the set of rows for which the *grouping-expression* is equal. For grouping, all null values from a *grouping-expression* are considered equal.

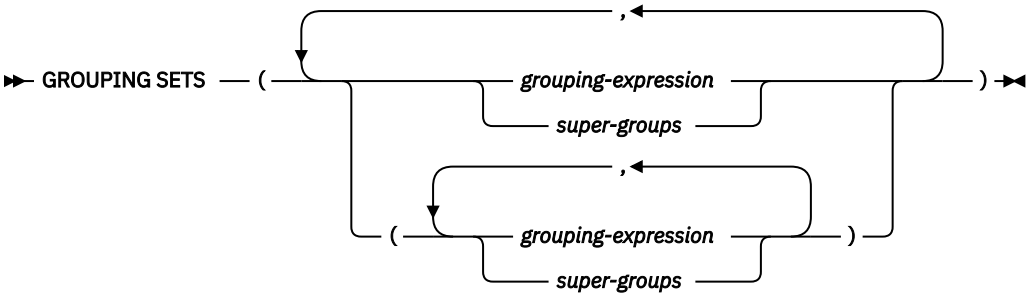
If a *grouping-expression* contains decimal floating-point columns, and multiple representations of the same number exist in these columns, the number that is returned can be any of the representations of the number.

A *grouping-expression* can be used in a search condition in a HAVING clause, in an expression in a SELECT clause, or in a *sort-key-expression* of an ORDER BY clause (see "order-by-clause" on page 703 for details). In each case, the reference specifies only one value for each group. For example, if the *grouping-expression* is *col1+col2*, then an allowed expression in the SELECT list is *col1+col2+3*. Associativity rules for expressions disallow the similar expression, *3+col1+col2*, unless parentheses are used to ensure that the corresponding expression is evaluated in the same order. Thus, *3+(col1+col2)* is also allowed in the SELECT list. If the concatenation operator is used, the *grouping-expression* must be used exactly as the expression was specified in the SELECT list.

If the *grouping-expression* contains varying-length strings with trailing blanks, the values in the group can differ in the number of trailing blanks and might not all have the same length. In that case, a reference to the *grouping-expression* still specifies only one value for each group. However, the value for a group is chosen arbitrarily from the available set of values or a normalized form that might or might not be from the available set of values. Thus, the actual length of the result value is unpredictable.

As noted, there are some cases where the GROUP BY clause cannot refer directly to a column that is specified in the SELECT clause as an expression (scalar-fullselect, not deterministic or external action functions). To group using such an expression, use a nested table expression or a common table expression to first provide a result table with the expression as a column of the result. For an example using nested table expressions, see Example 9 in "Examples of subselect queries" on page 708.

grouping-sets



A *grouping-sets* specification can be used to specify multiple grouping clauses in a single statement. This can be thought of as the union of two or more groups of rows into a single result set. It is logically equivalent to the union of multiple subselects with the group by clause in each subselect corresponding to one grouping set. A grouping set can be a single element or can be a list of elements delimited by parentheses, where an element is either a grouping-expression or a super-group. The groups can be computed with a single pass over the base table using *grouping-sets*.

A simple *grouping-expression* or the more complex forms of *super-groups* are supported by the *grouping-sets* specification. For a description of *super-groups*, see "super-groups" on page 692.

Note that grouping sets are the fundamental building blocks for GROUP BY operations. A simple GROUP BY with a single column can be considered a grouping set with one element. For example:

```
GROUP BY a
```

is the same as

```
GROUP BY GROUPING SETS((a))
```

and

```
GROUP BY a,b,c
```

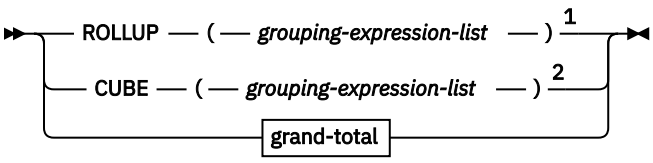
is the same as

```
GROUP BY GROUPING SETS((a,b,c))
```

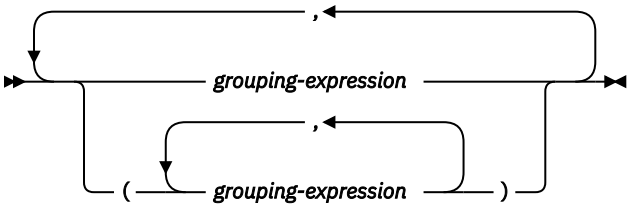
Non-aggregation columns from the SELECT list of the subselect that are excluded from a grouping set will return a null for such columns for each row generated for that grouping set. This reflects the fact that aggregation was done without considering the values for those columns.

The use of grouping sets is illustrated in Example 2 through Example 7 in "Examples of grouping sets, cube, and rollup queries" on page 696.

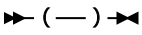
super-groups



grouping-expression-list



grand-total



Notes:

[1] Alternate specification when used alone in group-by-clause is: grouping-expression-list WITH ROLLUP.

[2] Alternate specification when used alone in group-by-clause is: grouping-expression-list WITH CUBE.

ROLLUP (*grouping-expression-list*)

A *ROLLUP grouping* is an extension to the GROUP BY clause that produces a result set containing *sub-total* rows in addition to the "regular" grouped rows. *Sub-total* rows are "super-aggregate" rows that contain further aggregates whose values are derived by applying the same aggregate functions that were used to obtain the grouped rows. These rows are called sub-total rows, because that is their most common use; however, any aggregate function can be used for the aggregation. For instance, MAX and AVG are used in Example 8 in "Examples of grouping sets, cube, and rollup queries" on page 696. The GROUPING aggregate function can be used to indicate if a row was generated by the super-group.

A ROLLUP grouping is a series of *grouping-sets*. The general specification of a ROLLUP with *n* elements

```
GROUP BY ROLLUP(C1,C2,...,Cn-1,Cn)
```

is equivalent to

```
GROUP BY GROUPING SETS((C1,C2,...,Cn-1,Cn)
                       (C1,C2,...,Cn-1)
                       ...
                       (C1,C2)
                       (C1)
                       () )
```

Note that the *n* elements of the ROLLUP translate to *n*+1 grouping sets. Note also that the order in which the *grouping-expressions* is specified is significant for ROLLUP. For example, the following clause:

```
GROUP BY ROLLUP(a,b)
```

is equivalent to:

```
GROUP BY GROUPING SETS((a,b)
                       (a)
                       () )
```

Similarly, the following clause:

```
GROUP BY ROLLUP(b,a)
```

is equivalent to:

```
GROUP BY GROUPING SETS((b,a)
                       (b)
                       () )
```

The ORDER BY clause is the only way to guarantee the order of the rows in the result set. Example 3 in "Examples of grouping sets, cube, and rollup queries" on page 696 illustrates the use of ROLLUP.

CUBE (*grouping-expression-list*)

A *CUBE grouping* is an extension to the GROUP BY clause that produces a result set that contains all the rows of a ROLLUP aggregation and, in addition, contains "cross-tabulation" rows. *Cross-tabulation* rows are additional "super-aggregate" rows that are not part of an aggregation with sub-totals. The GROUPING aggregate function can be used to indicate if a row was generated by the super-group.

Similar to a ROLLUP, a CUBE grouping can also be thought of as a series of *grouping-sets*. In the case of a CUBE, all permutations of the cubed *grouping-expression-list* are computed along with the grand total. Therefore, the *n* elements of a CUBE translate to 2**n* (2 to the power *n*) *grouping-sets*. For example, a specification of:

```
GROUP BY CUBE(a,b,c)
```

is equivalent to:

```
GROUP BY GROUPING SETS((a,b,c)
                       (a,b)
                       (a,c)
                       (b,c)
                       (a)
                       (b)
                       (c)
                       () )
```

Note that the three elements of the CUBE translate into eight grouping sets.

The order of specification of elements does not matter for CUBE. 'CUBE (DayOfYear, Sales_Person)' and 'CUBE (Sales_Person, DayOfYear)' yield the same result sets. The use of the word 'same' applies to content of the result set, not to its order. The ORDER BY clause is the only way to guarantee the order of the rows in the result set. The use of CUBE is illustrated in Example 4 in “Examples of grouping sets, cube, and rollup queries” on page 696.

grouping-expression-list

A *grouping-expression-list* is used within a CUBE or ROLLUP clause to define the number of elements in the CUBE or ROLLUP operation. This is controlled by using parentheses to delimit elements with multiple *grouping-expression*s.

For example, suppose that a query is to return the total expenses for the ROLLUP of City within a Province but not within a County. However, the clause:

```
GROUP BY ROLLUP(Province, County, City)
```

results in unwanted subtotal rows for the County. In the clause:

```
GROUP BY ROLLUP(Province, (County, City))
```

the composite (County, City) forms one element in the ROLLUP and, therefore, a query that uses this clause will yield the required result. In other words, the two-element ROLLUP:

```
GROUP BY ROLLUP(Province, (County, City))
```

generates:

```
GROUP BY GROUPING SETS((Province, County, City)
                       (Province)
                       () )
```

and the three-element ROLLUP generates:

```
GROUP BY GROUPING SETS((Province, County, City)
                       (Province, County)
                       (Province)
                       () )
```

Example 2 in “Examples of grouping sets, cube, and rollup queries” on page 696 also utilizes composite column values.

grand-total
Both CUBE and ROLLUP return a row which is the overall (grand total) aggregation. This can be separately specified with empty parentheses within the GROUPING SET clause. It can also be specified directly in the GROUP BY clause, although there is no effect on the result of the query. Example 4 in “Examples of grouping sets, cube, and rollup queries” on page 696 uses the grand-total syntax.

Combining grouping sets

This can be used to combine any of the types of GROUP BY clauses. When simple *grouping-expression* fields are combined with other groups, they are "appended" to the beginning of the resulting *grouping sets*. When ROLLUP or CUBE expressions are combined, they operate similarly to "multipliers" on the remaining expression, forming additional grouping set entries according to the definition of either ROLLUP or CUBE.

For instance, combining *grouping-expression* elements acts as follows:

```
GROUP BY a, ROLLUP(b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c)
                       (a,b)
                       (a) )
```

Or similarly,

```
GROUP BY a, b, ROLLUP(c,d)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c,d)
                       (a,b,c)
                       (a,b) )
```

Combining of *ROLLUP* elements acts as follows:

```
GROUP BY ROLLUP(a), ROLLUP(b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c)
                       (a,b)
                       (a)
                       (b,c)
                       (b)
                       () )
```

Similarly,

```
GROUP BY ROLLUP(a), CUBE(b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c)
                       (a,b)
                       (a,c)
                       (a)
                       (b,c)
                       (b)
                       (c)
                       () )
```

Combining of *CUBE* and *ROLLUP* elements acts as follows:

```
GROUP BY CUBE(a,b), ROLLUP(c,d)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c,d)
                       (a,b,c)
                       (a,b)
                       (a,c,d)
                       (a,c)
                       (a)
                       (b,c,d)
                       (b,c)
                       (b)
                       (c,d)
                       (c)
                       () )
```

Similar to a simple *grouping-expression*, combining grouping sets also eliminates duplicates within each grouping set. For instance,

```
GROUP BY a, ROLLUP(a,b)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b)
                       (a) )
```

A more complete example of combining grouping sets is to construct a result set that eliminates certain rows that might be returned for a full CUBE aggregation.

For example, consider the following GROUP BY clause:

```
GROUP BY Region,
         ROLLUP(Sales_Person, WEEK(Sales_Date)),
         CUBE(YEAR(Sales_Date), MONTH (Sales_Date))
```

The column listed immediately to the right of GROUP BY is grouped, those within the parenthesis following ROLLUP are rolled up, and those within the parenthesis following CUBE are cubed. Thus, the GROUP BY clause results in a cube of MONTH within YEAR which is then rolled up within WEEK within Sales_Person within the Region aggregation. It does not result in any grand total row or any cross-tabulation rows on Region, Sales_Person or WEEK(Sales_Date) so produces fewer rows than the clause:

```
GROUP BY ROLLUP (Region, Sales_Person, WEEK(Sales_Date),
                 YEAR(Sales_Date), MONTH(Sales_Date) )
```

Examples of grouping sets, cube, and rollup queries

The following examples illustrate the grouping, cube, and rollup forms of subselect queries.

The queries in Example 1 through Example 4 use a subset of the rows in the SALES tables based on the predicate 'WEEK(SALES_DATE) = 13'.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SALES AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
```

which results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           6 LUCCHESSI                 3
         13           6 LUCCHESSI                 1
         13           6 LEE                       2
         13           6 LEE                       2
```

```
          13           6 LEE                        3
          13           6 LEE                        5
          13           6 GOUNOT                     3
          13           6 GOUNOT                     1
          13           6 GOUNOT                     7
          13           7 LUCCHESSI                  1
          13           7 LUCCHESSI                  2
          13           7 LUCCHESSI                  1
          13           7 LEE                        7
          13           7 LEE                        3
          13           7 LEE                        7
          13           7 LEE                        4
          13           7 GOUNOT                     2
          13           7 GOUNOT                    18
          13           7 GOUNOT                     1
```

- *Example 1:* Here is a query with a basic GROUP BY clause over 3 columns:

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

  This results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           6 GOUNOT                   11
         13           6 LEE                      12
         13           6 LUCCHESSI                 4
         13           7 GOUNOT                   21
         13           7 LEE                      21
         13           7 LUCCHESSI                 4
```

- *Example 2:* Produce the result based on two different grouping sets of rows from the SALES table.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY GROUPING SETS ( (WEEK(SALES_DATE), SALES_PERSON),
                         (DAYOFWEEK(SALES_DATE), SALES_PERSON))
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

  This results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           - GOUNOT                   32
         13           - LEE                      33
         13           - LUCCHESSI                 8
          -           6 GOUNOT                   11
          -           6 LEE                      12
          -           6 LUCCHESSI                 4
          -           7 GOUNOT                   21
          -           7 LEE                      21
          -           7 LUCCHESSI                 4
```

  The rows with WEEK 13 are from the first grouping set and the other rows are from the second grouping set.

- *Example 3:* If you use the 3 distinct columns involved in the grouping sets of Example 2 and perform a ROLLUP, you can see grouping sets for (WEEK,DAY_WEEK,SALES_PERSON), (WEEK, DAY_WEEK), (WEEK) and grand total.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
  FROM SALES
  WHERE WEEK(SALES_DATE) = 13
```

```
GROUP BY ROLLUP ( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON )
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

This results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           6 GOUNOT                   11
         13           6 LEE                      12
         13           6 LUCCHESSI                 4
         13           6 -                        27
         13           7 GOUNOT                   21
         13           7 LEE                      21
         13           7 LUCCHESSI                 4
         13           7 -                        46
         13           - -                        73
          -           - -                        73
```

- *Example 4:* If you run the same query as Example 3 only replace ROLLUP with CUBE, you can see additional grouping sets for (WEEK,SALES_PERSON), (DAY_WEEK,SALES_PERSON), (DAY_WEEK), (SALES_PERSON) in the result.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY CUBE ( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON )
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

This results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           6 GOUNOT                   11
         13           6 LEE                      12
         13           6 LUCCHESSI                 4
         13           6 -                        27
         13           7 GOUNOT                   21
         13           7 LEE                      21
         13           7 LUCCHESSI                 4
         13           7 -                        46
         13           - GOUNOT                   32
         13           - LEE                      33
         13           - LUCCHESSI                 8
         13           - -                        73
          -           6 GOUNOT                   11
          -           6 LEE                      12
          -           6 LUCCHESSI                 4
          -           6 -                        27
          -           7 GOUNOT                   21
          -           7 LEE                      21
          -           7 LUCCHESSI                 4
          -           7 -                        46
          -           - GOUNOT                   32
          -           - LEE                      33
          -           - LUCCHESSI                 8
          -           - -                        73
```

- *Example 5:* Obtain a result set which includes a grand-total of selected rows from the SALES table together with a group of rows aggregated by SALES_PERSON and MONTH.

```
SELECT SALES_PERSON,
       MONTH(SALES_DATE) AS MONTH,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY GROUPING SETS ( (SALES_PERSON, MONTH(SALES_DATE)),
                         ()
                       )
ORDER BY SALES_PERSON, MONTH
```

This results in:

```
SALES_PERSON    MONTH       UNITS_SOLD
--------------- ----------- -----------
```

```
GOUNOT                   3          35
GOUNOT                   4          14
GOUNOT                  12           1
LEE                      3          60
LEE                      4          25
LEE                     12           6
LUCCHESSI                3           9
LUCCHESSI                4           4
LUCCHESSI               12           1
-                        -         155
```

- *Example 6:* This example shows two simple ROLLUP queries followed by a query which treats the two ROLLUPs as grouping sets in a single result set and specifies row ordering for each column involved in the grouping sets.
  - *Example 6-1:*

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY ROLLUP ( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE) )
ORDER BY WEEK, DAY_WEEK
```

    results in:

```
WEEK        DAY_WEEK    UNITS_SOLD
----------- ----------- -----------
         13           6          27
         13           7          46
         13           -          73
         14           1          31
         14           2          43
         14           -          74
         53           1           8
         53           -           8
          -           -         155
```

  - *Example 6-2:*

```
SELECT MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY ROLLUP ( MONTH(SALES_DATE), REGION );
ORDER BY MONTH, REGION
```

    results in:

```
MONTH       REGION          UNITS_SOLD
----------- --------------- -----------
          3 Manitoba                 22
          3 Ontario-North             8
          3 Ontario-South            34
          3 Quebec                   40
          3 -                       104
          4 Manitoba                 17
          4 Ontario-North             1
          4 Ontario-South            14
          4 Quebec                   11
          4 -                        43
         12 Manitoba                  2
         12 Ontario-South             4
         12 Quebec                    2
         12 -                         8
          - -                       155
```

  - *Example 6-3:*

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY GROUPING SETS ( ROLLUP( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE) ),
```

```
                          ROLLUP( MONTH(SALES_DATE), REGION )  )
ORDER BY WEEK, DAY_WEEK, MONTH, REGION
```

results in:

```
  WEEK        DAY_WEEK    MONTH       REGION          UNITS_SOLD
  ----------- ----------- ----------- --------------- -----------
           13           6           - -                        27
           13           7           - -                        46
           13           -           - -                        73
           14           1           - -                        31
           14           2           - -                        43
           14           -           - -                        74
           53           1           - -                         8
           53           -           - -                         8
            -           -           3 Manitoba                 22
            -           -           3 Ontario-North             8
            -           -           3 Ontario-South            34
            -           -           3 Quebec                   40
            -           -           3 -                       104
            -           -           4 Manitoba                 17
            -           -           4 Ontario-North             1
            -           -           4 Ontario-South            14
            -           -           4 Quebec                   11
            -           -           4 -                        43
            -           -          12 Manitoba                  2
            -           -          12 Ontario-South             4
            -           -          12 Quebec                    2
            -           -          12 -                         8
            -           -           - -                       155
            -           -           - -                       155
```

Using the two ROLLUPs as grouping sets causes the result to include duplicate rows. There are even two grand total rows.

Observe how the use of ORDER BY has affected the results:

- In the first grouped set, week 53 has been repositioned to the end.
- In the second grouped set, month 12 has now been positioned to the end and the regions now display in alphabetic order.
- Null values are sorted high.

- *Example 7:* In queries that perform multiple ROLLUPs in a single pass (such as Example 6-3) you might want to be able to indicate which grouping set produced each row. The following steps demonstrate how to provide a column (called GROUP) which indicates the *origin* of each row in the result set. *Origin* means which one of the two grouping sets produced the row in the result set.

  Step 1: Introduce a way of "generating" new data values, using a query which selects from a VALUES clause (which is an alternative form of a fullselect). This query shows how a table can be derived called "X" having 2 columns "R1" and "R2" and 1 row of data.

  ```
  SELECT R1,R2
  FROM (VALUES('GROUP 1','GROUP 2')) AS X(R1,R2);
  ```

  results in:

  ```
  R1      R2
  ------- -------
  GROUP 1 GROUP 2
  ```

  Step 2: Form the cross product of this table "X" with the SALES table. This add columns "R1" and "R2" to every row.

  ```
  SELECT R1, R2, WEEK(SALES_DATE) AS WEEK,
         DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
         MONTH(SALES_DATE) AS MONTH,
         REGION,
         SALES AS UNITS_SOLD
  FROM SALES,(VALUES('GROUP 1','GROUP 2')) AS X(R1,R2)
  ```

  This add columns "R1" and "R2" to every row.

Step 3: Now these columns can be combined with the grouping sets to include these columns in the rollup analysis.

```
SELECT R1, R2,
       WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION, SUM(SALES) AS UNITS_SOLD
FROM SALES,(VALUES('GROUP 1','GROUP 2')) AS X(R1,R2)
GROUP BY GROUPING SETS ((R1, ROLLUP(WEEK(SALES_DATE),
                         DAYOFWEEK(SALES_DATE))),
                       (R2,ROLLUP( MONTH(SALES_DATE), REGION ) )  )
ORDER BY WEEK, DAY_WEEK, MONTH, REGION
```

results in:

```
R1      R2      WEEK     DAY_WEEK  MONTH     REGION          UNITS_SOLD
------- ------- -------- --------- --------- --------------- -----------
GROUP 1 -             13         6         - -                          27
GROUP 1 -             13         7         - -                          46
GROUP 1 -             13         -         - -                          73
GROUP 1 -             14         1         - -                          31
GROUP 1 -             14         2         - -                          43
GROUP 1 -             14         -         - -                          74
GROUP 1 -             53         1         - -                           8
GROUP 1 -             53         -         - -                           8
-       GROUP 2        -         -         3 Manitoba                   22
-       GROUP 2        -         -         3 Ontario-North               8
-       GROUP 2        -         -         3 Ontario-South              34
-       GROUP 2        -         -         3 Quebec                     40
-       GROUP 2        -         -         3 -                         104
-       GROUP 2        -         -         4 Manitoba                   17
-       GROUP 2        -         -         4 Ontario-North               1
-       GROUP 2        -         -         4 Ontario-South              14
-       GROUP 2        -         -         4 Quebec                     11
-       GROUP 2        -         -         4 -                          43
-       GROUP 2        -         -        12 Manitoba                    2
-       GROUP 2        -         -        12 Ontario-South               4
-       GROUP 2        -         -        12 Quebec                      2
-       GROUP 2        -         -        12 -                           8
-       GROUP 2        -         -         - -                         155
GROUP 1 -              -         -         - -                         155
```

Step 4: Notice that because R1 and R2 are used in different grouping sets, whenever R1 is non-null in the result, R2 is null and whenever R2 is non-null in the result, R1 is null. That means you can consolidate these columns into a single column using the COALESCE function. You can also use this column in the ORDER BY clause to keep the results of the two grouping sets together.

```
SELECT COALESCE(R1,R2) AS GROUP,
       WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION, SUM(SALES) AS UNITS_SOLD
FROM SALES,(VALUES('GROUP 1','GROUP 2')) AS X(R1,R2)
GROUP BY GROUPING SETS ((R1, ROLLUP(WEEK(SALES_DATE),
                         DAYOFWEEK(SALES_DATE))),
                       (R2,ROLLUP( MONTH(SALES_DATE), REGION ) )  )
ORDER BY GROUP, WEEK, DAY_WEEK, MONTH, REGION;
```

results in:

```
GROUP   WEEK        DAY_WEEK    MONTH       REGION          UNITS_SOLD
------- ----------- ----------- ----------- --------------- -----------
GROUP 1          13           6           - -                          27
GROUP 1          13           7           - -                          46
GROUP 1          13           -           - -                          73
GROUP 1          14           1           - -                          31
GROUP 1          14           2           - -                          43
GROUP 1          14           -           - -                          74
GROUP 1          53           1           - -                           8
GROUP 1          53           -           - -                           8
GROUP 1           -           -           - -                         155
GROUP 2           -           -           3 Manitoba                   22
GROUP 2           -           -           3 Ontario-North               8
GROUP 2           -           -           3 Ontario-South              34
GROUP 2           -           -           3 Quebec                     40
```

```
GROUP 2           -           -           3 -                       104
GROUP 2           -           -           4 Manitoba                 17
GROUP 2           -           -           4 Ontario-North             1
GROUP 2           -           -           4 Ontario-South            14
GROUP 2           -           -           4 Quebec                   11
GROUP 2           -           -           4 -                        43
GROUP 2           -           -          12 Manitoba                  2
GROUP 2           -           -          12 Ontario-South             4
GROUP 2           -           -          12 Quebec                    2
GROUP 2           -           -          12 -                         8
GROUP 2           -           -           - -                       155
```

- *Example 8:* The following example illustrates the use of various aggregate functions when performing a CUBE. The example also makes use of cast functions and rounding to produce a decimal result with reasonable precision and scale.

```
SELECT MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD,
       MAX(SALES) AS BEST_SALE,
       CAST(ROUND(AVG(DECIMAL(SALES)),2) AS DECIMAL(5,2)) AS AVG_UNITS_SOLD
FROM SALES
GROUP BY CUBE(MONTH(SALES_DATE),REGION)
ORDER BY MONTH, REGION
```

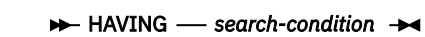
This results in:

```
MONTH       REGION          UNITS_SOLD  BEST_SALE   AVG_UNITS_SOLD
----------- --------------- ----------- ----------- --------------
          3 Manitoba                 22           7           3.14
          3 Ontario-North             8           3           2.67
          3 Ontario-South            34          14           4.25
          3 Quebec                   40          18           5.00
          3 -                       104          18           4.00
          4 Manitoba                 17           9           5.67
          4 Ontario-North             1           1           1.00
          4 Ontario-South            14           8           4.67
          4 Quebec                   11           8           5.50
          4 -                        43           9           4.78
         12 Manitoba                  2           2           2.00
         12 Ontario-South             4           3           2.00
         12 Quebec                    2           1           1.00
         12 -                         8           3           1.60
          - Manitoba                 41           9           3.73
          - Ontario-North             9           3           2.25
          - Ontario-South            52          14           4.00
          - Quebec                   53          18           4.42
          - -                       155          18           3.87
```

having-clause

The HAVING clause specifies an intermediate result table that consists of those groups of R for which the *search-condition* is true.

R is the result of the previous clause of the subselect. If this clause is not GROUP BY, R is considered to be a single group with no grouping columns.



Each *column-name* in the search condition must satisfy one of the following conditions:

- Unambiguously identify a grouping column of R.
- Be specified within an aggregate function.
- Be a correlated reference. A *column-name* is a correlated reference if it identifies a column of a *table-reference* in an outer subselect.

A group of R to which the search condition is applied supplies the argument for each aggregate function in the search condition, except for any function whose argument is a correlated reference.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a group of R, and the results used in applying the search condition.

In actuality, the subquery is executed for each group only if it contains a correlated reference. For an illustration of the difference, see Example 6 and Example 7 in "Examples of subselect queries" on page 708.

A correlated reference to a group of R must either identify a grouping column or be contained within an aggregate function.

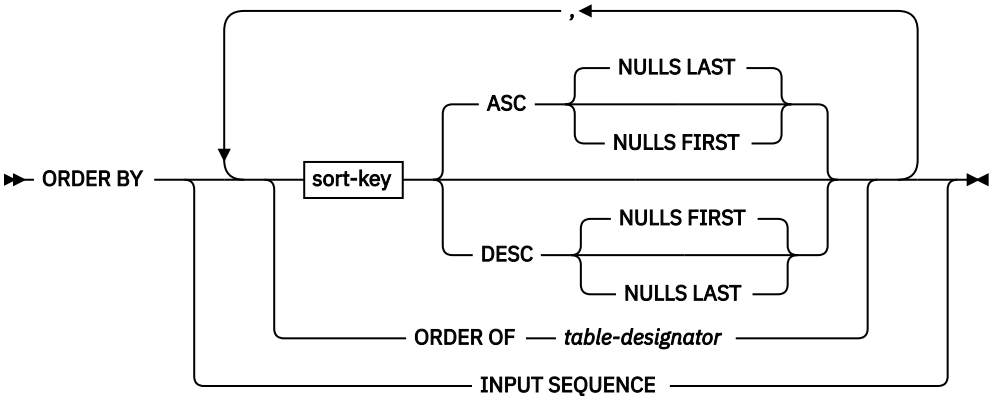
When HAVING is used without GROUP BY, the select list can only include column names when they are arguments to an aggregate function, correlated column references, global variables, host variables, literals, special registers, SQL variables, or SQL parameters.

Note: The following expressions can only be specified in a HAVING clause if they are contained within an aggregate function (SQLSTATE 42803):

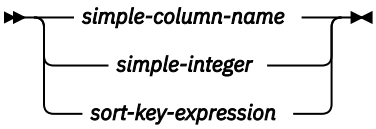
- ROW CHANGE TIMESTAMP FOR *table-designator*
- ROW CHANGE TOKEN FOR *table-designator*
- RID_BIT or RID scalar function

order-by-clause

The ORDER BY clause specifies an ordering of the rows of the result table.



sort-key



If a single sort specification (one *sort-key* with associated direction) is identified, the rows are ordered by the values of that sort specification. If more than one sort specification is identified, the rows are ordered by the values of the first identified sort specification, then by the values of the second identified sort specification, and so on. Each *sort-key* cannot have a data type of CLOB, DBCLOB, BLOB, XML, distinct type on any of these types, or structured type (SQLSTATE 42907).

A named column in the select list can be identified by a *sort-key* that is a *simple-integer* or a *simple-column-name*. An unnamed column in the select list must be identified by an *simple-integer* or, in some cases, by a *sort-key-expression* that matches the expression in the select list (see details of *sort-key-expression*). A column is unnamed if the AS clause is not specified and it is derived from a constant, an expression with operators, or a function.

Ordering is performed in accordance with comparison rules. If an ORDER BY clause contains decimal floating-point columns, and multiple representations of the same number exist in these columns, the ordering of the multiple representations of the same number is unspecified. The null value is higher than all other values. If the ORDER BY clause does not completely order the rows, rows with duplicate values of all identified columns are displayed in an arbitrary order.

simple-column-name
Usually identifies a column of the result table. In this case, *simple-column-name* must be the column name of a named column in the select list.

The *simple-column-name* can also identify a column name of a table, view, or nested table identified in the FROM clause if the query is a subselect. This includes columns defined as implicitly hidden. An error occurs in the following situations:

- If the subselect specifies DISTINCT in the select-clause (SQLSTATE 42822)
- If the subselect produces a grouped result and the *simple-column-name* is not a *grouping-expression* (SQLSTATE 42803)

Determining which column is used for ordering the result is described under "Column names in sort keys" in the "Notes" section.

simple-integer
Must be greater than 0 and not greater than the number of columns in the result table (SQLSTATE 42805). The integer *n* identifies the *n*th column of the result table.

sort-key-expression
An expression that is not simply a column name or an unsigned integer constant. The query to which ordering is applied must be a *subselect* to use this form of sort-key. The *sort-key-expression* cannot include a correlated scalar fullselect (SQLSTATE 42703) or a function with an external action (SQLSTATE 42845).

Any *column-name* within a *sort-key-expression* must conform to the rules described under "Column names in sort keys" in the "Notes" section.

There are a number of special cases that further restrict the expressions that can be specified.

- DISTINCT is specified in the SELECT clause of the subselect (SQLSTATE 42822).

  The sort-key-expression must match exactly with an expression in the select list of the subselect (scalar-fullselects are never matched).
- The subselect is grouped (SQLSTATE 42803).

  The sort-key-expression can:

  – be an expression in the select list of the subselect,
  – include a *grouping-expression* from the GROUP BY clause of the subselect
  – include an aggregate function, constant or host variable.

ASC
Order the rows in ascending order. This is the default.

DESC
Order the rows in descending order.

NULLS FIRST
When ordering rows in ascending or descending order, list null values before all other values.

NULLS LAST
When ordering rows in ascending or descending order, list null values after all other values.

ORDER OF *table-designator*
Specifies that the same ordering used in *table-designator* applies to the result table of the subselect. There must be a table reference matching *table-designator* in the FROM clause of the subselect that specifies this clause (SQLSTATE 42703). The ordering that is applied is the same as if the columns of the ORDER BY clause in the nested subselect (or fullselect) were included in the outer subselect (or fullselect), and these columns were specified in place of the ORDER OF clause.

Note that this form is not allowed in a fullselect (other than the degenerative form of a fullselect). For example, the following is not valid:

```
(SELECT C1 FROM T1
   ORDER BY C1)
```

```
UNION
SELECT C1 FROM T2
   ORDER BY ORDER OF T1
```

The following example *is* valid:

```
SELECT C1 FROM
   (SELECT C1 FROM T1
      UNION
    SELECT C1 FROM T2
    ORDER BY C1 ) AS UTABLE
ORDER BY ORDER OF UTABLE
```

INPUT SEQUENCE
Specifies that, for an INSERT statement, the result table will reflect the input order of ordered data rows. INPUT SEQUENCE ordering can only be specified if an INSERT statement is used in a FROM clause (SQLSTATE 428G4). See "table-reference" on page 644. If INPUT SEQUENCE is specified and the input data is not ordered, the INPUT SEQUENCE clause is ignored.

Notes

- **Column names in sort keys**:
  - The column name is qualified.

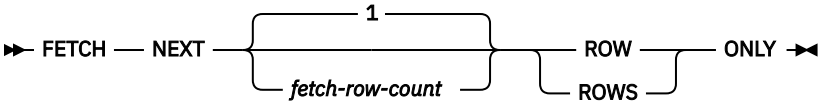
    The query must be a *subselect* (SQLSTATE 42877). The column name must unambiguously identify a column of some table, view or nested table in the FROM clause of the subselect (SQLSTATE 42702). The value of the column is used to compute the value of the sort specification.
  - The column name is unqualified.
    - The query is a subselect.

      If the column name is identical to the name of more than one column of the result table, the column name must unambiguously identify a column of some table, view or nested table in the FROM clause of the ordering subselect (SQLSTATE 42702). If the column name is identical to one column, that column is used to compute the value of the sort specification. If the column name is not identical to a column of the result table, then it must unambiguously identify a column of some table, view or nested table in the FROM clause of the fullselect in the select-statement (SQLSTATE 42702).
    - The query is not a subselect (it includes set operations such as union, except or intersect).

      The column name must not be identical to the name of more than one column of the result table (SQLSTATE 42702). The column name must be identical to exactly one column of the result table (SQLSTATE 42707), and this column is used to compute the value of the sort specification.
- **Limits**: The use of a *sort-key-expression* or a *simple-column-name* where the column is not in the select list might result in the addition of the column or expression to the temporary table used for sorting. This might result in reaching the limit of the number of columns in a table or the limit on the size of a row in a table. Exceeding these limits will result in an error if a temporary table is required to perform the sorting operation.

fetch-clause

The *fetch-clause* sets a maximum number of rows that can be retrieved.



The *fetch-clause* sets a maximum number of rows that can be retrieved. Use this clause to communicate to the database manager that the application is designed in such a way that it is not to retrieve more than *fetch-row-count* rows, regardless of how many rows there are in the intermediate result table. An attempt to fetch beyond *fetch-row-count* rows is handled the same way as normal end of data.

Determining a predictable set of rows to retrieve requires the specification of an ORDER BY clause with sort keys that would uniquely identify the sort order of each row in the intermediate result table. If the intermediate result table includes duplicate sort keys for some rows, the order of these rows is not deterministic. If there is no ORDER BY clause, the intermediate result table is not in a deterministic order. If the order of the intermediate result table is not deterministic, the set of rows that are retrieved is unpredictable.

fetch-row-count
An expression that specifies the maximum number of rows to retrieve. The expression must not contain a column reference, a scalar-fullselect, a function that is not deterministic, a function that has an external action, or a sequence reference (SQLSTATE 428H7). The numeric value must be a positive number or zero (SQLSTATE 2201W). If the data type of the expression is not BIGINT, the result of the expression is cast to a BIGINT value.

Use of the *fetch-clause* with a constant for *fetch-row-count* that is not greater than the maximum large integer influences query optimization of the subselect or fullselect. This influence on query optimization is based on the fact that, at most, a known number of rows will be retrieved. The database manager uses the *integer* from the *optimize-for-clause* to influence query optimization of the outermost fullselect if the following clauses are specified:

- The *fetch-clause* is specified in the outermost fullselect
- The *optimize-for-clause* is specified for the select statement

Limiting the result table to a specified number of rows can improve performance. In some cases, the database manager ceases to process the query when it has determined the specified number of rows. If the *offset-clause* is also specified with a constant for *offset-row-count*, the constant offset value is also considered when a determination is made to cease processing.

If the fullselect contains an SQL data change statement in the FROM clause, all the rows are modified regardless of the limit on the number of rows to fetch.

Notes

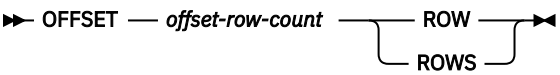
- The keywords FIRST and NEXT can be used interchangeably. The result is unchanged; however, using the keyword NEXT is more readable when the *offset-clause* is used.
- The keywords ROW and ROWS can be used interchangeably. The result is unchanged, however using ROWS is more readable when associated with a number of rows other than 1.
- ***Syntax alternatives***: The following are supported for compatibility with SQL used by other database products. These alternatives are non-standard and should not be used.

Table 113. Syntax alternatives

Alternative syntax	**Equivalent syntax**
LIMIT *x*	FETCH FIRST *x* ROWS ONLY
LIMIT *x* OFFSET *y*	OFFSET *y* ROWS FETCH NEXT *x* ROWS ONLY
LIMIT *y*, *x*	OFFSET *y* ROWS FETCH NEXT *x* ROWS ONLY

offset-clause

The *offset-clause* sets the number of rows to skip.



The *offset-clause* specifies the number of rows to skip before any rows are retrieved. Use this clause to communicate to the database manager that the application does not start retrieving rows until *offset-row-count* rows are skipped. If *offset-clause* is not specified, the default is equivalent to OFFSET 0 ROWS. An

attempt to skip more rows than the number of rows in the intermediate result table is handled the same way as an empty result table.

Determining a predictable set of rows to skip requires the specification of an ORDER BY clause with sort keys that would uniquely identify the sort order of each row in the intermediate result table. If the intermediate result table includes duplicate sort keys for some rows, the order of these rows is not deterministic. If there is no ORDER BY clause, the intermediate result table is not in a deterministic order. If the order of the intermediate result table is not deterministic, the set of skipped rows is unpredictable.

offset-row-count
An expression that specifies the number of rows to skip before any rows are retrieved. The expression must not contain a column reference, a scalar-fullselect, a function that is not deterministic, a function that has an external action, or a sequence reference (SQLSTATE 428H7). The numeric value must be a positive number or zero (SQLSTATE 2201X). If the data type of the expression is not BIGINT, the result of the expression is cast to a BIGINT value.

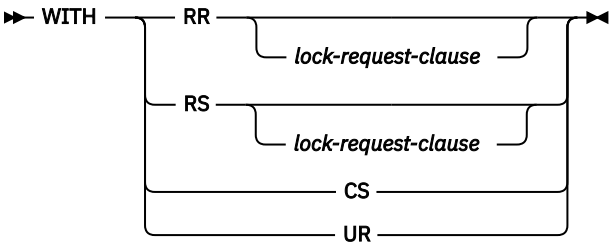
If the fullselect contains an SQL data change statement in the FROM clause, all the rows are modified regardless of the number of rows to skip.

Notes

- The keywords ROW and ROWS can be used interchangeably. The result is unchanged; however, using ROWS is more readable when associated with a number of rows other than 1.
- ***Syntax alternatives***: See the Notes entry that is associated with the fetch-clause for alternative syntax to set the number of rows to skip when the maximum number of rows to retrieve is specified.

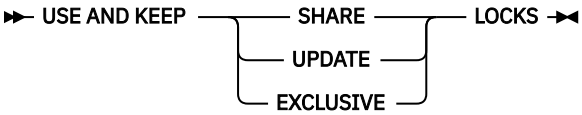
isolation-clause (subselect query)

The optional *isolation-clause* specifies the isolation level at which the subselect or fullselect is run, and whether a specific type of lock is to be acquired.



- RR - Repeatable Read
- RS - Read Stability
- CS - Cursor Stability
- UR - Uncommitted Read

lock-request-clause



The *lock-request-clause* applies only to queries and to positioning read operations within an insert, update, or delete operation. The insert, update, and delete operations themselves will run using locking determined by the database manager.

The optional *lock-request-clause* specifies the type of lock that the database manager is to acquire and hold:

SHARE
Concurrent processes can acquire SHARE or UPDATE locks on the data.

UPDATE
Concurrent processes can acquire SHARE locks on the data, but no concurrent process can acquire an UPDATE or EXCLUSIVE lock.

EXCLUSIVE
Concurrent processes cannot acquire a lock on the data.

***isolation-clause* restrictions**:

- The *isolation-clause* is not supported for a CREATE TABLE or ALTER TABLE statement (SQLSTATE 42601).
- The *isolation-clause* cannot be specified for a subselect or fullselect that will cause trigger invocation, referential integrity scans, or MQT maintenance (SQLSTATE 42601).
- A subselect or fullselect cannot include a *lock-request-clause* if that subselect or fullselect references any SQL functions that are not declared with the option INHERIT ISOLATION LEVEL WITH LOCK REQUEST (SQLSTATE 42601).
- A subselect or fullselect that contains a *lock-request-clause* are not be eligible for MQT routing.
- If an *isolation-clause* is specified for a *subselect* or *fullselect* within the body of an SQL function, SQL method, or trigger, the clause is ignored and a warning is returned.
- If an *isolation-clause* is specified for a *subselect* or *fullselect* that is used by a scrollable cursor, the clause is ignored and a warning is returned.
- Neither *isolation-clause* nor *lock-request-clause* can be specified in the context where they will cause conflict isolation or lock intent on a common table expression (SQLSTATE 42601). This restriction does not apply to aliases or base tables. The following examples create an isolation conflict on *a* and returns an error:
  - View:

    ```
    create view a as (...);
    (select * from a with RR USE AND KEEP SHARE LOCKS)
    UNION ALL
    (select * from a with UR);
    ```

  - Common table expression:

    ```
    WITH a as (...)
    (select * from a with RR USE AND KEEP SHARE LOCKS)
    UNION ALL
    (select * from a with UR);
    ```

- An *isolation-clause* cannot be specified in an XML context (SQLSTATE 2200M).
- The WITH clause specifying isolation cannot be specified at a subselect level in any statement that accesses a column-organized table (SQLSTATE 42858).
- ISOLATION LEVEL UR could behave differently on Column-organized tables and Row-oranized tables.

Examples of subselect queries

The following examples illustrate the susbelect query.

- Example 1 - Select all columns and rows from the EMPLOYEE table.

  ```
  SELECT * FROM EMPLOYEE
  ```

- Example 2 - Join the EMP_ACT and EMPLOYEE tables, select all the columns from the EMP_ACT table and add the employee's surname (LASTNAME) from the EMPLOYEE table to each row of the result.

  ```
  SELECT EMP_ACT.*, LASTNAME
    FROM EMP_ACT, EMPLOYEE
    WHERE EMP_ACT.EMPNO = EMPLOYEE.EMPNO
  ```

- Example 3 - Join the EMPLOYEE and DEPARTMENT tables, select the employee number (EMPNO), employee surname (LASTNAME), department number (WORKDEPT in the EMPLOYEE table, and DEPTNO in the DEPARTMENT table) and department name (DEPTNAME) of all employees who were born (BIRTHDATE) earlier than 1955.

```
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
   FROM EMPLOYEE, DEPARTMENT
   WHERE WORKDEPT = DEPTNO
   AND YEAR(BIRTHDATE) < 1955
```

- Example 4 - Select the job (JOB) and the minimum and maximum salaries (SALARY) for each group of rows with the same job code in the EMPLOYEE table, but only for groups with more than one row and with a maximum salary greater than or equal to 27000.

```
 SELECT JOB, MIN(SALARY), MAX(SALARY)
   FROM EMPLOYEE
   GROUP BY JOB
   HAVING COUNT(*) > 1
   AND MAX(SALARY) >= 27000
```

- Example 5 - Select all the rows of EMP_ACT table for employees (EMPNO) in department (WORKDEPT) 'E11'. (Employee department numbers are shown in the EMPLOYEE table.)

```
SELECT *
  FROM EMP_ACT
  WHERE EMPNO IN
           (SELECT EMPNO
               FROM EMPLOYEE
               WHERE WORKDEPT = 'E11')
```

- Example 6 - From the EMPLOYEE table, select the department number (WORKDEPT) and maximum departmental salary (SALARY) for all departments whose maximum salary is less than the average salary for all employees.

```
SELECT WORKDEPT, MAX(SALARY)
  FROM EMPLOYEE
  GROUP BY WORKDEPT
  HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                           FROM EMPLOYEE)
```

  The subquery in the HAVING clause is run once in this example.

- Example 7 - Using the EMPLOYEE table, select the department number (WORKDEPT) and maximum departmental salary (SALARY) for all departments whose maximum salary is less than the average salary in all other departments.

```
SELECT WORKDEPT, MAX(SALARY)
   FROM EMPLOYEE EMP_COR
    GROUP BY WORKDEPT
    HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                             FROM EMPLOYEE
                             WHERE NOT WORKDEPT = EMP_COR.WORKDEPT)
```

  In contrast to Example 6, the subquery in the HAVING clause is run for each group.

- Example 8 - Determine the employee number and salary of sales representatives along with the average salary and head count of their departments.

  This query must first create a nested table expression (DINFO) to get the AVGSALARY and EMPCOUNT columns, and the DEPTNO column that is used in the WHERE clause.

```
SELECT THIS_EMP.EMPNO, THIS_EMP.SALARY, DINFO.AVGSALARY, DINFO.EMPCOUNT
 FROM EMPLOYEE THIS_EMP,
       (SELECT OTHERS.WORKDEPT AS DEPTNO,
                          AVG(OTHERS.SALARY) AS AVGSALARY,
                          COUNT(*) AS EMPCOUNT
         FROM EMPLOYEE OTHERS
         GROUP BY OTHERS.WORKDEPT
        ) AS DINFO
 WHERE THIS_EMP.JOB = 'SALESREP'
  AND THIS_EMP.WORKDEPT = DINFO.DEPTNO
```

Using a nested table expression for this case saves the processing resources of creating the DINFO view as a regular view. During statement preparation, accessing the catalog for the view is avoided. Because of the context of the rest of the query, only the rows for the department of the sales representatives are considered by the view.

- Example 9 - Display the average education level and salary for five random groups of employees.

  This query requires the use of a nested table expression to set a random value for each employee so that it can later be used in the GROUP BY clause.

```
SELECT RANDID , AVG(EDLEVEL), AVG(SALARY)
  FROM ( SELECT EDLEVEL, SALARY, INTEGER(RAND()*5) AS RANDID
           FROM EMPLOYEE
       ) AS EMPRAND
  GROUP BY RANDID
```

- Example 10 - Query the EMP_ACT table and return those project numbers that have an employee whose salary is in the top 10 of all employees.

```
SELECT EMP_ACT.EMPNO,PROJNO
   FROM EMP_ACT
   WHERE EMP_ACT.EMPNO IN
       (SELECT EMPLOYEE.EMPNO
        FROM EMPLOYEE
        ORDER BY SALARY DESC
        FETCH FIRST 10 ROWS ONLY)
```

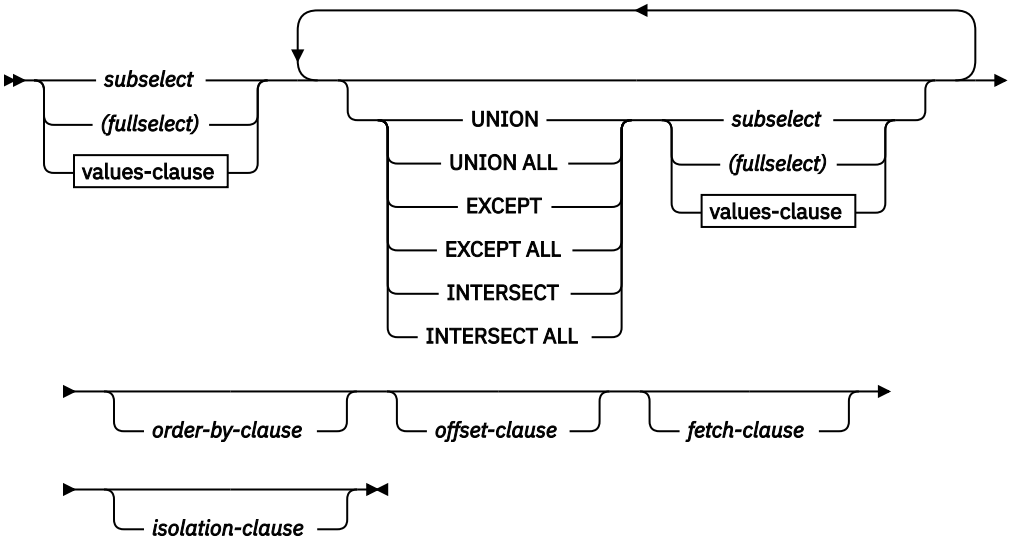
- Example 11 - Assuming that PHONES and IDS are two SQL variables with array values of the same cardinality, turn these arrays into a table with three columns (one for each array and one for the position), and one row per array element.

```
 SELECT T.PHONE, T.ID, T.INDEX FROM UNNEST(PHONES, IDS)
   WITH ORDINALITY AS T(PHONE, ID, INDEX)
   ORDER BY T.INDEX
```

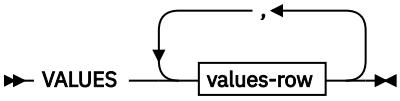
fullselect

The *fullselect* is a component of the select-statement, the INSERT statement, and the CREATE VIEW statement. It is also a component of certain predicates which, in turn, are components of a statement.

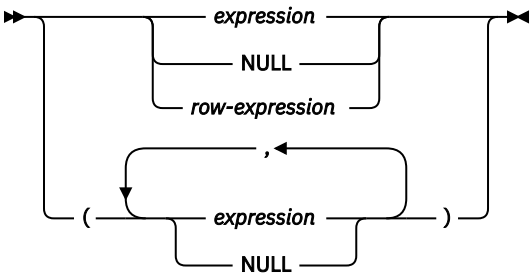
A fullselect that is a component of a predicate is called a *subquery*, and a fullselect that is enclosed in parentheses is sometimes called a subquery.



values-clause



values-row



The set operators UNION, EXCEPT, and INTERSECT correspond to the relational operators union, difference, and intersection.

A fullselect specifies a result table. If a set operator is not used, the result of the fullselect is the result of the specified subselect or values-clause.

The authorization for a *fullselect* is described in the Authorization section in "SQL queries".

values-clause

Derives a result table by specifying the actual values, using expressions or row expressions, for each column of a row in the result table. Multiple rows may be specified. If multiple rows are specified, the extended indicator variable values of DEFAULT and UNASSIGNED must not be used (SQLSTATE 22539). The result type of any expression in the *values-clause* cannot be a row type (SQLSTATE 428H2).

NULL can only be used with multiple specifications of *values-row*, either as the column value of a single column result table or within a *row-expression*, and at least one row in the same column must not be NULL (SQLSTATE 42608).

A *values-row* is specified by:

- A single expression for a single column result table
- *n* expressions (or NULL) separated by commas and enclosed in parentheses, where *n* is the number of columns in the result table or, a row expression for a multiple column result table.

A multiple row VALUES clause must have the same number of columns in each *values-row* (SQLSTATE 42826).

The following examples show *values-clause* and their meaning.

```
VALUES (1),(2),(3)          - 3 rows of 1 column
VALUES 1, 2, 3              - 3 rows of 1 column
VALUES (1, 2, 3)            - 1 row of 3 columns
VALUES (1,21),(2,22),(3,23) - 3 rows of 2 columns
```

A *values-clause* that is composed of *n* specifications of *values-row*, RE_1 to RE_n, where *n* is greater than 1, is equivalent to:

```
RE1 UNION ALL RE2 ... UNION ALL REn
```

This means that the corresponding columns of each *values-row* must be comparable (SQLSTATE 42825).

UNION or UNION ALL

Derives a result table by combining two other result tables (R1 and R2). If UNION ALL is specified, the result consists of all rows in R1 and R2. If UNION is specified without the ALL option, the result is the set of all rows in either R1 or R2, with the duplicate rows eliminated. In either case, however, each row of the UNION table is either a row from R1 or a row from R2.

EXCEPT or EXCEPT ALL
Derives a result table by combining two other result tables (R1 and R2). If EXCEPT ALL is specified, the result consists of all rows that do not have a corresponding row in R2, where duplicate rows are significant. If EXCEPT is specified without the ALL option, the result consists of all rows that are only in R1, with duplicate rows in the result of this operation eliminated.

For compatibility with other SQL implementations, MINUS can be specified as a synonym for EXCEPT.

INTERSECT or INTERSECT ALL
Derives a result table by combining two other result tables (R1 and R2). If INTERSECT ALL is specified, the result consists of all rows that are in both R1 and R2. If INTERSECT is specified without the ALL option, the result consists of all rows that are in both R1 and R2, with the duplicate rows eliminated.

The expression that corresponds to the nth column in R1 and R2 can reference columns with column masks. The nth column of the result of the set operation can be derived from the masked values in R1 or R2.

With the set operation, the elimination of the duplicate rows is based on the unmasked values in R1 and R2. Because all rows are from R1 or R2, the output values in the result table of the set operation may vary when one or more of the following conditions occur:

- The expression corresponding to the nth column in R1 references columns with column masks, but the expression corresponding to the nth column in R2 does not. The opposite is also true.
- The expressions corresponding to the nth column in R1 and R2 reference columns with different column masks.
- The column mask definition references columns that are not the same target column for which the column mask is defined, and those columns are not part of the set operation. It is recommended that the column mask definition does not reference other columns from the target table.

For example, a row in R1 is derived from the masked value, and a row in R2 is derived from the unmasked value. If the row in the result table is from R1, the masked value is returned. If the row in the result table is from R2, the unmasked value is returned.

order-by-clause
Refer to "subselect" on page 639 for details of the *order-by-clause*. A fullselect that contains an ORDER BY clause cannot be specified in (SQLSTATE 428FJ):

- A materialized query table
- The outermost fullselect of a view

Note: An ORDER BY clause in a fullselect does not affect the order of the rows that are returned by a query. An ORDER BY clause affects only the order of the rows that are returned if it is specified in the outermost fullselect. Specify an *order-by-clause* to ensure a predictable order for determining the set of rows from the fullselect if the *offset-clause* or *fetch-clause* are specified.

offset-clause
Refer to "subselect" on page 639 for details of the *offset-clause*. A fullselect that contains an OFFSET clause cannot be specified in the following situations (SQLSTATE 428FJ):

- A materialized query table
- The outermost fullselect of a view

fetch-clause
Refer to "subselect" on page 639 for details of the *fetch-clause*. A fullselect that contains a FETCH clause cannot be specified in the following situations (SQLSTATE 428FJ):

- A materialized query table
- The outermost fullselect of a view

isolation-clause

Refer to "subselect" on page 639 for details of the *isolation-clause*. If *isolation-clause* is specified for a fullselect and it could apply equally to a subselect of the fullselect, *isolation-clause* is applied to the fullselect. For example, consider the following query.

```
SELECT NAME FROM PRODUCT
UNION
SELECT NAME FROM CATALOG
WITH UR
```

Even though the isolation clause WITH UR could apply only to the subselect SELECT NAME FROM CATALOG, it is applied to the whole fullselect.

The number of columns in the result tables R1 and R2 must be the same (SQLSTATE 42826). If the ALL keyword is not specified, R1 and R2 must not include any columns having a data type of CLOB, DBCLOB, BLOB, distinct type on any of these types, or structured type (SQLSTATE 42907).

The column name of the *n*th column of the result table is the name of the *n*th column of R1 if it is named. Otherwise, the *n*th column of the result table is unnamed. If the fullselect is used as a select-statement, a generated name is provided when the statement is described. The generated name cannot be used in other parts of the SQL statement such as the ORDER BY clause or the UPDATE clause. The generated name can be determined by performing a DESCRIBE of the SQL statement and consulting the SQLNAME field.

Duplicate rows: Two rows are duplicates if each value in the first is equal to the corresponding value of the second. For determining duplicates, two null values are considered equal, and two decimal floating-point representations of the same number are considered equal. For example, 2.00 and 2.0 have the same value (2.00 and 2.0 compare as equal) but have different exponents, which allows you to represent both 2.00 and 2.0. So, for example, if the result table of a UNION operation contains a decimal floating-point column and multiple representations of the same number exist, the one that is returned (for example, 2.00 or 2.0) is unpredictable. For more information, see "Numeric comparisons" on page 65.

When multiple operations are combined in an expression, operations within parentheses are performed first. If there are no parentheses, the operations are performed from left to right with the exception that all INTERSECT operations are performed before UNION or EXCEPT operations.

In the following example, the values of tables R1 and R2 are shown on the left. The other headings listed show the values as a result of various set operations on R1 and R2.

R1	R2	UNION ALL	UNION	EXCEPT ALL	EXCEPT	INTER-SECT ALL	INTER-SECT
1	1	1	1	1	2	1	1
1	1	1	2	2	5	1	3
1	3	1	3	2		3	4
2	3	1	4	2		4	
2	3	1	5	4			
2	3	2		5			
3	4	2					
4		2					
4		3					
5		3					
		3					
		3					
		3					

R1	R2	UNION ALL	UNION	EXCEPT ALL	EXCEPT	INTER-SECT ALL	INTER-SECT
		4					
		4					
		4					
		5					

Examples of fullselect queries

The following examples illustrate fullselect queries.

- *Example 1:* Select all columns and rows from the EMPLOYEE table.

```
SELECT * FROM EMPLOYEE
```

- *Example 2:* List the employee numbers (EMPNO) of all employees in the EMPLOYEE table whose department number (WORKDEPT) either begins with 'E' **or** who are assigned to projects in the EMP_ACT table whose project number (PROJNO) equals 'MA2100', 'MA2110', or 'MA2112'.

```
SELECT EMPNO
   FROM EMPLOYEE
   WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO
   FROM EMP_ACT
   WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

- *Example 3:* Make the same query as in example 2, and, in addition, "tag" the rows from the EMPLOYEE table with 'emp' and the rows from the EMP_ACT table with 'emp_act'. Unlike the result from example 2, this query might return the same EMPNO more than once, identifying which table it came from by the associated "tag".

```
SELECT EMPNO, 'emp'
   FROM EMPLOYEE
   WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO, 'emp_act' FROM EMP_ACT
   WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

- *Example 4:* Make the same query as in example 2, only use UNION ALL so that no duplicate rows are eliminated.

```
SELECT EMPNO
    FROM EMPLOYEE
    WHERE WORKDEPT LIKE 'E%'
UNION ALL
SELECT EMPNO
   FROM EMP_ACT
   WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

- *Example 5:* Make the same query as in Example 3, only include an additional two employees currently not in any table and tag these rows as "new".

```
SELECT EMPNO, 'emp'
   FROM EMPLOYEE
   WHEREWORKDEPTLIKE 'E%'
UNION
SELECT EMPNO, 'emp_act'
   FROM EMP_ACT
   WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
UNION
   VALUES ('NEWAAA', 'new'), ('NEWBBB', 'new')
```

- *Example 6:* This example of EXCEPT produces all rows that are in T1 but not in T2.

```
(SELECT * FROM T1)
EXCEPT ALL
(SELECT * FROM T2)
```

If no null values are involved, this example returns the same results as

```
SELECT ALL *
   FROM T1
   WHERE NOT EXISTS (SELECT * FROM T2
                     WHERE T1.C1 = T2.C1 AND T1.C2 = T2.C2 AND...)
```

- *Example 7:* This example of INTERSECT produces all rows that are in both tables T1 and T2, removing duplicates.

```
(SELECT * FROM T1)
INTERSECT
(SELECT * FROM T2)
```

If no null values are involved, this example returns the same result as

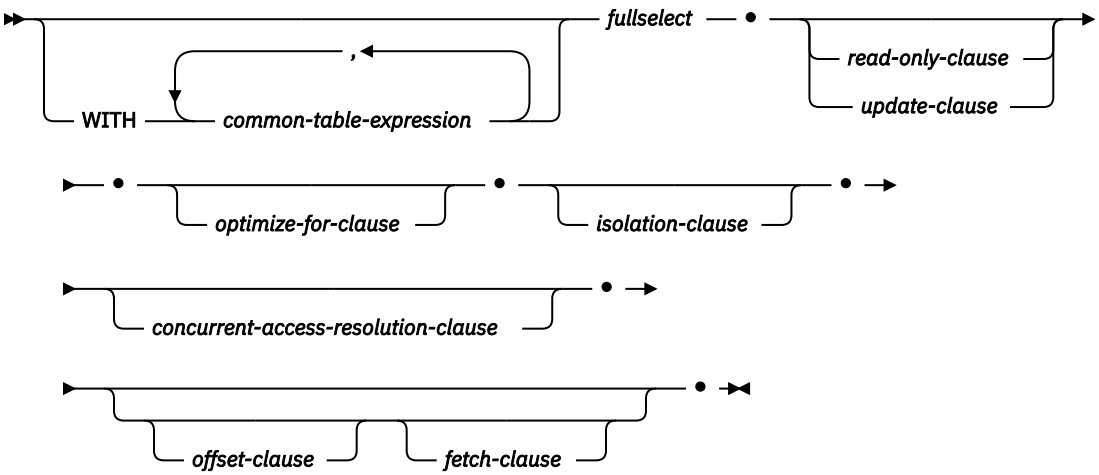
```
SELECT DISTINCT * FROM T1
   WHERE EXISTS (SELECT * FROM T2
                 WHERE T1.C1 = T2.C1 AND T1.C2 = T2.C2 AND...)
```

where C1, C2, and so on represent the columns of T1 and T2.

select-statement

The *select-statement* is the form of a query that can be specified in a DECLARE CURSOR statement, either directly, or prepared and then referenced. It can also be issued through the use of dynamic SQL statements, causing a result table to be displayed on the user's screen.

The table specified by a *select-statement* is the result of the fullselect.



The authorization for a *select-statement* is described in the Authorization section in "SQL queries".

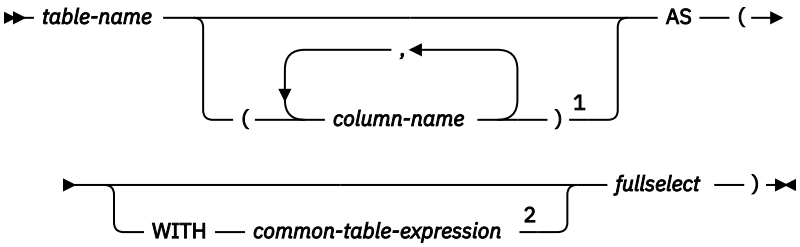
For details about the clauses in the select-statement query, refer to the following topics:

- “common-table-expression” on page 716
- “update-clause” on page 720
- “read-only-clause” on page 721
- “optimize-for-clause” on page 721
- “isolation-clause (select-statement query)” on page 722
- “lock-request-clause” on page 722

- "concurrent-access-resolution-clause" on page 722
- "offset-clause" on page 706
- "fetch-clause" on page 705
- "Retrieval of result sets from an SQL data change statement" on page 725

common-table-expression

A *common table expression* permits defining a result table with a *table-name* that can be specified as a table name in any FROM clause of the fullselect that follows.



Notes:

[1] If a common table expression is recursive, or if the fullselect results in duplicate column names, column names must be specified.

[2] For Netezza compatibility, the WITH clause is supported in common table expressions. This SQL compatibility enhancement is only available in Db2 Version 11.5 Mod Pack 2 and later versions.

Multiple common table expressions can be specified following the single WITH keyword. Each common table expression specified can also be referenced by name in the FROM clause of subsequent common table expressions.

If a list of columns is specified, it must consist of as many names as there are columns in the result table of the fullselect. Each *column-name* must be unique and unqualified. If these column names are not specified, the names are derived from the select list of the fullselect used to define the common table expression.

The *table-name* of a common table expression must be different from any other common table expression *table-name* in the same statement (SQLSTATE 42726). If the common table expression is specified in an INSERT statement the *table-name* cannot be the same as the table or view name that is the object of the insert (SQLSTATE 42726). A common table expression *table-name* can be specified as a table name in any FROM clause throughout the fullselect. A *table-name* of a common table expression overrides any existing table, view or alias (in the catalog) with the same qualified name.

If more than one common table expression is defined in the same statement, cyclic references between the common table expressions are not permitted (SQLSTATE 42835). A *cyclic reference* occurs when two common table expressions *dt1* and *dt2* are created such that *dt1* refers to *dt2* and *dt2* refers to *dt1*.

If the fullselect of a common table expression contains a *data-change-table-reference* in the FROM clause, the common table expression is said to modify data. A common table expression that modifies data is always evaluated when the statement is processed, regardless of whether the common table expression is used anywhere else in the statement. If there is at least one common table expression that reads or modifies data, all common table expressions are processed in the order in which they occur, and each common table expression that reads or modifies data is completely executed, including all constraints and triggers, before any subsequent common table expressions are executed.

The common table expression is also optional before to the fullselect in the CREATE VIEW and INSERT statements.

A common table expression can be used in the following situations:

- In place of a view to avoid creating the view (when general use of the view is not required and positioned updates or deletes are not used)

- To enable grouping by a column that is derived from a scalar subselect or function that is not deterministic or has external action
- When the required result table is based on host variables
- When the same result table must be shared in a fullselect
- When the result must be derived using recursion
- When multiple SQL data change statements must be processed within the query

If the fullselect of a common table expression contains a reference to itself in a FROM clause, the common table expression is a *recursive common table expression*. Queries using recursion are useful in supporting applications such as bill of materials (BOM), reservation systems, and network planning.

The following must be true of a recursive common table expression:

- Each fullselect that is part of the recursion cycle must start with SELECT or SELECT ALL. Use of SELECT DISTINCT is not allowed (SQLSTATE 42925). Furthermore, the unions must use UNION ALL (SQLSTATE 42925).
- The column names must be specified following the *table-name* of the common table expression (SQLSTATE 42908).
- The first fullselect of the first union (the initialization fullselect) must not include a reference to any column of the common table expression in any FROM clause (SQLSTATE 42836).
- If a column name of the common table expression is referred to in the iterative fullselect, the data type, length, and code page for the column are determined based on the initialization fullselect. The corresponding column in the iterative fullselect must have the same data type and length as the data type and length determined based on the initialization fullselect and the code page must match (SQLSTATE 42825). However, for character string types, the length of the two data types can differ. In this case, the column in the iterative fullselect must have a length that will always be assignable to the length determined from the initialization fullselect.
- Each fullselect that is part of the recursion cycle must not include any aggregate functions, group-by-clauses, or having-clauses (SQLSTATE 42836).

  The FROM clauses of these fullselects can include at most one reference to a common table expression that is part of a recursion cycle (SQLSTATE 42836).
- The iterative fullselect and the overall recursive fullselect must not include an order-by-clause (SQLSTATE 42836).
- Subqueries (scalar or quantified) must not be part of any recursion cycles (SQLSTATE 42836).

When developing recursive common table expressions, remember that an infinite recursion cycle (loop) can be created. Check that recursion cycles will stop. This is especially important if the data involved is cyclic. A recursive common table expression is expected to include a predicate that will prevent an infinite loop. The recursive common table expression is expected to include:

- In the iterative fullselect, an integer column incremented by a constant.
- A predicate in the where clause of the iterative fullselect in the form "counter_col < constant" or "counter _col < :hostvar".

A warning is issued if this syntax is not found in the recursive common table expression (SQLSTATE 01605).

Recursion example: bill of materials

Bill of materials (BOM) applications are a common requirement in many business environments. To illustrate the capability of a recursive common table expression for BOM applications, consider a table of parts with associated subparts and the quantity of subparts required by the part.

For this example, create the table as follows:

```
CREATE TABLE PARTLIST
               (PART VARCHAR(8),
                SUBPART VARCHAR(8),
                QUANTITY INTEGER);
```

To give query results for this example, assume that the PARTLIST table is populated with the following values:

```
PART     SUBPART  QUANTITY
-------- -------- -----------
00       01                5
00       05                3
01       02                2
01       03                3
01       04                4
01       06                3
02       05                7
02       06                6
03       07                6
04       08               10
04       09               11
05       10               10
05       11               10
06       12               10
06       13               10
07       14                8
07       12                8
```

Example 1: Single level explosion

The first example is called single level explosion. It answers the question, "What parts are needed to build the part identified by '01'?". The list will include the direct subparts, subparts of the subparts and so on. However, if a part is used multiple times, its subparts are only listed once.

```
WITH RPL (PART, SUBPART, QUANTITY) AS
     (  SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
        FROM PARTLIST ROOT
        WHERE ROOT.PART = '01'
      UNION ALL
        SELECT CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
        FROM RPL PARENT, PARTLIST CHILD
        WHERE  PARENT.SUBPART = CHILD.PART
     )
SELECT DISTINCT PART, SUBPART, QUANTITY
 FROM RPL
  ORDER BY PART, SUBPART, QUANTITY;
```

The preceding query includes a common table expression, identified by the name *RPL*, that expresses the recursive part of this query. It illustrates the basic elements of a recursive common table expression.

The first operand (fullselect) of the UNION, referred to as the *initialization fullselect*, gets the direct children of part '01'. The FROM clause of this fullselect refers to the source table and will never reference itself (*RPL* in this case). The result of this first fullselect goes into the common table expression *RPL* (Recursive PARTLIST). As in this example, the UNION must always be a UNION ALL.

The second operand (fullselect) of the UNION uses *RPL* to compute subparts of subparts by having the FROM clause refer to the common table expression *RPL* and the source table with a join of a part from the source table (child) to a subpart of the current result contained in *RPL* (parent). The result goes back to *RPL* again. The second operand of UNION is then used repeatedly until no more children exist.

The SELECT DISTINCT in the main fullselect of this query ensures the same part/subpart is not listed more than once.

The result of the query is as follows:

```
PART     SUBPART  QUANTITY
-------- -------- -----------
01       02                2
01       03                3
01       04                4
01       06                3
02       05                7
02       06                6
03       07                6
04       08               10
04       09               11
05       10               10
```

```
  05       11               10
  06       12               10
  06       13               10
  07       12                8
  07       14                8
```

Observe in the result that part '01' goes to '02' which goes to '06' and so on. Further, notice that part '06' is reached twice, once through '01' directly and another time through '02'. In the output, however, its subcomponents are listed only once (this is the result of using a SELECT DISTINCT) as required.

It is important to remember that with recursive common table expressions it is possible to introduce an *infinite loop*. In this example, an infinite loop would be created if the search condition of the second operand that joins the parent and child tables was coded as:

```
   PARENT.SUBPART = CHILD.SUBPART
```

This example of causing an infinite loop is obviously a case of not coding what is intended. Exercise care when determining what to code so that there is a definite end of the recursion cycle.

The result produced by this example query can be produced in an application program without using a recursive common table expression. However, this approach would require starting of a new query for every level of recursion. Furthermore, the application needs to put all the results back in the database to order the result. This approach complicates the application logic and does not perform well. The application logic becomes even harder and more inefficient for other bill of material queries, such as summarized and indented explosion queries.

Example 2: Summarized explosion

The second example is a summarized explosion. The question posed here is, what is the total quantity of each part required to build part '01'. The main difference from the single level explosion is the requirement to aggregate the quantities. The first example indicates the quantity of subparts required for the part whenever it is required. It does not indicate how many of the subparts are needed to build part '01'.

```
WITH RPL (PART, SUBPART, QUANTITY) AS
   (
      SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
       FROM PARTLIST ROOT
       WHERE ROOT.PART = '01'
    UNION ALL
      SELECT PARENT.PART, CHILD.SUBPART, PARENT.QUANTITY*CHILD.QUANTITY
       FROM RPL PARENT, PARTLIST CHILD
       WHERE PARENT.SUBPART = CHILD.PART
   )
SELECT PART, SUBPART, SUM(QUANTITY) AS "Total QTY Used"
 FROM RPL
  GROUP BY PART, SUBPART
  ORDER BY PART, SUBPART;
```

In the preceding query, the select list of the second operand of the UNION in the recursive common table expression, identified by the name *RPL*, shows the aggregation of the quantity. To find out how much of a subpart is used, the quantity of the parent is multiplied by the quantity per parent of a child. If a part is used multiple times in different places, it requires another final aggregation. This is done by the grouping over the common table expression *RPL* and using the SUM aggregate function in the select list of the main fullselect.

The result of the query is as follows:

```
  PART     SUBPART  Total Qty Used
  -------- -------- --------------
  01       02                   2
  01       03                   3
  01       04                   4
  01       05                  14
  01       06                  15
  01       07                  18
  01       08                  40
  01       09                  44
```

```
  01       10                  140
  01       11                  140
  01       12                  294
  01       13                  150
  01       14                  144
```

Looking at the output, consider the line for subpart '06'. The total quantity used value of 15 is derived from a quantity of 3 directly for part '01' and a quantity of 6 for part '02' which is needed 2 times by part '01'.

Example 3: Controlling depth

The question might come to mind, what happens when there are more levels of parts in the table than you are interested in for your query? That is, how is a query written to answer the question, "What are the first two levels of parts needed to build the part identified by '01'?" For the sake of clarity in the example, the level is included in the result.

```
WITH RPL (LEVEL, PART, SUBPART, QUANTITY) AS
      (
         SELECT 1,               ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
          FROM PARTLIST ROOT
          WHERE ROOT.PART = '01'
       UNION ALL
         SELECT PARENT.LEVEL+1, CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
          FROM RPL PARENT, PARTLIST CHILD
          WHERE PARENT.SUBPART = CHILD.PART
            AND PARENT.LEVEL < 2
      )
 SELECT PART, LEVEL, SUBPART, QUANTITY
   FROM RPL;
```

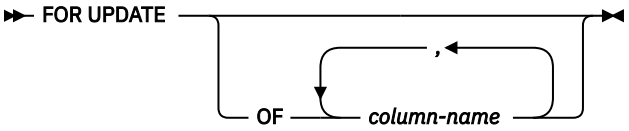
This query is similar to example 1. The column *LEVEL* was introduced to count the levels from the original part. In the initialization fullselect, the value for the *LEVEL* column is initialized to 1. In the subsequent fullselect, the level from the parent is incremented by 1. Then to control the number of levels in the result, the second fullselect includes the condition that the parent level must be less than 2. This ensures that the second fullselect only processes children to the second level.

The result of the query is:

```
  PART     LEVEL       SUBPART  QUANTITY
  -------- ----------- -------- -----------
  01                 1 02                 2
  01                 1 03                 3
  01                 1 04                 4
  01                 1 06                 3
  02                 2 05                 7
  02                 2 06                 6
  03                 2 07                 6
  04                 2 08                10
  04                 2 09                11
  06                 2 12                10
  06                 2 13                10
```

update-clause

The FOR UPDATE clause identifies the columns that can be targets in an assignment clause in a subsequent positioned UPDATE statement. Each *column-name* must be unqualified and must identify a column of the table or view identified in the first FROM clause of the fullselect.



If a FOR UPDATE clause is specified with a *column-name* list, and extended indicator variables are not enabled, then *column-name* must be an updatable column (SQLSTATE 42808).

If a FOR UPDATE clause is specified without a *column-name* list, then the implicit *column-name* list is determined as follows:

- If extended indicator variables are enabled, all of the columns of the table or view identified in the first FROM clause of the fullselect are included.
- If extended indicator variables are not enabled, all of the updatable columns of the table or view identified in the first FROM clause of the fullselect are included.

The FOR UPDATE clause cannot be used if one of the following conditions is true:

- The cursor associated with the select-statement is not deletable .
- One of the selected columns is a non-updatable column of a catalog table and the FOR UPDATE clause has not been used to exclude that column.

read-only-clause

The FOR READ ONLY clause indicates that the result table is read-only and therefore the cursor cannot be referred to in Positioned UPDATE and DELETE statements. FOR FETCH ONLY has the same meaning.

Some result tables are read-only by nature. (For example, a table based on a read-only view.) FOR READ ONLY can still be specified for such tables, but the specification has no effect.

For result tables in which updates and deletes are allowed, specifying FOR READ ONLY (or FOR FETCH ONLY) can possibly improve the performance of FETCH operations by allowing the database manager to do blocking. For example, in programs that contain dynamic SQL statements without the FOR READ ONLY or ORDER BY clause, the database manager might open cursors as if the FOR UPDATE clause were specified. It is recommended, therefore, that the FOR READ ONLY clause be used to improve performance, except in cases where queries will be used in positioned UPDATE or DELETE statements.

A read-only result table must not be referred to in a Positioned UPDATE or DELETE statement, whether it is read-only by nature or specified as FOR READ ONLY (FOR FETCH ONLY).

optimize-for-clause

The OPTIMIZE FOR clause requests special processing of the *select statement*.

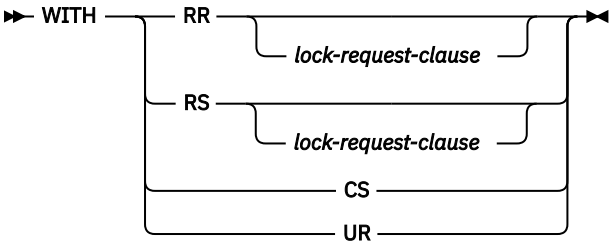
If this clause is omitted, it is assumed that all rows of the result table will be retrieved; if it is specified, it is assumed that the number of rows retrieved will probably not exceed *n*, where *n* is the value of *integer*. The value of *n* must be a positive integer (not zero). Use of the OPTIMIZE FOR clause influences query optimization, based on the assumption that *n* rows will be retrieved. In addition, for cursors that are blocked, this clause will influence the number of rows that will be returned in each block (that is, no more than *n* rows will be returned in each block). If both the *fetch-clause* and the *optimize-for-clause* are specified, the lower of the integer values from these clauses will be used to influence the communications buffer size. The values are considered independently for optimization purposes.

This clause does not limit the number of rows that can be fetched, or affect the result in any other way than performance. Using OPTIMIZE FOR *n* ROWS can improve performance if no more than *n* rows are retrieved, but might degrade performance if more than *n* rows are retrieved.

If the value of *n* multiplied by the size of the row exceeds the size of the communication buffer, the OPTIMIZE FOR clause will have no affect on the data buffers. The size of the communication buffer is defined by the **rqrioblk** or the **aslheapsz** configuration parameter.

isolation-clause (select-statement query)

The optional *isolation-clause* specifies the isolation level at which the statement is executed, and whether a specific type of lock is to be acquired.

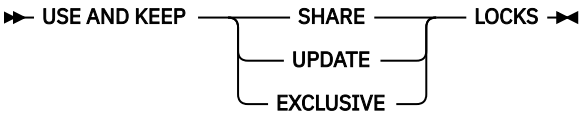


- RR - Repeatable Read (row-organized tables only SQLSTATE 42858)
- RS - Read Stability (row-organized tables only SQLSTATE 42858)
- CS - Cursor Stability
- UR - Uncommitted Read

The default isolation level of the statement is the isolation level of the package in which the statement is bound. When a nickname is used in a *select-statement* to access data in IBM family and Microsoft SQL Server data sources, the *isolation-clause* can be included in the statement to specify the statement isolation level. If the *isolation-clause* is included in statements that access other data sources, the specified isolation level is ignored. The current isolation level on the federated server is mapped to a corresponding isolation level at the data source on each connection to the data source. After a connection is made to a data source, the isolation level cannot be changed for the duration of the connection.

lock-request-clause

The optional *lock-request-clause* specifies the type of lock that the database manager is to acquire and hold.



SHARE
Concurrent processes can acquire SHARE or UPDATE locks on the data.

UPDATE
Concurrent processes can acquire SHARE locks on the data, but no concurrent process can acquire an UPDATE or EXCLUSIVE lock.

EXCLUSIVE
Concurrent processes cannot acquire a lock on the data.

The *lock-request-clause* applies to all base table and index scans required by the query, including those within subqueries, SQL functions and SQL methods. It has no affect on locks placed by procedures, external functions, or external methods. Any SQL function or SQL method called (directly or indirectly) by the statement must be created with INHERIT ISOLATION LEVEL WITH LOCK REQUEST (SQLSTATE 42601). The *lock-request-clause* cannot be used with a modifying query that might activate triggers or that requires referential integrity checks (SQLSTATE 42601).

concurrent-access-resolution-clause

The optional *concurrent-access-resolution-clause* specifies the concurrent access resolution to use for *select-statement*.

WAIT FOR OUTCOME specifies to wait for the commit or rollback when encountering data in the process of being updated or deleted. Rows encountered that are in the process of being inserted are not skipped. The settings for the registry variables DB2_EVALUNCOMMITTED, DB2_SKIPDELETED, and DB2_SKIPINSERTED are ignored. This clause applies when the isolation level is CS or RS. It is ignored when an isolation level of UR or RR is in effect, or when the table is column-organized.

This clause causes the following behavior and settings to be overridden:

- Any higher level setting such as bind options, CLI settings, JDBC settings, or lock modifications.

The SKIP LOCKED DATA clause specifies that rows are skipped when incompatible locks that would block the progress of the statement are held on the rows by other transactions. These rows can belong to any accessed table addressed in the statement, including tables accessed in a subquery. This clause applies when the isolation level is CS or RS and is ignored when an isolation level of UR or RR is in effect. It applies to row and block level locks.

This clause causes the following behavior and settings to be overridden:

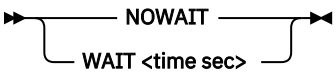
- Any higher level setting such as bind options, CLI settings, JDBC settings, or lock modifications still apply, with the modified behavior that if a lock request is made and there is a conflict, the corresponding row is skipped. SKIP LOCKED DATA specified on a statement does override a higher level WAIT FOR OUTCOME setting.

SKIP LOCKED DATA is ignored if it is specified when WITH RR or WITH UR. The default isolation level of the statement depends on the isolation of the package or plan with which the statement is bound, and whether the result table is read-only. If the default isolation level of the statement is Repeatable Read or Uncommitted Read, then SKIP LOCKED DATA is ignored.

SKIP LOCKED DATA clause is strictly SQL based. Also, it cannot be specified for the following:

- Positioned updates and deletes
- Subquery
- Update/delete statements on column organized table
- **PREPARE** command
- **BIND** command

Attention: The following feature is available in Db2 11.5.6 and later versions.



The NOWAIT and WAIT clauses specify the number of seconds to wait for a lock before returning an error indicating that a lock cannot be obtained.

When using the WAIT clause, `<time sec>` is an integer between -1 and 32767.

Note: For `NOWAIT` and `WAIT 0`, locks are not waited for. If no lock is available at the time of the request, a -911 error is returned.

When a WAIT value of `-1` is specified, lock timeout detection is turned off. In this situation a lock is waited for (if one is not available at the time of the request) until either of the following events occur:

- The lock is granted.
- A deadlock occurs.

Use of the NOWAIT and WAIT clauses overwrites the value of the LOCKTIMEOUT database configuration variable and the value of the CURRENT LOCK TIMEOUT special register for this select statement. This means that adding the NOWAIT/WAIT clause with a wait time value of **t** has

the same effect as executing the select statement with a LOCKTIMEOUT value or CURRENT LOCK TIMEOUT value of **t**.

Examples of select-statement queries

The following examples illustrate the select-statement query.

- *Example 1:* Select all columns and rows from the EMPLOYEE table.

```
SELECT * FROM EMPLOYEE
```

- *Example 2:* Select the project name (PROJNAME), start date (PRSTDATE), and end date (PRENDATE) from the PROJECT table. Order the result table by the end date with the most recent dates appearing first.

```
SELECT PROJNAME, PRSTDATE, PRENDATE
 FROM PROJECT
 ORDER BY PRENDATE DESC
```

- *Example 3:* Select the department number (WORKDEPT) and average departmental salary (SALARY) for all departments in the EMPLOYEE table. Arrange the result table in ascending order by average departmental salary.

```
SELECT WORKDEPT, AVG(SALARY)
   FROM EMPLOYEE
   GROUP BY WORKDEPT
   ORDER BY 2
```

- *Example 4:* Declare a cursor named UP_CUR to be used in a C program to update the start date (PRSTDATE) and the end date (PRENDATE) columns in the PROJECT table. The program must receive both of these values together with the project number (PROJNO) value for each row.

```
EXEC SQL  DECLARE UP_CUR CURSOR FOR
            SELECT PROJNO, PRSTDATE, PRENDATE
              FROM PROJECT
              FOR UPDATE OF PRSTDATE, PRENDATE;
```

- *Example 5:* This example names the expression SAL+BONUS+COMM as TOTAL_PAY

```
 SELECT SALARY+BONUS+COMM AS TOTAL_PAY
   FROM EMPLOYEE
   ORDER BY TOTAL_PAY
```

- *Example 6:* Determine the employee number and salary of sales representatives along with the average salary and head count of their departments. Also, list the average salary of the department with the highest average salary.

  Using a common table expression for this case saves the processing resources of creating the DINFO view as a regular view. During statement preparation, accessing the catalog for the view is avoided and, because of the context of the rest of the fullselect, only the rows for the department of the sales representatives are considered by the view.

```
WITH
   DINFO (DEPTNO, AVGSALARY, EMPCOUNT) AS
       (SELECT OTHERS.WORKDEPT, AVG(OTHERS.SALARY), COUNT(*)
         FROM EMPLOYEE OTHERS
         GROUP BY OTHERS.WORKDEPT
       ),
   DINFOMAX AS
       (SELECT MAX(AVGSALARY) AS AVGMAX FROM DINFO)
SELECT THIS_EMP.EMPNO, THIS_EMP.SALARY,
       DINFO.AVGSALARY, DINFO.EMPCOUNT, DINFOMAX.AVGMAX
 FROM EMPLOYEE THIS_EMP, DINFO, DINFOMAX
 WHERE THIS_EMP.JOB = 'SALESREP'
 AND THIS_EMP.WORKDEPT = DINFO.DEPTNO
```

- *Example 7:* Given two tables, EMPLOYEE and PROJECT, replace employee SALLY with a new employee GEORGE, assign all projects lead by SALLY to GEORGE, and return the names of the updated projects.

```
WITH
  NEWEMP AS (SELECT EMPNO FROM NEW TABLE
               (INSERT INTO EMPLOYEE(EMPNO, FIRSTNME)
                   VALUES(NEXT VALUE FOR EMPNO_SEQ, 'GEORGE'))),
  OLDEMP AS (SELECT EMPNO FROM EMPLOYEE WHERE FIRSTNME = 'SALLY'),
  UPPROJ AS (SELECT PROJNAME FROM NEW TABLE
               (UPDATE PROJECT
                   SET RESPEMP = (SELECT EMPNO FROM NEWEMP)
                   WHERE RESPEMP = (SELECT EMPNO FROM OLDEMP))),
  DELEMP AS (SELECT EMPNO FROM OLD TABLE
               (DELETE FROM EMPLOYEE
                   WHERE EMPNO = (SELECT EMPNO FROM OLDEMP)))
SELECT PROJNAME FROM UPPROJ;
```

- *Example 8:* Retrieve data from the DEPT table. That data will later be updated with a searched update, and will be locked when the query executes.

```
SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DEPT
  WHERE ADMRDEPT ='A00'
  FOR READ ONLY WITH RS USE AND KEEP EXCLUSIVE LOCKS
```

- *Example 9:* Select all columns and rows from the EMPLOYEE table. If another transaction is concurrently updating, deleting, or inserting data in the EMPLOYEE table, the select operation will wait to get the data until after the other transaction is completed.

```
SELECT * FROM EMPLOYEE WAIT FOR OUTCOME
```

Retrieval of result sets from an SQL data change statement

Applications that modify tables with INSERT, UPDATE, or DELETE statements might require additional processing on the modified rows. To facilitate this processing, you can embed SQL data-change operations in the FROM clause of SELECT and SELECT INTO statements.

Within a single unit of work, applications can retrieve a result set containing the modified rows from a table or view modified by an SQL data-change operation.

For example, the following statement updates the salaries of all the records in the EMPLOYEE table in the SAMPLE database and then returns the employee number and new salary for all the updated rows.

```
SELECT empno, salary FROM FINAL TABLE
    (UPDATE employee SET salary = salary * 1.10 WHERE job = 'CLERK')
```

To return data successfully, SELECT statements that retrieve result sets FROM SQL data-change operations require the SQL data-change operations to run successfully. The success of SQL data-change operations includes the processing of all constraints and triggers, if applicable.

For instance, suppose a user with SELECT privileges, but without INSERT privileges on the EMPLOYEE table attempts a SELECT FROM INSERT statement on the EMPLOYEE table. The INSERT operation fails because of the missing privileges, and as a result, the entire SELECT statement fails.

Consider the following query, where records from the EMPLOYEE table are selected and then inserted into a different table, named EMP. This SELECT statement will fail.

```
SELECT empno FROM FINAL TABLE
    (INSERT INTO emp(name, salary)
    SELECT firstnme || midinit || lastname, salary
    FROM employee)
```

If the EMPLOYEE table has 100 rows and row 90 has a SALARY value of $9,999,000.00, then the addition of $10,000.00 would cause a decimal overflow to occur. The overflow would force the database manager to roll back the insertions into the EMP table.

Intermediate result tables

The modified rows of the table or view targeted by an SQL data-change operation in the FROM clause of a SELECT statement compose an intermediate result table. The intermediate result table includes all the columns of the target table or view, in addition to any include columns defined in the SQL data-change operation. You can reference all of the columns in an intermediate result table by name in the select list, the ORDER BY clause, or the WHERE clause.

The contents of the intermediate result table are dependant on the qualifier specified in the FROM clause. You must include one of the following FROM clause qualifiers in SELECT statements that retrieve result sets as intermediate result tables.

OLD TABLE
The rows in the intermediate result table will contain values of the target table rows at the point immediately preceding the execution of before triggers and the SQL data-change operation. The OLD TABLE qualifier applies to UPDATE and DELETE operations.

NEW TABLE
The rows in the intermediate result table will contain values of the target table rows at the point immediately after the SQL data-change statement has been executed, but before referential integrity evaluation and the firing of any after triggers. The NEW TABLE qualifier applies to UPDATE and INSERT operations.

FINAL TABLE
This qualifier returns the same intermediate result table as NEW TABLE. In addition, the use of FINAL TABLE guarantees that no after trigger or referential integrity constraint will further modify the target of the UPDATE or INSERT operation. The FINAL TABLE qualifier applies to UPDATE and INSERT operations.

The FROM clause qualifiers determine what version of the targeted data is in the intermediate result table. These qualifiers do not affect the insertion, deletion, or updates of target table rows.

Target tables and views

When selecting result sets FROM SQL data-change operations, the target can be either a table or a view.

In SQL data-change operations against views, the result table cannot include rows that no longer satisfy the view definition for NEW TABLE and FINAL TABLE. If you embed an INSERT or UPDATE statement that references a view in a SELECT statement, the view must be defined as WITH CASCADED CHECK OPTION. Alternatively, the view must satisfy the restrictions that would allow you to define it as WITH CASCADED CHECK OPTION.

If the target of SQL data-change operations embedded in the FROM clause of a SELECT statement is a fullselect, the result table can include rows that no longer qualify in the fullselect. This is because the predicates in the WHERE clause are not re-evaluated against the updated values.

Result set sorting based on INPUT SEQUENCE

To SELECT rows in the same order as they are inserted into the target table or view, use the INPUT SEQUENCE keywords in the ORDER BY clause. Use of the INPUT SEQUENCE keywords does not force rows to be inserted in the same order they are provided.

The following example demonstrates the use of the INPUT SEQUENCE keywords in the ORDER BY clause to sort the result set of an INSERT operation.

```
CREATE TABLE orders (purchase_date DATE,
                     sales_person VARCHAR(16),
                     region VARCHAR(10),
                     quantity SMALLINT,
                     order_num INTEGER NOT NULL
                       GENERATED ALWAYS AS IDENTITY (START WITH 100,
                                                     INCREMENT BY 1))

SELECT * FROM FINAL TABLE
         (INSERT INTO orders
          (purchase_date, sales_person, region, quantity)
```

```
          VALUES (CURRENT DATE,'Judith','Beijing',6),
                 (CURRENT DATE,'Marieke','Medway',5),
                 (CURRENT DATE,'Hanneke','Halifax',5))
        ORDER BY INPUT SEQUENCE

PURCHASE_DATE SALES_PERSON     REGION     QUANTITY ORDER_NUM
------------- ---------------- ---------- -------- -----------
07/18/2003    Judith           Beijing           6         100
07/18/2003    Marieke          Medway            5         101
07/18/2003    Hanneke          Halifax           5         102
```

You can also sort result set rows using include columns.

SQL statements

This topic contains tables that list the SQL statements classified by type.

- SQL schema statements (Table 114 on page 727)
- SQL data change statements (Table 115 on page 733)
- SQL data statements (Table 116 on page 733)
- SQL transaction statements (Table 117 on page 734)
- SQL connection statements (Table 118 on page 734)
- SQL dynamic statements (Table 119 on page 734)
- SQL session statements (Table 120 on page 735)
- SQL embedded host language statements (Table 121 on page 736)
- SQL control statements (Table 122 on page 737)

Table 114. SQL schema statements

SQL Statement	Purpose
“ALTER AUDIT POLICY ” on page 750	Modifies the definition of an audit policy at the current server.
“ALTER BUFFERPOOL ” on page 752	Changes the definition of a buffer pool.
“ALTER DATABASE ” on page 757	Adds new storage paths to the collection of paths that are used for automatic storage table spaces.
“ALTER EVENT MONITOR ” on page 761	Changes the definition of a TABLE or UNFORMATTED EVENT TABLE event monitor.
“ALTER DATABASE PARTITION GROUP ” on page 754	Changes the definition of a database partition group.
“ALTER FUNCTION ” on page 766	Modifies an existing function by changing the properties of the function.
“ALTER HISTOGRAM TEMPLATE ” on page 769	Modifies the template describing the type of histogram that can be used to override one or more of the default histograms of a service class or a work class.
“ALTER INDEX ” on page 770	Changes the definition of an index.
“ALTER MASK ” on page 771	Changes the definition of a column mask.
“ALTER METHOD ” on page 772	Modifies an existing method by changing the method body associated with the method.
“ALTER MODULE ” on page 773	Changes the definition of a module.
“ALTER NICKNAME ” on page 779	Changes the definition of a nickname.
“ALTER PACKAGE ” on page 788	Alters bind options for a package at the current server without having to bind or rebind the package.

Table 114. SQL schema statements (continued)

SQL Statement	Purpose
“ALTER PERMISSION ” on page 790	Changes the definition of a row permission.
“ALTER PROCEDURE (external) ” on page 791	Modifies an existing external procedure by changing the properties of the procedure.
“ALTER PROCEDURE (sourced) ” on page 794	Modifies an existing sourced procedure by changing the data type of one or more parameters of the sourced procedure.
“ALTER PROCEDURE (SQL) ” on page 795	Modifies an existing SQL procedure by changing the properties of the procedure.
“ALTER SCHEMA ” on page 796	Modifies an existing schema by changing the data capture attribute of the schema.
“ALTER SECURITY LABEL COMPONENT ” on page 797	Modifies a security label component.
“ALTER SECURITY POLICY ” on page 800	Modifies a security policy.
“ALTER SEQUENCE ” on page 803	Changes the definition of a sequence.
“ALTER SERVER ” on page 806	Changes the definition of a data source in a federated system.
“ALTER SERVICE CLASS ” on page 809	Changes the definition of a service class.
“ALTER STOGROUP ” on page 818	Changes the definition of a storage group.
“ALTER TABLE” on page 822	Changes the definition of a table.
“ALTER TABLESPACE ” on page 880	Changes the definition of a table space.
“ALTER THRESHOLD ” on page 893	Changes the definition of a threshold.
“ALTER TRIGGER ” on page 905	Changes the definition of a trigger.
“ALTER TRUSTED CONTEXT ” on page 906	Changes the definition of a trusted context at the current server.
“ALTER TYPE (structured) ” on page 913	Changes the definition of a structured type.
“ALTER USAGE LIST” on page 919	Changes the definition of a usage list.
“ALTER USER MAPPING ” on page 920	Changes the definition of a user authorization mapping.
“ALTER VIEW ” on page 922	Changes the definition of a view by altering a reference type column to add a scope.
“ALTER WORK ACTION SET ” on page 923	Adds, alters, or drops work actions within a work action set.
“ALTER WORK CLASS SET ” on page 936	Adds, alters, or drops work classes within a work class set.
“ALTER WORKLOAD ” on page 941	Changes a workload.
“ALTER WRAPPER ” on page 954	Updates the options that, along with a wrapper module, are used to access data sources of a specific type.
“ALTER XSROBJECT ” on page 955	Enables or disables decomposition support for a specific XML schema.
“AUDIT ” on page 958	Determines the audit policy that is to be used for a particular database or database object at the current server.
“COMMENT ” on page 973	Replaces or adds a comment to the description of an object.
“CREATE ALIAS ” on page 1019	Defines an alias for a module, nickname, sequence, table, view, or another alias.

Table 114. SQL schema statements (continued)

SQL Statement	Purpose
“CREATE AUDIT POLICY ” on page 1022	Defines an auditing policy at the current server.
“CREATE BUFFERPOOL ” on page 1024	Defines a new buffer pool.
“CREATE DATABASE PARTITION GROUP ” on page 1027	Defines a database partition group.
“CREATE EVENT MONITOR ” on page 1029	Specifies events in the database to monitor.
“CREATE EVENT MONITOR (activities) ” on page 1046	Specifies activity events in the database to monitor.
“CREATE EVENT MONITOR (change history) ” on page 1055	Specifies change history events in the database to monitor.
“CREATE EVENT MONITOR (locking) ” on page 1061	Specifies locking events in the database to monitor.
“CREATE EVENT MONITOR (package cache) statement” on page 1065	Specifies package cache statement events in the database to monitor.
“CREATE EVENT MONITOR (statistics) ” on page 1071	Specifies statistics events in the database to monitor.
“CREATE EVENT MONITOR (threshold violations) ” on page 1081	Specifies threshold violation events in the database to monitor.
“CREATE EXTERNAL TABLE ” on page 1095	Defines an external table.
“CREATE FUNCTION ” on page 1123	Registers a user-defined function.
“CREATE FUNCTION ” on page 1123	Registers a user-defined function.
“CREATE FUNCTION (aggregate interface) ” on page 1124	Registers a user-defined aggregate function at the current server.
“CREATE FUNCTION (external scalar) ” on page 1140	Registers a user-defined external scalar function.
“CREATE FUNCTION (external table) ” on page 1166	Registers a user-defined external table function.
“CREATE FUNCTION (OLE DB external table) ” on page 1187	Registers a user-defined OLE DB external table function.
“CREATE FUNCTION (sourced or template) ” on page 1196	Registers a user-defined sourced function or a function template.
“CREATE FUNCTION (SQL scalar, table, or row) ” on page 1208	Defines a user-defined SQL function.
“CREATE FUNCTION MAPPING ” on page 1224	Defines a function mapping.
“CREATE GLOBAL TEMPORARY TABLE ” on page 1228	Defines a created temporary table.
“CREATE HISTOGRAM TEMPLATE ” on page 1239	Defines a template describing the type of histogram that can be used to override one or more of the default histograms of a service class or a work class.
“CREATE INDEX ” on page 1240	Defines an index on a table.

Table 114. SQL schema statements (continued)

SQL Statement	Purpose
“CREATE INDEX EXTENSION ” on page 1261	Defines an extension object for use with indexes on tables with structured or distinct type columns.
“CREATE MASK ” on page 1266	Defines a column mask.
“CREATE METHOD ” on page 1271	Defines a method body to associate with a previously defined method specification.
“CREATE MODULE ” on page 1276	Defines a module.
“CREATE NICKNAME ” on page 1277	Defines a nickname.
“CREATE PERMISSION ” on page 1288	Defines a row permission.
“CREATE PROCEDURE ” on page 1291	Defines a procedure.
“CREATE PROCEDURE (external) ” on page 1292	Defines an external procedure.
“CREATE PROCEDURE (sourced) ” on page 1307	Defines a procedure (the sourced procedure) that is based on another procedure (the source procedure). In a federated system, a federated procedure is a sourced procedure whose source procedure is at a supported data source.
“CREATE PROCEDURE (SQL) ” on page 1312	Defines an SQL procedure.
“CREATE ROLE ” on page 1320	Defines a role at the current server.
“CREATE SCHEMA ” on page 1321	Defines a schema.
“CREATE SECURITY LABEL COMPONENT ” on page 1324	Defines a component that is to be used as part of a security policy.
“CREATE SECURITY LABEL ” on page 1326	Defines a security label.
“CREATE SECURITY POLICY ” on page 1327	Defines a security policy.
“CREATE SEQUENCE ” on page 1328	Defines a sequence.
“CREATE SERVER ” on page 1343	Defines a data source to a federated database.
“CREATE SERVICE CLASS ” on page 1333	Defines a service class.
“CREATE STOGROUP ” on page 1349	Defines a new storage group within the database.
“CREATE SYNONYM ” on page 1351	Defines a synonym for a module, nickname, sequence, table, view, or another synonym.
“CREATE TABLE ” on page 1351	Defines a table.
“CREATE TABLESPACE ” on page 1428	Defines a table space.
“CREATE THRESHOLD ” on page 1443	Defines a threshold.
“CREATE TRANSFORM ” on page 1457	Defines transformation functions.
“CREATE TRIGGER ” on page 1460	Defines a trigger.
“CREATE TRUSTED CONTEXT ” on page 1474	Defines a trusted context at the current server.
“CREATE TYPE ” on page 1479	Defines a user-defined data type at the current server.

Table 114. SQL schema statements (continued)

SQL Statement	Purpose
“CREATE TYPE (array) ” on page 1480	Defines an array type.
“CREATE TYPE (cursor) ” on page 1485	Defines a cursor type.
“CREATE TYPE (distinct) ” on page 1487	Defines a distinct data type.
“CREATE TYPE (row) ” on page 1495	Defines a row type.
“CREATE TYPE (structured) ” on page 1500	Defines a structured data type.
“CREATE TYPE MAPPING ” on page 1521	Defines a mapping between data types.
“CREATE USAGE LIST” on page 1527	Defines a usage list in order to monitor all unique sections (DML statements) that have referenced a particular table or index during their execution.
“CREATE USER MAPPING ” on page 1529	Defines a mapping between user authorizations.
“CREATE VARIABLE ” on page 1531	Defines a global variable.
“CREATE VIEW ” on page 1539	Defines a view of one or more table, view or nickname.
“CREATE WORK ACTION SET ” on page 1552	Defines a work action set and work actions within the work action set.
“CREATE WORK CLASS SET ” on page 1560	Defines a work class set.
“CREATE WORKLOAD ” on page 1564	Defines a workload.
“CREATE WRAPPER ” on page 1579	Registers a wrapper.
“DROP ” on page 1616	Deletes objects in the database.
“GRANT (database authorities) ” on page 1675	Grants authorities on the entire database.
“GRANT (exemption) ” on page 1680	Grants an exemption on an access rule for a specified label-based access control (LBAC) security policy.
“GRANT (global variable privileges) ” on page 1682	Grants one or more privileges on a created global variable.
“GRANT (index privileges) ” on page 1684	Grants the CONTROL privilege on indexes in the database.
“GRANT (module privileges) ” on page 1686	Grants privileges on a module.
“GRANT (package privileges) ” on page 1687	Grants privileges on packages in the database.
“GRANT (role) ” on page 1690	Grants roles to users, groups, or to other roles.
“GRANT (routine privileges) ” on page 1692	Grants privileges on a routine (function, method, or procedure).
“GRANT (schema privileges and authorities) ” on page 1696	Grants privileges on a schema.
“GRANT (security label) ” on page 1701	Grants a label-based access control (LBAC) security label for read access, write access, or for both read and write access.
“GRANT (sequence privileges) ” on page 1703	Grants privileges on a sequence.
“GRANT (server privileges) ” on page 1705	Grants privileges to query a specific data source.

Table 114. SQL schema statements (continued)

SQL Statement	Purpose
“GRANT (SETSESSIONUSER privilege) ” on page 1707	Grants the privilege to use the SET SESSION AUTHORIZATION statement.
“GRANT (table space privileges) ” on page 1708	Grants privileges on a table space.
“GRANT (table, view, or nickname privileges) ” on page 1710	Grants privileges on tables, views and nicknames.
“GRANT (workload privileges) ” on page 1716	Grants the USAGE privilege on a workload.
“GRANT (XSR object privileges) ” on page 1717	Grants the USAGE privilege on an XSR object.
“REFRESH TABLE ” on page 1757	Refreshes the data in a materialized query table.
“RENAME ” on page 1762	Renames an existing table.
“RENAME STOGROUP ” on page 1764	Renames an existing storage group.
“RENAME TABLESPACE ” on page 1765	Renames an existing table space.
“REVOKE (database authorities) ” on page 1771	Revokes authorities from the entire database.
“REVOKE (exemption) ” on page 1775	Revokes the exemption on an access rule for a specified label-based access control (LBAC) security policy.
“REVOKE (global variable privileges) ” on page 1777	Revokes one or more privileges on a created global variable.
“REVOKE (index privileges) ” on page 1778	Revokes the CONTROL privilege on given indexes.
“REVOKE (module privileges) ” on page 1780	Revokes privileges on a module.
“REVOKE (package privileges) ” on page 1781	Revokes privileges from given packages in the database.
“REVOKE (role) ” on page 1783	Revokes roles from users, groups, or other roles.
“REVOKE (routine privileges) ” on page 1785	Revokes privileges on a routine (function, method, or procedure).
“REVOKE (schema privileges and authorities) ” on page 1789	Revokes privileges on a schema.
“REVOKE (security label) ” on page 1792	Revokes a label-based access control (LBAC) security label for read access, write access, or for both read and write access.
“REVOKE (sequence privileges)” on page 1793	Revokes privileges on a sequence.
“REVOKE (server privileges) ” on page 1795	Revokes privileges to query a specific data source.
“REVOKE (SETSESSIONUSER privilege) ” on page 1797	Revokes the privilege to use the SET SESSION AUTHORIZATION statement.
“REVOKE (table space privileges) ” on page 1798	Revokes the USE privilege on a given table space.

Table 114. SQL schema statements (continued)

SQL Statement	Purpose
"REVOKE (table, view, or nickname privileges) " on page 1799	Revokes privileges from given tables, views or nicknames.
"REVOKE (workload privileges) " on page 1804	Revokes the USAGE privilege on a workload.
"REVOKE (XSR object privileges) " on page 1805	Revokes the USAGE privilege on an XSR object.
"SET INTEGRITY " on page 1851	Sets the set integrity pending state and checks data for constraint violations.
"TRANSFER OWNERSHIP " on page 1892	Transfers ownership of a database object.

Table 115. SQL data change statements

SQL Statement	Purpose
"DELETE " on page 1599	Deletes one or more rows from a table.
"INSERT " on page 1721	Inserts one or more rows into a table.
"MERGE " on page 1735	Updates a target (a table or view) using data from a source (result of a table reference).
"TRUNCATE " on page 1902	Deletes all rows from a table.
"UPDATE " on page 1905	Updates the values of one or more columns in one or more rows of a table.

Table 116. SQL data statements

SQL Statement	Purpose
"ALLOCATE CURSOR " on page 749	Allocates a cursor for the result set identified by the result set locator variable.
"ASSOCIATE LOCATORS " on page 956	Gets the result set locator value for each result set returned by a procedure.
"CLOSE " on page 971	Closes a cursor.
"DECLARE CURSOR " on page 1581	Defines an SQL cursor.
"FETCH " on page 1659	Assigns values of a row to host variables.
"FLUSH AUTHENTICATION CACHE " on page 1668	Removes cached users in the authentication cache
"FLUSH BUFFERPOOLS " on page 1663	Writes out the dirty pages in the buffer pools to disk.
"FLUSH EVENT MONITOR " on page 1663	Writes out the active internal buffer of an event monitor.
"FLUSH FEDERATED CACHE " on page 1664	The FLUSH FEDERATED CACHE statement flushes the federated cache, allowing fresh metadata to be obtained the next time an SQL statement is issued against the remote table or view using a federated three part name.
"FLUSH OPTIMIZATION PROFILE CACHE " on page 1665	Removes the cached optimization profiles.

Table 116. SQL data statements (continued)

SQL Statement	Purpose
“FLUSH PACKAGE CACHE ” on page 1667	Removes all cached dynamic SQL statements currently in the package cache.
“FREE LOCATOR ” on page 1671	Removes the association between a locator variable and its value.
“LOCK TABLE ” on page 1732	Either prevents concurrent processes from changing a table or prevents concurrent processes from using a table.
“OPEN ” on page 1746	Prepares a cursor that will be used to retrieve values when the FETCH statement is issued.
“SELECT INTO ” on page 1810	Specifies a result table of no more than one row and assigns the values to host variables.
“SET variable ” on page 1878	Assigns values to variables.
“VALUES INTO ” on page 1921	Specifies a result table of no more than one row and assigns the values to host variables.

Table 117. SQL transaction statements

SQL Statement	Purpose
“COMMIT ” on page 982	Terminates a unit of work and commits the database changes made by that unit of work.
“RELEASE SAVEPOINT ” on page 1761	Releases a savepoint within a transaction.
“ROLLBACK ” on page 1806	Terminates a unit of work and backs out the database changes made by that unit of work.
“SAVEPOINT ” on page 1808	Sets a savepoint within a transaction.

Table 118. SQL connection statements

SQL Statement	Purpose
“CONNECT (type 1) ” on page 1006	Connects to an application server according to the rules for remote unit of work.
“CONNECT (type 2) ” on page 1012	Connects to an application server according to the rules for application-directed distributed unit of work.
“DISCONNECT ” on page 1614	Terminates one or more connections when there is no active unit of work.
“RELEASE (connection) ” on page 1760	Places one or more connections in the release-pending state.
“SET CONNECTION ” on page 1814	Changes the state of a connection from dormant to current, making the specified location the current server.

Table 119. SQL dynamic statements

SQL Statement	Purpose
“DESCRIBE ” on page 1608	Obtains information about an object.
“DESCRIBE INPUT ” on page 1608	Obtains information about the input parameter markers of a prepared statement.
“DESCRIBE OUTPUT ” on page 1611	Obtains information about a prepared statement or information about the select list columns in a prepared SELECT statement.

Table 119. SQL dynamic statements (continued)

SQL Statement	Purpose
“EXECUTE ” on page 1645	Executes a prepared SQL statement.
“EXECUTE IMMEDIATE ” on page 1653	Prepares and executes an SQL statement.
“PREPARE ” on page 1752	Prepares an SQL statement (with optional parameters) for execution.

Table 120. SQL session statements

SQL Statement	Purpose
“DECLARE GLOBAL TEMPORARY TABLE ” on page 1586	Defines a declared temporary table.
“EXPLAIN ” on page 1655	Captures information about the chosen access plan.
“SET COMPILATION ENVIRONMENT ” on page 1813	Changes the current compilation environment in the connection to match the values contained in the compilation environment provided by a deadlock event monitor.
“SET CURRENT DECFLOAT ROUNDING MODE ” on page 1816	Verifies that the specified rounding mode is the value that is currently set for the CURRENT DECFLOAT ROUNDING MODE special register.
“SET CURRENT DEFAULT TRANSFORM GROUP ” on page 1817	Changes the value of the CURRENT DEFAULT TRANSFORM GROUP special register.
“SET CURRENT DEGREE ” on page 1818	Changes the value of the CURRENT DEGREE special register.
“SET CURRENT EXPLAIN MODE ” on page 1820	Changes the value of the CURRENT EXPLAIN MODE special register.
“SET CURRENT EXPLAIN SNAPSHOT ” on page 1822	Changes the value of the CURRENT EXPLAIN SNAPSHOT special register.
“SET CURRENT FEDERATED ASYNCHRONY ” on page 1824	Changes the value of the CURRENT FEDERATED ASYNCHRONY special register.
“SET CURRENT IMPLICIT XMLPARSE OPTION ” on page 1825	Changes the value of the CURRENT IMPLICIT XMLPARSE OPTION special register.
“SET CURRENT ISOLATION ” on page 1826	Changes the value of the CURRENT ISOLATION special register.
“SET CURRENT LOCALE LC_MESSAGES ” on page 1827	Changes the value of the CURRENT LOCALE LC_MESSAGES special register.
“SET CURRENT LOCALE LC_TIME ” on page 1828	Changes the value of the CURRENT LOCALE LC_TIME special register.
“SET CURRENT LOCK TIMEOUT ” on page 1829	Changes the value of the CURRENT LOCK TIMEOUT special register.
“SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION ” on page 1830	Changes the value of the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register.
“SET CURRENT MDC ROLLOUT MODE ” on page 1832	Assigns a value to the CURRENT MDC ROLLOUT MODE special register.
“SET CURRENT OPTIMIZATION PROFILE ” on page 1834	Assigns a value to the CURRENT OPTIMIZATION PROFILE special register.

Table 120. SQL session statements (continued)

SQL Statement	Purpose
“SET CURRENT PACKAGE PATH ” on page 1836	Assigns a value to the CURRENT PACKAGE PATH special register.
“SET CURRENT PACKAGESET ” on page 1839	Sets the schema name for package selection.
“SET CURRENT QUERY OPTIMIZATION ” on page 1841	Changes the value of the CURRENT QUERY OPTIMIZATION special register.
“SET CURRENT REFRESH AGE ” on page 1843	Changes the value of the CURRENT REFRESH AGE special register.
“SET CURRENT SQL_CCFLAGS ” on page 1845	Changes the value of the CURRENT SQL_CCFLAGS special register.
“SET CURRENT TEMPORAL BUSINESS_TIME ” on page 1846	Changes the value of the CURRENT TEMPORAL BUSINESS_TIME special register.
“SET CURRENT TEMPORAL SYSTEM_TIME ” on page 1847	Changes the value of the CURRENT TEMPORAL SYSTEM_TIME special register.
“SET ENCRYPTION PASSWORD ” on page 1848	Sets the password for encryption.
“SET EVENT MONITOR STATE ” on page 1850	Activates or deactivates an event monitor.
“SET PASSTHRU ” on page 1867	Opens a session for submitting data source native SQL directly to the data source.
“SET PATH ” on page 1868	Changes the value of the CURRENT PATH special register.
“SET ROLE ” on page 1870	Verifies that the authorization ID of the session is a member of a specific role.
“SET SCHEMA ” on page 1871	Changes the value of the CURRENT SCHEMA special register.
“SET SERVER OPTION ” on page 1873	Sets server option settings.
“SET SESSION AUTHORIZATION ” on page 1874	Changes the value of the SESSION USER special register.
“SET USAGE LIST STATE” on page 1876	Manages the state of a usage list and the associated data and memory.

Table 121. SQL embedded host language statements

SQL Statement	Purpose
“BEGIN DECLARE SECTION ” on page 961	Marks the beginning of a host variable declaration section.
“END DECLARE SECTION ” on page 1645	Marks the end of a host variable declaration section.
“GET DIAGNOSTICS ” on page 1671	Used to obtain information about the previously executed SQL statement.
“INCLUDE ” on page 1719	Inserts code or declarations into a source program.
“RESIGNAL ” on page 1767	Used to resignal an error or warning condition.
“SIGNAL ” on page 1889	Used to signal an error or warning condition.
“WHENEVER ” on page 1924	Defines actions to be taken on the basis of SQL return codes.

Table 122. SQL control statements

SQL Statement	Purpose
“CALL ” on page 962	Calls a procedure.
“CASE ” on page 969	Selects an execution path based on multiple conditions.
“Compound SQL ” on page 984	Encloses SQL statements with BEGIN and END keywords.
“Compound SQL (inlined) ” on page 984	Combines one or more other SQL statements into an dynamic block.
“Compound SQL (embedded) ” on page 988	Combines one or more other SQL statements into an executable block.
“Compound SQL (compiled) ” on page 991	Groups other statements together in an SQL procedure.
“FOR ” on page 1668	Executes a statement or group of statements for each row of a table.
“GOTO ” on page 1674	Used to branch to a user-defined label within an SQL procedure.
“IF ” on page 1718	Selects an execution path based on the evaluation of a condition.
“ITERATE ” on page 1730	Causes the flow of control to return to the beginning of a labelled loop.
“LEAVE ” on page 1731	Transfers program control out of a loop or a compound statement.
“LOOP ” on page 1733	Repeats the execution of a statement or a group of statements.
“PIPE ” on page 1750	Returns a row from a compiled table function.
“REPEAT ” on page 1766	Executes a statement or group of statements until a search condition is true.
“RESIGNAL ” on page 1767	Used to resignal an error or warning condition.
“RETURN ” on page 1769	Used to return from a routine.
“SIGNAL ” on page 1889	Used to signal an error or warning condition.
“WHILE ” on page 1926	Repeats the execution of a statement or group of statements while a specified condition is true.

How SQL statements are invoked

SQL statements are classified as executable or non-executable.

An *executable statement* can be invoked in four ways. It can be:

- Issued interactively
- Prepared and executed dynamically
- Embedded in an application program
- Embedded in an SQL procedure, trigger, compound SQL (compiled), or compound SQL (inlined) with some restrictions:
  - Refer to "SQL-procedure-statement" in “Compound SQL (compiled) ” on page 991 for the set of executable statements supported in SQL procedures and compound SQL (compiled) statements.
  - Refer to "SQL-statement" in “Compound SQL (inlined) ” on page 984 statement for the set of executable statements supported in compound SQL (inlined) statements.
  - Refer to "SQL-procedure-statement" in “CREATE TRIGGER ” on page 1460 for the set of executable statements supported in a trigger.

Depending on the statement, some or all of these methods can be used. Statements embedded in REXX are prepared and executed dynamically.

A *non-executable statement* can only be embedded in an application program.

Another SQL statement construct is the select-statement. A *select-statement* can be invoked in three ways. It can be:

- Issued interactively
- Prepared dynamically, referenced in DECLARE CURSOR, and executed implicitly by OPEN, FETCH and CLOSE (dynamic invocation)
- Included in DECLARE CURSOR, and executed implicitly by OPEN, FETCH and CLOSE (static invocation)

Embedding a statement in an application program

SQL statements can be included in a source program that will be submitted to a precompiler. Such statements are said to be *embedded* in the program.

An embedded statement can be placed anywhere in the program where a host language statement is allowed. Each embedded statement must be preceded by the keywords EXEC SQL.

An executable statement embedded in an application program is executed every time a statement of the host language would be executed if it were specified in the same place. Thus, a statement within a loop is executed every time the loop is executed, and a statement within a conditional construct is executed only when the condition is satisfied.

An embedded statement can contain references to host variables. A host variable referenced in this way can be used in two ways. It can be used:

- As input (the current value of the host variable is used in the execution of the statement)
- As output (the variable is assigned a new value as a result of executing the statement)

In particular, all references to host variables in expressions and predicates are effectively replaced by current values of the variables; that is, the variables are used as input.

All executable statements should be followed by a test of the SQL return code. Alternatively, the WHENEVER statement (which is itself non-executable) can be used to change the flow of control immediately after the execution of an embedded statement.

All objects referenced in data manipulation language (DML) statements must exist when the statements are bound to a database.

An embedded non-executable statement is processed only by the precompiler. The precompiler reports any errors encountered in the statement. The statement is *never* processed during program execution; therefore, such statements should not be followed by a test of the SQL return code.

Statements can be included in the SQL-procedure-body portion of the CREATE PROCEDURE statement. Such statements are said to be embedded in the SQL procedure. Whenever an SQL statement description refers to a *host-variable*, an *SQL-variable* can be used if the statement is embedded in an SQL procedure.

Dynamic preparation and execution

An application program can dynamically build an SQL statement in the form of a character string placed in a host variable.

In general, the statement is built from some data available to the program (for example, input from a workstation). The statement (not a select-statement) constructed can be prepared for execution by means of the (embedded) PREPARE statement, and executed by means of the (embedded) EXECUTE statement. Alternatively, an (embedded) EXECUTE IMMEDIATE statement can be used to prepare and execute the statement in one step.

A statement that is going to be dynamically prepared must not contain references to host variables. It can instead contain parameter markers. (For rules concerning parameter markers, see "PREPARE".) When the prepared statement is executed, the parameter markers are effectively replaced by current values of the

host variables specified in the EXECUTE statement. Once prepared, a statement can be executed several times with different values for the host variables. Parameter markers are not allowed in the EXECUTE IMMEDIATE statement.

Successful or unsuccessful execution of the statement is indicated by the setting of an SQL return code in the SQLCA after the EXECUTE (or EXECUTE IMMEDIATE) statement completes. The SQL return code should be checked, as previously described. For more information, see "Detecting and processing error and warning conditions in host language applications" on page 739.

Static invocation of a select-statement

A select-statement can be included as a part of the (non-executable) DECLARE CURSOR statement.

Such a statement is executed every time the cursor is opened by means of the (embedded) OPEN statement. After the cursor is open, the result table can be retrieved, one row at a time, by successive executions of the FETCH statement.

Used in this way, the select-statement can contain references to host variables. These references are effectively replaced by the values that the variables have when the OPEN statement executes.

Dynamic invocation of a select-statement

An application program can dynamically build a select-statement in the form of a character string placed in a host variable.

In general, the statement is built from some data available to the program (for example, a query obtained from a workstation). The statement so constructed can be prepared for execution by means of the (embedded) PREPARE statement, and referenced by a (non-executable) DECLARE CURSOR statement. The statement is then executed every time the cursor is opened by means of the (embedded) OPEN statement. After the cursor is open, the result table can be retrieved, one row at a time, by successive executions of the FETCH statement.

Used in this way, the select-statement must not contain references to host variables. It can contain parameter markers instead. The parameter markers are effectively replaced by the values of the host variables specified in the OPEN statement.

Interactive invocation

A capability for entering SQL statements from a workstation is part of the architecture of the database manager. A statement entered in this way is said to be issued interactively.

Such a statement must be an executable statement that does not contain parameter markers or references to host variables, because these make sense only in the context of an application program.

SQL use with other host systems

SQL statement syntax exhibits minor variations among different types of host systems (Db2 for z/OS, Db2 for IBM i, Db2).

Regardless of whether the SQL statements in an application are static or dynamic, it is important - if the application is meant to access different database host systems - to ensure that the SQL statements and precompile/bind options are supported on the database systems that the application will access.

Further information about SQL statements used in other host systems can be found in the *SQL Reference* manuals for Db2 for z/OS and Db2 for IBM i.

Detecting and processing error and warning conditions in host language applications

An application program containing executable SQL statements can use either SQLCODE or SQLSTATE values to handle return codes from SQL statements.

There are two ways in which an application can get access to these values.

- Include a structure named SQLCA. The SQLCA includes an integer variable named SQLCODE and a character string variable named SQLSTATE. In REXX, an SQLCA is provided automatically. In other languages, an SQLCA can be obtained by using the INCLUDE SQLCA statement.
- If LANGLEVEL SQL92E is specified as a precompile option, a variable named SQLCODE or SQLSTATE can be declared in the SQL declare section of the program. If neither of these variables is declared in the SQL declare section, it is assumed that a variable named SQLCODE is declared elsewhere in the program. With LANGLEVEL SQL92E, the program should not have an INCLUDE SQLCA statement.

An SQLCODE is set by the database manager after each SQL statement executes. All database managers conform to the ISO/ANSI SQL standard, as follows:

- If SQLCODE = 0 and SQLWARN0 is blank, execution was successful.
- If SQLCODE = 100, "no data" was found. For example, a FETCH statement returned no data, because the cursor was positioned after the last row of the result table.
- If SQLCODE > 0 and not = 100, execution was successful with a warning.
- If SQLCODE = 0 and SQLWARN0 = 'W', execution was successful, but one or more warning indicators were set.
- If SQLCODE < 0, execution was not successful.

The meaning of SQLCODE values other than 0 and 100 is product-specific.

An SQLSTATE is set by the database manager after each SQL statement executes. Application programs can check the execution of SQL statements by testing SQLSTATE instead of SQLCODE. SQLSTATE provides common codes for common error conditions. Application programs can test for specific errors or classes of errors. The coding scheme is the same for all IBM database managers, and is based on the ISO/ANSI SQL92 standard.

SQL comments

Static SQL statements can include host language or SQL comments. Dynamic SQL statements can include SQL comments.

There are two types of SQL comments:

simple comments
Simple comments are introduced by two consecutive hyphens (--) and end with the end of line.

bracketed comments
Bracketed comments are introduced by /* and end with */.

The following rules apply to the use of simple comments:

- The two hyphens must be on the same line and must not be separated by a space.
- Simple comments can be started wherever a space is valid (except within a delimiter token or between 'EXEC' and 'SQL').
- Simple comments cannot be continued to the next line.
- In COBOL, the hyphens must be preceded by a space.

The following rules apply to the use of bracketed comments:

- The /* must be on the same line and must not be separated by a space.
- The */ must be on the same line and must not be separated by a space.
- Bracketed comments can be started wherever a space is valid (except within a delimiter token or between 'EXEC' and 'SQL').
- Bracketed comments can be continued to subsequent lines.

Examples

- *Example 1:* This example shows how to include simple comments in a statement:

```
CREATE VIEW PRJ_MAXPER       -- PROJECTS WITH MOST SUPPORT PERSONNEL
  AS SELECT PROJNO, PROJNAME -- NUMBER AND NAME OF PROJECT
    FROM PROJECT
    WHERE DEPTNO = 'E21'     -- SYSTEMS SUPPORT DEPT CODE
    AND PRSTAFF > 1
```

- *Example 2:* This example shows how to include bracketed comments in a statement:

```
CREATE VIEW PRJ_MAXPER       /* PROJECTS WITH MOST SUPPORT
                                  PERSONNEL                */
  AS SELECT PROJNO, PROJNAME /* NUMBER AND NAME OF PROJECT  */
    FROM PROJECT
    WHERE DEPTNO = 'E21'     /* SYSTEMS SUPPORT DEPT CODE   */
    AND PRSTAFF > 1
```

Conditional compilation in SQL

Conditional compilation allows SQL to include compiler directives which are used to determine the actual SQL that gets compiled.

There are two types of compiler directives that can be used for conditional compilation:

Selection directive
A compiler control statement used to determine the selection of a code fragment. The _IF directive can reference inquiry directives or global variables that are defined as a constant.

Inquiry directive
A reference to a compiler named constant that is assigned by the system or specified as a conditional compilation named constant in CURRENT SQL_CCFLAGS. An inquiry directive can be used directly or in a selection directive.

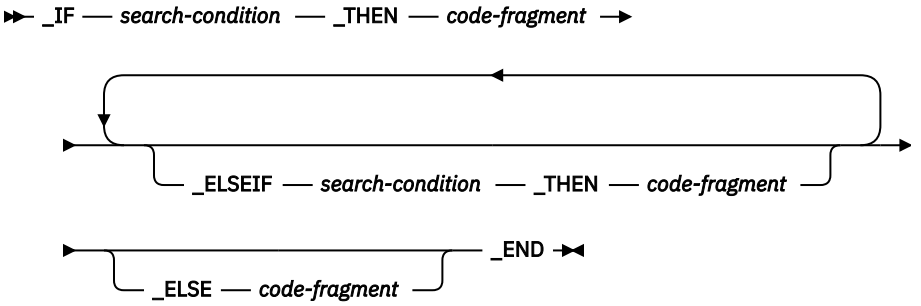
These directives can be used in the following contexts:

- SQL procedure definitions
- Compiled SQL function definitions
- Compiled trigger definitions
- Oracle PL/SQL package definitions

A directive can only appear after the object type (FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, or TRIGGER) has been identified in the data definition language statement.

Selection directive

The selection directive is very similar to the IF statement except there are prefixes on the keywords to indicate use of conditional compilation and the terminating keyword is _END.



search-condition
Specifies the condition that is evaluated to determine what *code-fragment*, if any, is included. If the condition is unknown or false, evaluation continues with the next search condition, until a condition

is true, the _ELSE clause is reached, or the end of the selection directive is reached. The search condition can include only the following elements (SQLSTATE 428HV):

- Constants of type BOOLEAN, INTEGER, or VARCHAR
- NULL constants
- Inquiry directives
- Global constants, where the defined constant value is a simple literal of type BOOLEAN, INTEGER, or VARCHAR
- Basic predicates
- NULL predicates
- Predicates that are a Boolean constant or a Boolean inquiry directive
- Logical operators (AND, OR, and NOT)

code-fragment
A portion of SQL code that can be included in the context of the SQL statement where the selection directive appears. There must not be a selection directive in *code-fragment* (SQLSTATE 428HV).

Inquiry directive

An inquiry directive is used to inquire about the compilation environment. An inquiry directive is specified in an SQL statement as an ordinary identifier prefixed with two underscore characters. The actual identifier can represent one of the following values:

- A compilation environment value defined by the system
- A compilation value defined by a user at the database level or at the individual session level

The only compilation environment variable defined by the system is __SQL_LINE, which provides the line number of SQL that is currently being compiled.

A user-defined compilation value can be defined at the database level using the **sql_ccflags** database configuration parameter or at a session level by assigning it to the CURRENT SQL_CCFLAGS special register.

If an inquiry directive is referenced but is not defined, processing continues assuming that the value for the inquiry directive is the null value.

Notes

- **References to global variables defined as constants:** A reference to a global variable (which can also be a reference to a module variable published in a module) in a selection directive is used to provide a value based on a constant at the time of compilation only. The referenced global variable must meet the following requirements:
  - Exist at the current server (SQLSTATE 42704)
  - Have a data type of BOOLEAN, INTEGER, or VARCHAR (SQLSTATE 428HV)
  - Be defined using the CONSTANT clause with a single constant value (SQLSTATE 428HV)

  Such a global variable is known as a global constant. Subsequent changes to the global constant do not have any impact on statements that are already compiled.
- **Syntax alternatives:** If the data server environment is enabled for PL/SQL statement execution:
  - ELSIF can be specified instead of ELSEIF
  - A dollar character ($) can be used instead of an underscore character (_) as the prefix for the keywords for conditional compilation
  - Two dollar characters ($$) can be used instead of two underscore characters (__) as the prefix for an inquiry directive

The dollar character prefix is intended only to support existing SQL statements that use inquiry directives and is not recommended for use when writing new SQL statements.

Example

Specify a database-wide setting for a compilation value called DBV97 that has a value of TRUE.

```
UPDATE DATABASE CONFIGURATION USING SQL_CCFLAGS DB2V97:TRUE
```

The value is available as the default for any subsequent connection to the database.

If a particular session needed a maximum number of years compilation value for use in defining some routines in the current session, the default SQL_CCFLAGS can be extended using the SET CURRENT SQL_CCFLAGS statement.

```
BEGIN
  DECLARE CCFLAGS_LIST VARCHAR(1024);
  SET CCFLAGS_LIST = CURRENT SQL_CCFLAGS CONCAT ',max_years:50';
  SET CURRENT SQL_CCFLAGS = CCFLAGS_LIST;
END
```

The use of CURRENT SQL_CCFLAGS on the right side of the assignment to the CCFLAGS_LIST variable keeps the existing SQL_CCFLAGS settings, while the string constant provides the additional compilation values.

Here is an example of a CREATE PROCEDURE statement that uses the contents of the CURRENT SQL_CCFLAGS.

```
CREATE PROCEDURE CHECK_YEARS (IN YEARS_WORKED INTEGER)
BEGIN
   _IF __DB2V97 _THEN
      IF YEARS_WORKED > __MAX_YEARS THEN
      ...
      END IF;
   _END
```

The inquiry directive __DB2V97 is used as a Boolean value to determine if the code can be included. The inquiry directive __MAX_YEARS is replaced during compilation by the constant value 50.

About SQL control statements

SQL control statements, also called SQL Procedural Language (SQL PL), are SQL statements that allow SQL to be used in a manner similar to writing a program in a structured programming language.

SQL control statements provide the capability to control the logic flow, declare, and set variables, and handle warnings and exceptions. Some SQL control statements include other nested SQL statements. SQL control statements can be used in the body of a routine, trigger or a compound statement.

References to SQL parameters, SQL variables, and global variables

SQL parameters, SQL variables, and global variables can be referenced anywhere in an SQL procedure statement where an expression or variable can be specified.

Host variables cannot be specified in SQL routines, SQL triggers or dynamic compound statements. SQL parameters can be referenced anywhere in the routine body, and can be qualified with the routine name. SQL variables can be referenced anywhere in the compound statement in which they are declared, and can be qualified with the label name specified at the beginning of the compound statement. If an SQL parameter or SQL variable has a row data type, fields can be referenced anywhere an SQL parameter or SQL variable can be referenced. Global variables can be referenced within any expression as long as the expression is not required to be deterministic. The following scenarios require deterministic expressions, which preclude the use of global variables:

- Check constraints
- Definitions of generated columns

- Refresh immediate MQTs

All SQL parameters, SQL variables, row variable fields, and global variables are considered nullable. The name of an SQL parameter, SQL variable, row variable field, or global variable in an SQL routine can be the same as the name of a column in a table or view referenced in the routine. The name of an SQL variable or row variable field can also be the same as the name of another SQL variable or row variable field declared in the same routine. This can occur when the two SQL variables are declared in different compound statements. The compound statement that contains the declaration of an SQL variable determines the scope of that variable. For more information, see “Compound SQL (compiled) ” on page 991.

The name of an SQL variable or SQL parameter in an SQL routine can be the same as the name of an identifier used in certain SQL statements. If the name is not qualified, the following rules describe whether the name refers to the identifier or to the SQL parameter or SQL variable:

- In the SET PATH and SET SCHEMA statements, the name is checked as an SQL parameter or SQL variable. If not found as an SQL variable or SQL parameter, it is used as an identifier.
- In the CONNECT, DISCONNECT, RELEASE, and SET CONNECTION statements, the name is used as an identifier.

Names that are the same should be explicitly qualified. Qualifying a name clearly indicates whether the name refers to a column, SQL variable, SQL parameter, row variable field, or global variable. If the name is not qualified, or qualified but still ambiguous, the following rules describe whether the name refers to a column, an SQL variable, an SQL parameter, or a global variable:

- If the tables and views specified in an SQL routine body exist at the time the routine is created, the name is first checked as a column name. If not found as a column, it is then checked as an SQL variable in the compound statement, then checked as an SQL parameter, and then, finally, checked as a global variable.
- If the referenced tables or views do not exist at the time the routine is created, the name is first checked as an SQL variable in the compound statement, then as an SQL parameter, and then as a global variable. The variable can be declared within the compound statement that contains the reference, or within a compound statement in which that compound statement is nested. If two SQL variables are within the same scope and have the same name, which can happen if they are declared in different compound statements, the SQL variable that is declared in the innermost compound statement is used. If not found, it is assumed to be a column.

References to SQL labels

Labels can be specified on most SQL procedure statements.

The compound statement that contains the statement that defines a label determines the scope of that label name. A label name must be unique within the compound statement in which it is defined, including any labels defined in compound statements that are nested within that compound statement (SQLSTATE 42734). The label must not be the same as a label specified on the compound statement itself (SQLSTATE 42734), or the same as the name of the routine that contains the SQL procedure statement (SQLSTATE 42734).

A label name can only be referenced within the compound statement in which it is defined, including any compound statements that are nested within that compound statement. A label can be used to qualify the name of an SQL variable, or it can be specified as the target of a GOTO, LEAVE, or ITERATE statement.

References to SQL condition names

The name of an SQL condition can be the same as the name of another SQL condition declared in the same routine.

This can occur when the two SQL conditions are declared in different compound statements. The compound statement that contains the declaration of an SQL condition name determines the scope of that condition name. A condition name must be unique within the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement (SQLSTATE 42734). A condition name can only be referenced within the compound statement

in which it is declared, including any compound statements that are nested within that compound statement. When there is a reference to a condition name, the condition that is declared in the innermost compound statement is the condition that is used. For more information, see "Compound SQL (inlined)".

References to SQL statement names

The name of an SQL statement can be the same as the name of another SQL statement declared in the same routine.

This can occur when the two SQL statements are declared in different compound statements. The compound statement that contains the declaration of an SQL statement name determines the scope of that statement name. A statement name must be unique within the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement (SQLSTATE 42734). A statement name can only be referenced within the compound statement in which it is declared, including any compound statements that are nested within that compound statement. When there is a reference to a statement name, the statement that is declared in the innermost compound statement is the statement that is used. For more information, see "Compound SQL (inlined)".

References to SQL cursor names

Cursor names include the names of declared cursors and the names of cursor variables.

The name of an SQL cursor can be the same as the name of another SQL cursor declared in the same routine. This can occur when the two SQL cursors are declared in different compound statements.

The compound statement that contains the declaration of an SQL cursor determines the scope of that cursor name. A cursor name must be unique within the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement (SQLSTATE 42734). A cursor name can only be referenced within the compound statement in which it is declared, including any compound statements that are nested within that compound statement. When there is a reference to a cursor name, the cursor that is declared in the innermost compound statement is the cursor that is used. For more information, see "Compound SQL (inlined)".

If the cursor constructor assigned to a cursor variable contains a reference to a local SQL variable, then any OPEN statement that uses the cursor variable must be within the scope where the local SQL variable was declared.

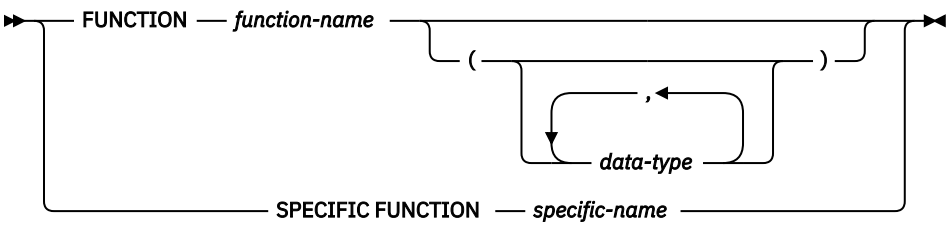
Function, method, and procedure designators

This topic describes syntax fragments that are used to uniquely identify a function, method, or procedure that is not defined in a module.

Function designator

A function designator uniquely identifies a single function. Function designators typically appear in DDL statements for functions (such as DROP or ALTER). A function designator must not identify a module function (SQLSTATE 42883).

function-designator



FUNCTION *function-name*

Identifies a particular function, and is valid only if there is exactly one function instance with the name *function-name* in the schema. The identified function can have any number of parameters defined for it. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. If no function by this name exists in the named or implied schema, an error (SQLSTATE 42704) is raised. If there is more than one instance of the function in the named or implied schema, an error (SQLSTATE 42725) is raised.

FUNCTION *function-name (data-type,...)*

Provides the function signature, which uniquely identifies the function. The function resolution algorithm is not used.

function-name

Specifies the name of the function. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

(data-type,...)

Values must match the data types that were specified (in the corresponding position) on the CREATE FUNCTION statement. The number of data types, and the logical concatenation of the data types, is used to identify the specific function instance.

If a data type is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision, or scale for the parameterized data types. Instead, an empty set of parentheses can be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (REAL or DOUBLE).

If length, precision, or scale is coded, the value must exactly match that specified in the CREATE FUNCTION statement. When length is specified for character and graphic string data types, the string unit of the length attribute must exactly match that specified in the CREATE FUNCTION statement.

A type of FLOAT(*n*) does not need to match the defined value for *n*, because $0 < n < 25$ means REAL, and $24 < n < 54$ means DOUBLE. Matching occurs on the basis of whether the type is REAL or DOUBLE.

If no function with the specified signature exists in the named or implied schema, an error (SQLSTATE 42883) is raised.

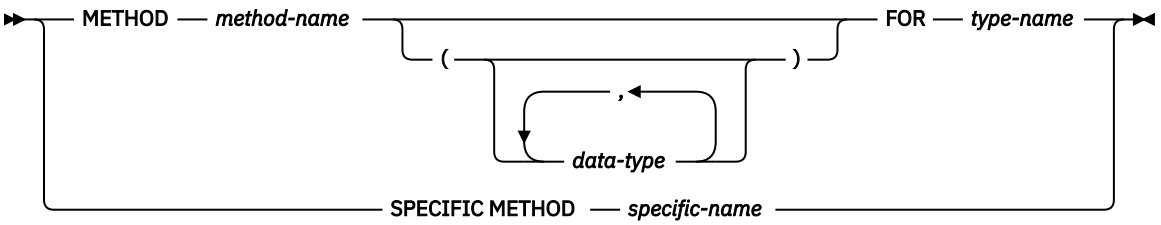
SPECIFIC FUNCTION *specific-name*

Identifies a particular user-defined function, using the name that is specified or defaulted to at function creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific function instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

Method designator

A method designator uniquely identifies a single method. Method designators typically appear in DDL statements for methods (such as DROP or ALTER).

method-designator



METHOD *method-name*
Identifies a particular method, and is valid only if there is exactly one method instance with the name *method-name* for the type *type-name*. The identified method can have any number of parameters defined for it. If no method by this name exists for the type, an error (SQLSTATE 42704) is raised. If there is more than one instance of the method for the type, an error (SQLSTATE 42725) is raised.

METHOD *method-name (data-type,...)*
Provides the method signature, which uniquely identifies the method. The method resolution algorithm is not used.

method-name
Specifies the name of the method for the type *type-name*.

(data-type,...)
Values must match the data types that were specified (in the corresponding position) on the CREATE TYPE statement. The number of data types, and the logical concatenation of the data types, is used to identify the specific method instance.

If a data type is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision, or scale for the parameterized data types. Instead, an empty set of parentheses can be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (REAL or DOUBLE).

If length, precision, or scale is coded, the value must exactly match that specified in the CREATE TYPE statement. When length is specified for character and graphic string data types, the string unit of the length attribute must exactly match that specified in the CREATE TYPE statement.

A type of FLOAT(*n*) does not need to match the defined value for *n*, because $0 < n < 25$ means REAL, and $24 < n < 54$ means DOUBLE. Matching occurs on the basis of whether the type is REAL or DOUBLE.

If no method with the specified signature exists for the type in the named or implied schema, an error (SQLSTATE 42883) is raised.

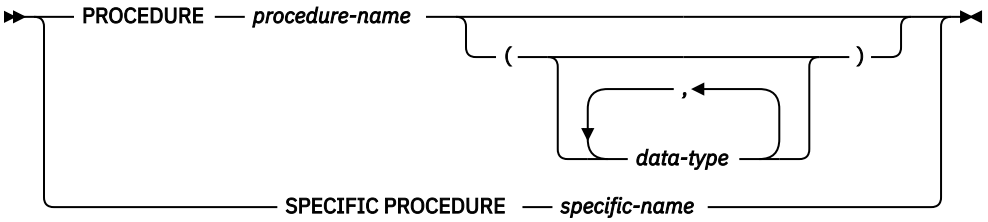
FOR *type-name*
Names the type with which the specified method is to be associated. The name must identify a type already described in the catalog (SQLSTATE 42704). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

SPECIFIC METHOD *specific-name*
Identifies a particular method, using the name that is specified or defaulted to at method creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific method instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

Procedure designator

A procedure designator uniquely identifies a single procedure. Procedure designators typically appear in DDL statements for procedures (such as DROP or ALTER). A procedure designator must not identify a module procedure (SQLSTATE 42883).

procedure-designator



PROCEDURE *procedure-name*
Identifies a particular procedure, and is valid only if there is exactly one procedure instance with the name *procedure-name* in the schema. The identified procedure can have any number of parameters defined for it. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. If no procedure by this name exists in the named or implied schema, an error (SQLSTATE 42704) is raised. If there is more than one instance of the procedure in the named or implied schema, an error (SQLSTATE 42725) is raised.

PROCEDURE *procedure-name (data-type,...)*
Provides the procedure signature, which uniquely identifies the procedure. The procedure resolution algorithm is not used.

procedure-name
Specifies the name of the procedure. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

(data-type,...)
Values must match the data types that were specified (in the corresponding position) on the CREATE PROCEDURE statement. The number of data types, and the logical concatenation of the data types, is used to identify the specific procedure instance.

If a data type is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision, or scale for the parameterized data types. Instead, an empty set of parentheses can be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (REAL or DOUBLE).

If length, precision, or scale is coded, the value must exactly match that specified in the CREATE PROCEDURE statement. When length is specified for character and graphic string data types, the string unit of the length attribute must exactly match that specified in the CREATE PROCEDURE statement.

A type of FLOAT(*n*) does not need to match the defined value for *n*, because $0 < n < 25$ means REAL, and $24 < n < 54$ means DOUBLE. Matching occurs on the basis of whether the type is REAL or DOUBLE.

If no procedure with the specified signature exists in the named or implied schema, an error (SQLSTATE 42883) is raised.

SPECIFIC PROCEDURE *specific-name*
Identifies a particular procedure, using the name that is specified or defaulted to at procedure creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a

qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific procedure instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

ALLOCATE CURSOR

The ALLOCATE CURSOR statement allocates a cursor for the result set identified by the result set locator variable.

For more information about result set locator variables, see the description of the ASSOCIATE LOCATORS statement.

Invocation

This statement can only be embedded in an SQL procedure. It is not an executable statement and cannot be dynamically prepared.

Authorization

None required.

Syntax

Description

cursor-name
Names the cursor. The name must not identify a cursor that has already been declared in the source SQL procedure (SQLSTATE 24502).

CURSOR FOR RESULT SET *rs-locator-variable*

Names a result set locator variable that has been declared in the source SQL procedure, according to the rules for declaring result set locator variables. For more information about declaring SQL variables, see "Compound SQL (Procedure) statement".

The result set locator variable must contain a valid result set locator value, as returned by the ASSOCIATE LOCATORS SQL statement (SQLSTATE 0F001).

Rules

- The following rules apply when using an allocated cursor:
  - An allocated cursor cannot be opened with the OPEN statement (SQLSTATE 24502).
  - An allocated cursor cannot be used in a positioned UPDATE or DELETE statement (SQLSTATE 42828).
  - An allocated cursor can be closed with the CLOSE statement. Closing an allocated cursor closes the associated cursor.
  - Only one cursor can be allocated to each result set.
- Allocated cursors last until a rollback operation, an implicit close, or an explicit close.
- A commit operation destroys allocated cursors that are not defined WITH HOLD.
- Destroying an allocated cursor closes the associated cursor in the SQL procedure.

Example

This SQL procedure example defines and associates cursor C1 with the result set locator variable LOC1 and the related result set returned by the SQL procedure:

```
ALLOCATE C1 CURSOR FOR RESULT SET LOC1;
```

ALTER AUDIT POLICY

The ALTER AUDIT POLICY statement modifies the definition of an audit policy at the current server.

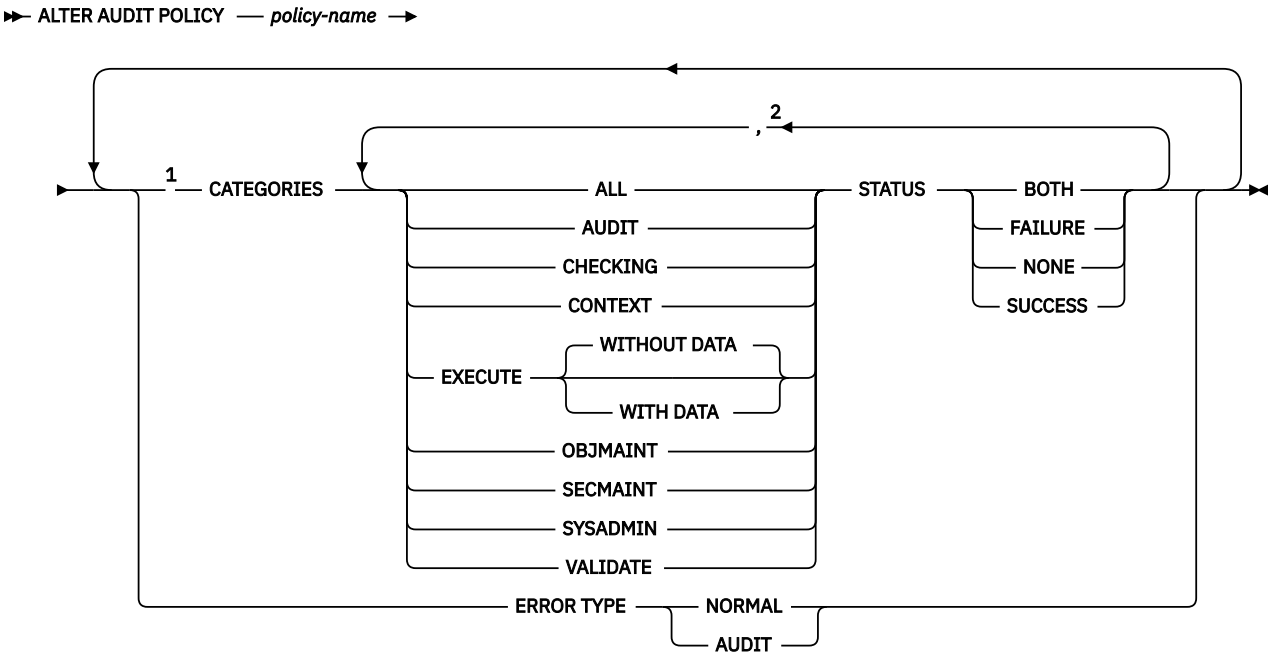
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include SECADM authority.

Syntax



Notes:

1 Each of the CATEGORIES and ERROR TYPE clauses can be specified at most once (SQLSTATE 42614).

2 Each category can be specified at most once (SQLSTATE 42614), and no other category can be specified if ALL is specified (SQLSTATE 42601).

Description

policy-name

Identifies the audit policy that is to be altered. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The name must uniquely identify an existing audit policy at the current server (SQLSTATE 42704).

CATEGORIES

A list of one or more audit categories for which a new status value is specified. If ALL is not specified, the STATUS of any category that is not explicitly specified remains unchanged.

ALL

Sets all categories to the same status. The EXECUTE category is WITHOUT DATA.

AUDIT
Generates records when audit settings are changed or when the audit log is accessed.

CHECKING
Generates records during authorization checking of attempts to access or manipulate database objects or functions.

CONTEXT
Generates records to show the operation context when a database operation is performed.

EXECUTE
Generates records to show the execution of SQL statements.

WITHOUT DATA or WITH DATA
Specifies whether or not input data values provided for any host variables and parameter markers should be logged as part of the EXECUTE category.

WITHOUT DATA
Input data values provided for any host variables and parameter markers are not logged as part of the EXECUTE category.

WITH DATA
Input data values provided for any host variables and parameter markers are logged as part of the EXECUTE category. Not all input values are logged; specifically, LOB, LONG, XML, and structured type parameters appear as the null value. Date, time, and timestamp fields are logged in ISO format. The input data values are converted to the database code page before being logged. If code page conversion fails, no errors are returned and the unconverted data is logged.

OBJMAINT
Generates records when data objects are created or dropped.

SECMAINT
Generates records when object privileges, database privileges, or DBADM authority is granted or revoked. Records are also generated when the database manager security configuration parameters **sysadm_group**, **sysctrl_group**, or **sysmaint_group** are modified.

SYSADMIN
Generates records when operations requiring SYSADM, SYSMAINT, or SYSCTRL authority are performed.

VALIDATE
Generates records when users are authenticated or when system security information related to a user is retrieved.

STATUS
Specifies a status for the specified category.

BOTH
Successful and failing events will be audited.

FAILURE
Only failing events will be audited.

SUCCESS
Only successful events will be audited.

NONE
No events in this category will be audited.

ERROR TYPE
Specifies whether audit errors are to be returned or ignored.

NORMAL
Any errors generated by the audit are ignored and only the SQLCODEs for errors associated with the operation being performed are returned to the application.

AUDIT
All errors, including errors occurring within the audit facility itself, are returned to the application.

Rules

- An AUDIT-exclusive SQL statement must be followed by a COMMIT or ROLLBACK statement (SQLSTATE 5U021). AUDIT-exclusive SQL statements are:
  - AUDIT
  - CREATE AUDIT POLICY, ALTER AUDIT POLICY, or DROP (AUDIT POLICY)
  - DROP (ROLE) or DROP (TRUSTED CONTEXT) if the role or trusted context is associated with an audit policy
- An AUDIT-exclusive SQL statement cannot be issued within a global transaction (SQLSTATE 51041) such as, for example, an XA transaction.

Notes

- Only one uncommitted AUDIT-exclusive SQL statement is allowed at a time across all database partitions. If an uncommitted AUDIT-exclusive SQL statement is executing, subsequent AUDIT-exclusive SQL statements wait until the current AUDIT-exclusive SQL statement commits or rolls back.
- Changes are written to the system catalog, but do not take effect until they are committed, even for the connection that issues the statement.
- If the audit policy that is being altered is currently associated with a database object, the changes do not take effect until the next unit of work for the application that is affected by the change. For example, if the audit policy is in use for the database, no current units of work will see the change to the policy until after a COMMIT or a ROLLBACK statement for that unit of work completes.

Example

Alter the SECMAINT, CHECKING, and VALIDATE categories of an audit policy named DBAUDPRF to audit both successes and failures.

```
ALTER AUDIT POLICY DBAUDPRF
  CATEGORIES SECMAINT STATUS BOTH,
             CHECKING STATUS BOTH,
             VALIDATE STATUS BOTH
```

ALTER BUFFERPOOL

The ALTER BUFFERPOOL statement is used to modify the characteristics or behavior of a buffer pool. There are a number of reasons to use the ALTER BUFFERPOOL statement, for example, to enable self-tuning memory.

The ALTER BUFFERPOOL statement can modify a buffer pool in the following ways:

- Modify the size of the buffer pool on all members or on a single member
- Enable or disable automatic sizing of the buffer pool
- Add this buffer pool definition to a new database partition group
- Modify the block area of the buffer pool for block-based I/O

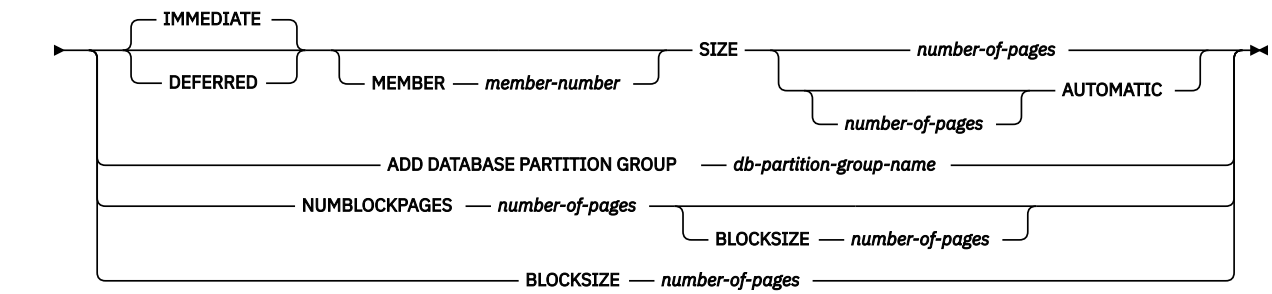
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include SYSCTRL or SYSADM authority.

Syntax



Description

bufferpool-name
Names the buffer pool. This is a one-part name. It is an SQL identifier (either ordinary or delimited). It must be a buffer pool described in the catalog.

IMMEDIATE or DEFERRED
Indicates whether or not the buffer pool size will be changed immediately.

IMMEDIATE
The buffer pool size will be changed immediately. If there is not enough reserved space in the database shared memory to allocate new space (SQLSTATE 01657), the statement is executed as DEFERRED.

DEFERRED
The buffer pool size will be changed when the database is reactivated (all applications need to be disconnected from the database). Reserved memory space is not needed; the database will allocate the required memory from the system at activation time.

MEMBER *member-number*
Specifies the member on which the size of the buffer pool is modified. An exception entry is created in the SYSCAT.BUFFERPOOLEXCEPTIONS catalog view. The member must be in one of the database partition groups for the buffer pool (SQLSTATE 42729). If this clause is not specified, the size of the buffer pool is modified on all members except those that have an exception entry in SYSCAT.BUFFERPOOLEXCEPTIONS.

SIZE
Specifies a new size for the buffer pool, or enables or disables self tuning for this buffer pool.

number-of-pages
The number of pages for the new buffer pool size. If the buffer pool is already a self-tuning buffer pool, and the SIZE *number-of-pages* clause is specified, the alter operation disables self-tuning for this buffer pool.

AUTOMATIC
Enables self tuning for this buffer pool. The database manager adjusts the size of the buffer pool in response to workload requirements. If the number of pages is specified, the current buffer pool size is set to that value unless the deferred keyword is also specified, in which case the number of pages will be ignored. On subsequent database activations, the buffer pool size is based on the last tuning value that is determined by the self-tuning memory manager (STMM). The STMM enforces a minimum size for automatic buffer pools, which is the minimum of the current size and 5000 pages. To determine the current size of buffer pools that are enabled for self tuning, use the MON_GET_BUFFERPOOL routine and examine the current size of the buffer pools. The size of the buffer pool is found in the**bp_cur_buffsz** monitor element. When AUTOMATIC is specified, the MEMBER clause cannot be specified (SQLSTATE 42601).

ADD DATABASE PARTITION GROUP *db-partition-group-name*
Adds this database partition group to the list of database partition groups to which the buffer pool definition is applicable. For any member in the database partition group that does not already have

the buffer pool defined, the buffer pool is created on the member using the default size specified for the buffer pool. Table spaces in *db-partition-group-name* may specify this buffer pool. The database partition group must currently exist in the database (SQLSTATE 42704).

NUMBLOCKPAGES ***number-of-pages***
Specifies the number of pages that should exist in the block-based area. The number of pages must not be greater than 98 percent of the number of pages for the buffer pool, based on the NPAGES value in SYSCAT.BUFFERPOOLS (SQLSTATE 54052). Specifying the value 0 disables block I/O. The actual value of NUMBLOCKPAGES used will be a multiple of BLOCKSIZE.

NUMBLOCKPAGES is not supported in a Db2 pureScale environment (SQLSTATE 56038).

BLOCKSIZE ***number-of-pages***
Specifies the number of pages in a block. The block size must be a value between 2 and 256 (SQLSTATE 54053). The default value is 32.

BLOCKSIZE is not supported in a Db2 pureScale environment (SQLSTATE 56038).

Notes

- Only the buffer pool size can be changed dynamically (immediately). All other changes are deferred, and will only come into effect after the database is reactivated.
- If the statement is executed as deferred, although the buffer pool definition is transactional and the changes to the buffer pool definition will be reflected in the catalog tables on commit, no changes to the actual buffer pool will take effect until the next time the database is started. The current attributes of the buffer pool will exist until then, and there will not be any impact to the buffer pool in the interim. Tables created in table spaces of new database partition groups will use the default buffer pool. The statement is IMMEDIATE by default when that keyword applies.
- There should be enough real memory on the machine for the total of all the buffer pools, as well as for the rest of the database manager and application requirements.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - DBPARTITIONNUM or NODE can be specified in place of MEMBER except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
  - NODEGROUP can be specified in place of DATABASE PARTITION GROUP

ALTER DATABASE PARTITION GROUP

The ALTER DATABASE PARTITION GROUP statement is used to add one or more database partitions to a database partition group, or drop one or more database partitions from a database partition group.

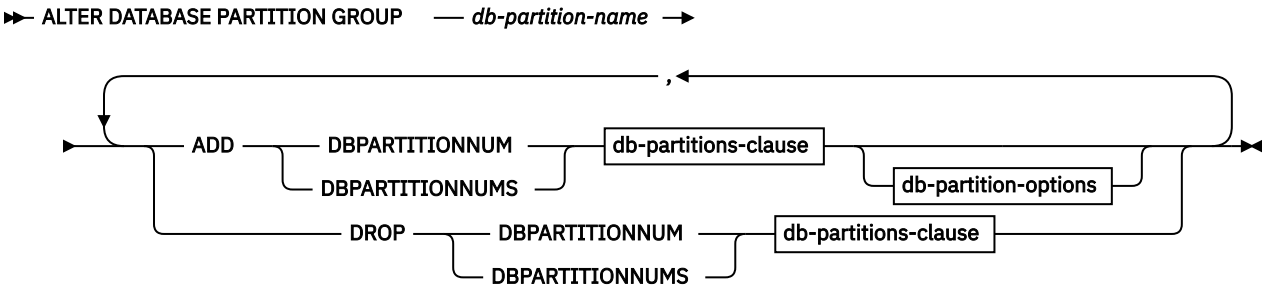
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

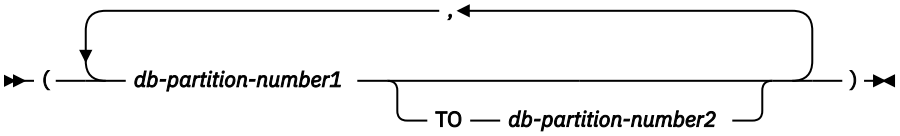
Authorization

The authorization ID of the statement must have SYSCTRL or SYSADM authority.

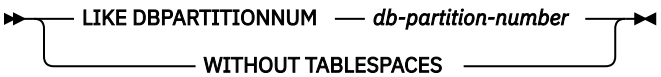
Syntax



db-partitions-clause



db-partition-options



Description

db-partition-name
Names the database partition group. This is a one-part name. It is an SQL identifier (either ordinary or delimited). It must be a database partition group described in the catalog. IBMCATGROUP and IBMTEMPGROUP cannot be specified (SQLSTATE 42832).

ADD DBPARTITIONNUM
Specifies the specific database partition or partitions to add to the database partition group. DBPARTITIONNUMS is a synonym for DBPARTITIONNUM. Any specified database partition must not already be defined in the database partition group (SQLSTATE 42728).

DROP DBPARTITIONNUM
Specifies the specific database partition or partitions to drop from the database partition group. DBPARTITIONNUMS is a synonym for DBPARTITIONNUM. Any specified database partition must already be defined in the database partition group (SQLSTATE 42729).

db-partitions-clause
Specifies the database partition or partitions to be added or dropped.

db-partition-number1
Specify a specific database partition number.

TO *db-partition-number2*
Specify a range of database partition numbers. The value of *db-partition-number2* must be greater than or equal to the value of *db-partition-number1* (SQLSTATE 428A9).

db-partition-options

LIKE DBPARTITIONNUM *db-partition-number*
Specifies that the containers for the existing table spaces in the database partition group will be the same as the containers on the specified *db-partition-number*. The specified database partition must be a partition that existed in the database partition group before this statement, and that is not included in a DROP DBPARTITIONNUM clause of the same statement.

For table spaces that are defined to use automatic storage (that is, table spaces that were created with the MANAGED BY AUTOMATIC STORAGE clause of the CREATE TABLESPACE statement, or for which no MANAGED BY clause was specified at all), the containers will not necessarily match those from the specified partition. Instead, containers will automatically be assigned by the database manager based on the storage paths that are associated with the database, and this

might or might not result in the same containers being used. The size of each table space is based on the initial size that was specified when the table space was created, and might not match the current size of the table space on the specified partition.

WITHOUT TABLESPACES
Specifies that the containers for existing table spaces in the database partition group are not created on the newly added database partition or partitions. The ALTER TABLESPACE statement using the *db-partitions-clause* or the MANAGED BY AUTOMATIC STORAGE clause must be used to define containers for use with the table spaces that are defined on this database partition group. If this option is not specified, the default containers are specified on newly added database partitions for each table space defined on the database partition group.

This option is ignored for table spaces that are defined to use automatic storage (that is, table spaces that were created with the MANAGED BY AUTOMATIC STORAGE clause of the CREATE TABLESPACE statement, or for which no MANAGED BY clause was specified at all). There is no way to defer container creation for these table spaces. Containers will automatically be assigned by the database manager based on the storage paths that are associated with the database. The size of each table space will be based on the initial size that was specified when the table space was created.

Rules

- Each database partition specified by number must be defined in the `db2nodes.cfg` file (SQLSTATE 42729).
- Each *db-partition-number* listed in the *db-partitions-clause* must be for a unique database partition (SQLSTATE 42728).
- A valid database partition number is between 0 and 999 inclusive (SQLSTATE 42729).
- A database partition cannot appear in both the ADD and DROP clauses (SQLSTATE 42728).
- There must be at least one database partition remaining in the database partition group. The last database partition cannot be dropped from a database partition group (SQLSTATE 428C0).
- If neither the LIKE DBPARTITIONNUM clause nor the WITHOUT TABLESPACES clause is specified when adding a database partition, the default is to use the lowest database partition number of the existing database partitions in the database partition group (say it is *2*) and proceed as if LIKE DBPARTITIONNUM *2* had been specified. For an existing database partition to be used as the default, it must have containers defined for all the table spaces in the database partition group (column IN_USE of SYSCAT.DBPARTITIONGROUPDEF is not 'T').
- The ALTER DATABASE PARTITION GROUP statement might fail (SQLSTATE 55071) if an add database partition server request is either pending or in progress. This statement might also fail (SQLSTATE 55077) if a new database partition server is added online to the instance and not all applications are aware of the new database partition server.

Notes

- When a database partition is added to a database partition group, a catalog entry is made for the database partition (see SYSCAT.DBPARTITIONGROUPDEF). The distribution map is changed immediately to include the new database partition, along with an indicator (IN_USE) that the database partition is in the distribution map if either:
  - no table spaces are defined in the database partition group or
  - no tables are defined in the table spaces defined in the database partition group and the WITHOUT TABLESPACES clause was not specified.

  The distribution map is not changed and the indicator (IN_USE) is set to indicate that the database partition is not included in the distribution map if either:
  - Tables exist in table spaces in the database partition group or

  - Table spaces exist in the database partition group and the WITHOUT TABLESPACES clause was specified (unless all of the table spaces are defined to use automatic storage, in which case the WITHOUT TABLESPACES clause is ignored)

  To change the distribution map, the REDISTRIBUTE DATABASE PARTITION GROUP command must be used. This redistributes any data, changes the distribution map, and changes the indicator. Table space containers need to be added before attempting to redistribute data if the WITHOUT TABLESPACES clause was specified.
- When a database partition is dropped from a database partition group, the catalog entry for the database partition (see SYSCAT.DBPARTITIONGROUPDEF) is updated. If there are no tables defined in the table spaces defined in the database partition group, the distribution map is changed immediately to exclude the dropped database partition and the entry for the database partition in the database partition group is dropped. If tables exist, the distribution map is not changed and the indicator (IN_USE) is set to indicate that the database partition is waiting to be dropped. The REDISTRIBUTE DATABASE PARTITION GROUP command must be used to redistribute the data and drop the entry for the database partition from the database partition group.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - NODE can be specified in place of DBPARTITIONNUM
  - NODES can be specified in place of DBPARTITIONNUMS
  - NODEGROUP can be specified in place of DATABASE PARTITION GROUP

Example

Assume that you have a six-partition database that has the following database partitions: 0, 1, 2, 5, 7, and 8. Two database partitions (3 and 6) are added to the system.

- *Example 1*: Assume that you want to add database partitions 3 and 6 to a database partition group called MAXGROUP, and have table space containers like those on database partition 2. The statement is as follows:

```
ALTER DATABASE PARTITION GROUP MAXGROUP
  ADD DBPARTITIONNUMS (3,6)LIKE DBPARTITIONNUM 2
```

- *Example 2*: Assume that you want to drop database partition 1 and add database partition 6 to database partition group MEDGROUP. You will define the table space containers separately for database partition 6 using ALTER TABLESPACE. The statement is as follows:

```
ALTER DATABASE PARTITION GROUP MEDGROUP
  ADD DBPARTITIONNUM(6)WITHOUT TABLESPACES
  DROP DBPARTITIONNUM(1)
```

ALTER DATABASE

The ALTER DATABASE statement adds new storage paths to, or removes existing storage paths from, the collection of paths that are used for automatic storage table spaces.

An automatic storage table space is a table space that has been created using automatic storage; that is, the MANAGED BY AUTOMATIC STORAGE clause has been specified on the CREATE TABLESPACE statement, or no MANAGED BY clause has been specified at all. If a database is enabled for automatic storage, container and space management characteristics of its table spaces can be completely determined by the database manager. If the database is not currently enabled for automatic storage then the act of adding storage paths will enable it.

Important: This statement is deprecated and might be removed in a future release. Use the CREATE STOGROUP or ALTER STOGROUP statements instead.

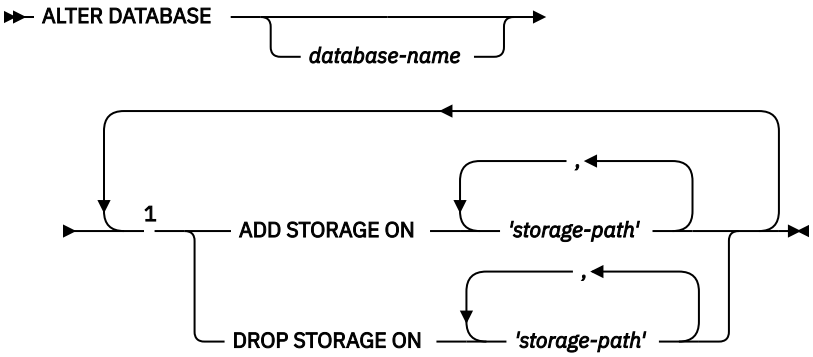
Invocation

The statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include either SYSADM or SYSCTRL authority.

Syntax



Notes:

1 Each clause can be specified only once.

Description

database-name
An optional value specifying the name of the database that is to be altered. If specified, the value must match the name of the database to which the application is currently connected (not the alias that the client might have cataloged); otherwise, an error is returned (SQLSTATE 42961).

ADD STORAGE ON
Specifies that one or more new storage paths are to be added to the collection of storage paths that are used for automatic storage table spaces.

'storage-path'
A string constant that specifies the location where containers for automatic storage table spaces are to be created. The format of the string depends on the operating system, as illustrated in the following table:

Operating system	String format
Linux AIX	An absolute path.
Windows	The letter name of a drive.

DROP STORAGE ON
Specifies that one or more storage paths are to be removed from the collection of storage paths that are used for automatic storage table spaces. If table spaces are actively using a storage path being dropped, then the state of the storage path is changed from "In Use" to "Drop Pending" and future use of the storage path will be prevented.

'storage-path'
A string constant that specifies the location from which storage paths are to be removed. The format of the string depends on the operating system, as illustrated in the following table:

Operating system	String format
Linux AIX	An absolute path.
Windows	The letter name of a drive.

Rules

- For a database that is running on version 10.1 or later, the operations of this statement are applied to the default storage group for the database. If no storage group is defined for the database, the name IBMSTOGROUP is used.
- A storage path being added, must be valid according to the naming rules for paths, and must be accessible (SQLSTATE 57019). Similarly, in a partitioned database environment, the storage path must exist and be accessible on every database partition (SQLSTATE 57019).
- A storage path being dropped must currently exist in the database (SQLSTATE 57019) and cannot already be in the "Drop Pending" state (SQLSTATE 55073).
- A database enabled for automatic storage must have at least one storage path. Dropping all storage paths from the database is not permitted (SQLSTATE 428HH).
- The ALTER DATABASE statement cannot be executed while a database partition server is being added (SQLSTATE 55071).
- DROP STORAGE ON cannot be specified in a Db2 pureScale environment (SQLSTATE 56038).

Notes

- When adding new storage paths:
  - Existing regular and large table spaces using automatic storage will not initially use these new paths. The database manager might choose to create new table space containers on these paths only if an out-of-space condition occurs.
  - Existing temporary table spaces managed by automatic storage do not automatically use new storage paths. The database must be stopped normally then restarted for containers in these table spaces to use the new storage path or paths. As an alternative, the temporary table spaces can be dropped and recreated. When created, these table spaces automatically use all storage paths that have sufficient free space.
- Adding storage paths to the database to enable automatic storage will not cause the database to convert existing non-automatic storage enabled table spaces to use automatic storage.
- Although ADD STORAGE and DROP STORAGE are logged operations, whether they are redone during a rollforward operation depends on how the database was restored. If the restore operation does not redefine the storage paths that are associated with the database, the log record that contains the storage path change is redone, and the storage paths that are described in the log record are added or dropped during the rollforward operation. However, if the storage paths *are* redefined during the restore operation, the rollforward operation will not redo ADD STORAGE or DROP STORAGE log records, because it is assumed that you have already set up the storage paths.
- When free space is calculated for a storage path on a database partition, the database manager checks for the existence of the following directories or mount points within the storage path, and will use the first one that is found.

```
<storage path>/<instance name>/NODE####/<database name>
<storage path>/<instance name>/NODE####
<storage path>/<instance name>
<storage path>
```

  Where:

  - <storage path> is a storage path associated with the database
  - <instance name> is the instance under which the database resides

  - NODE#### corresponds to the database partition number (for example, NODE0000 or NODE0001)
  - <database name> is the name of the database

  File systems can be mounted at a point beneath the storage path, and the database manager will recognize that the actual amount of free space available for table space containers might not be the same amount that is associated with the storage path directory itself.

  Consider an example in which two logical database partitions exist on one physical machine, and there is a single storage path (/dbdata). Each database partition will use this storage path, but you might want to isolate the data from each partition within its own file system. In this case, a separate file system can be created for each partition and it can be mounted at /dbdata/<instance>/NODE####. When creating containers on the storage path and determining free space, the database manager will not retrieve free space information for /dbdata, but instead will retrieve it for the corresponding /dbdata/<instance>/NODE#### directory.
- In general, the same storage paths must be used for each partition in a partitioned database environment. One exception to this is the case in which database partition expressions are used within the storage path. Doing this allows the database partition number to be reflected in the storage path, such that the resulting path name is different on each partition.
- When dropping a storage path that is in use by one or more table spaces, the state of the path changes from "In Use" to "Drop Pending". Future growth on the path will not occur. Before the path can be fully removed from the database, each affected table space must be rebalanced (using the REBALANCE clause of the ALTER TABLESPACE statement) so that its container data is moved off the storage path. Rebalance is only supported for regular and large table spaces. Temporary table spaces should be dropped and recreated to have their containers removed from the dropped path. When the path is no longer in use by any table space, it will be physically removed from the database.

  For a partitioned database, the path is maintained independently on each partition. When a path is no longer in use on a given database partition, it will be physically removed from that partition. Other partitions may still show the path as being in the "Drop Pending" state.

  The list of automatic storage table spaces using drop pending storage paths can be determined by issuing the following SQL statement:

```
SELECT DISTINCT A.TBSP_NAME, A.TBSP_ID, A.TBSP_CONTENT_TYPE
  FROM TABLE(MON_GET_TABLESPACE(NULL,-2)) AS A
  WHERE A.TBSP_PATHS_DROPPED = 1
```

- When dropping a storage path that was originally specified using a database partition expression, the same storage path string, including the database partition expression, must be used in the drop. If a database partition expression was specified then this path string can be found in the "Path with db partition expression" element (db_storage_path_with_dpe) of a database snapshot. This element is not shown if a database partition expression was not included in the original path specified.
- It is possible for a given storage path to be added to a database multiple times. When using the DROP STORAGE ON clause, specifying that particular path once will drop *all* instances of the path from the database.

Examples

1. Add two paths under the /db directory (/db/filesystem1 and /db/filesystem2) and a third path named /filesystem3 to the space for automatic storage table spaces that is associated with the currently connected database.

   ```
   ALTER DATABASE ADD STORAGE ON '/db/filesystem1', '/db/filesystem2',
     '/filesystem3'
   ```

2. Add drives D and E to the space for automatic storage table spaces that is associated with the SAMPLE database.

   ```
   ALTER DATABASE SAMPLE ADD STORAGE ON 'D:', 'E:\'
   ```

3. Add directory F:\DBDATA and drive G to the space for automatic storage table spaces that is associated with the currently connected database.

```
ALTER DATABASE ADD STORAGE ON 'F:\DBDATA', 'G:'
```

4. Add a storage path that uses a database partition expression to differentiate the storage paths on each of the database partitions.

```
ALTER DATABASE ADD STORAGE ON '/dataForPartition $N'
```

   The storage path that would be used on database partition 0 is /dataForPartition0; on database partition 1, it would be /dataForPartition1; and so on.

5. Add storage paths to a database that is not automatic storage enabled, for the purposes of enabling automatic storage for the database.

```
CREATE DATABASE MYDB AUTOMATIC STORAGE NO
CONNECT TO MYDB
ALTER DATABASE ADD STORAGE ON '/db/filesystem1', '/db/filesystem2'
```

   Database MYDB is now enabled for automatic storage.

6. Remove paths /db/filesystem1 and /db/filesystem2 from the currently connected database.

```
ALTER DATABASE DROP STORAGE ON '/db/filesystem1', '/db/filesystem2'
```

   After the storage is dropped successfully, use the ALTER TABLESPACE statement with the REBALANCE clause for each table space that was using these storage paths to rebalance the table space.

7. A storage path with a database partition expression (/dataForPartition $N) was previously added to the database and now it is to be removed.

```
ALTER DATABASE DROP STORAGE ON '/dataForPartition $N'
```

   After the storage is dropped successfully, use the ALTER TABLESPACE statement with the REBALANCE clause for each table space that was using these storage paths to rebalance the table space.

ALTER EVENT MONITOR

The ALTER EVENT MONITOR statement alters the definition of an event monitor that has a target for the event monitor data of TABLE.

Invocation

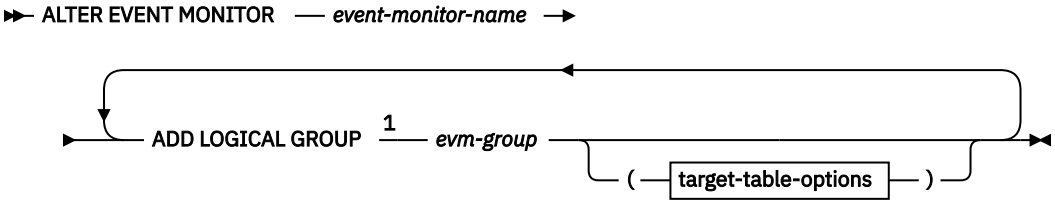
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

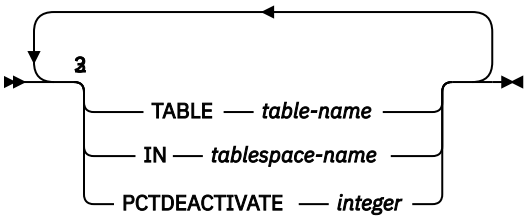
The privileges held by the authorization ID of the statement must include one of the following authorities:

- DBADM authority
- SQLADM authority

Syntax



target-table-options



Notes:

[1] A logical group can be added only to TABLE event monitors (not UNFORMATTED EVENT TABLE event monitors).

[2] Each clause can be specified only once.

[3] Clauses can be separated with a space or a comma.

Description

event-monitor-name

The *event-monitor-name* must identify an event monitor that exists at the current server and has a target for the event monitor data of TABLE.

ADD LOGICAL GROUP

Adds a logical group to the event monitor that has a target for the data of TABLE.

evm-group

Identifies the logical data group for which a target table is being added. The value depends upon the type of event monitor, as shown in the following table:

Table 123. Values for evm-group based on the type of event monitor

Type of Event Monitor	evm-group value
Database	• DB • CONTROL[1] • DBMEMUSE
Tables	• TABLE • CONTROL[1]
Deadlocks	• CONNHEADER • DEADLOCK • DLCONN • CONTROL[1]

Table 123. Values for evm-group based on the type of event monitor (continued)

Type of Event Monitor	evm-group value
Deadlocks with details	• CONNHEADER • DEADLOCK • DLCONN[2] • DLLOCK[3] • CONTROL[1]
Deadlocks with details history	• CONNHEADER • DEADLOCK • DLCONN[2] • DLLOCK[3] • STMTHIST • CONTROL[1]
Deadlocks with details history values	• CONNHEADER • DEADLOCK • DLCONN[2] • DLLOCK[3] • STMTHIST • STMTVALS • CONTROL[1]
Table spaces	• TABLESPACE • CONTROL[1]
Buffer pools	• BUFFERPOOL • CONTROL[1]
Connections	• CONNHEADER • CONN • CONTROL[1] • CONNMEMUSE
Statements	• CONNHEADER • STMT • SUBSECTION[4] • CONTROL[1]
Transactions	• CONNHEADER • XACT • CONTROL[1]

Table 123. Values for evm-group based on the type of event monitor (continued)

Type of Event Monitor	evm-group value
Activities	• ACTIVITY • ACTIVITYMETRICS • ACTIVITYSTMT • ACTIVITYVALS • CONTROL[1]
Statistics	• QSTATS • SCSTATS • SCMETRICS • WCSTATS • WLSTATS • WLMETRICS • HISTOGRAMBIN • CONTROL[1]
Threshold Violations	• THRESHOLDVIOLATIONS • CONTROL[1]
Locking[5]	• LOCK • LOCK_PARTICIPANTS • LOCK_PARTICIPANT_ACTIVITIES • LOCK_ACTIVITY_VALUES • CONTROL[1]
Package Cache[5]	• PKGCACHE • PKGCACHE_METRICS • CONTROL[1]
Unit of Work[5]	• UOW • UOW_METRICS • UOW_PACKGE_LIST • UOW_EXECUTABLE_LIST • CONTROL[1]

Table 123. Values for evm-group based on the type of event monitor (continued)

Type of Event Monitor	evm-group value
Change History	• CHANGESUMMARY • EVMONSTART • TXNCOMPLETION • DDLSTMTEXEC • DBDBMCFG • REGVAR • UTILSTART • UTILSTOP • UTILPHASE • UTILLOCATION • CONTROL[1]

[1] Logical data groups dbheader (conn_time element only), start, and overflow, are all written to the CONTROL group. The overflow group is written if the event monitor is non-blocked and events were discarded.

[2] Corresponds to the DETAILED_DLCONN event.

[3] Corresponds to the LOCK logical data groups that occur within each DETAILED_DLCONN event.

[4] Created only for partitioned database environments.

[5] Refers to the Formatted Event Table version of this event monitor type.

TABLE ***table-name***
Specifies the name of the target table. The target table must be a non-partitioned table. If the name is unqualified, the table schema defaults to the value in the CURRENT SCHEMA special register. If no name is provided, the unqualified name is derived from *evm-group* and *event-monitor-name* as follows:

```
substring(evm-group CONCAT '_'
   CONCAT event-monitor-name,1,128)
```

IN ***tablespace-name***
Defines the table space in which the table is to be created. If no table space name is provided, the table space is chosen using the same process as when a table is created without a table space name using the CREATE TABLE statement.

When specifying the table space name for an activities, locking, package cache, or unit of work event monitor, the table space's page size affects the INLINE LOB lengths used. Therefore, consider specifying a table space with as large a page size as possible to improve the INSERT performance of the event monitor.

PCTDEACTIVATE ***integer***
If a table is being created in a DMS table space, PCTDEACTIVATE specifies how full the table space must be before the event monitor automatically deactivates. The specified value, which represents a percentage, can range from 0 to 100. The default value is 100 (meaning that the event monitor deactivates when the table space becomes completely full). This option is ignored for SMS table spaces. When a target table space has auto-resize enabled, it is recommended that PCTDEACTIVATE be set to 100.

Notes

- **When system catalog changes take effect:** Changes are written to the system catalog, but do not take effect until they are committed and the event monitor is reactivated.

Example

The event monitor ACT is missing the ACTIVITYMETRICS group. Alter the event monitor to add this group and give the table the name "ACTMETRICS".

```
ALTER EVENT MONITOR ACT
   ADD LOGICAL GROUP ACTIVITYMETRICS TABLE ACTMETRICS
```

ALTER FUNCTION

The ALTER FUNCTION statement modifies the properties of an existing function.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- ALTERIN privilege on the schema of the function
- Owner of the function, as recorded in the OWNER column of the SYSCAT.ROUTINES catalog view
- SCHEMAADM authority on the schema of the function
- DBADM authority

To alter the EXTERNAL NAME of a function, the privileges held by the authorization ID of the statement must also include at least one of the following authorities:

- CREATE_EXTERNAL_ROUTINE authority on the database
- DBADM authority

To alter a function to be not fenced, the privileges held by the authorization ID of the statement must also include at least one of the following authorities:

- CREATE_NOT_FENCED_ROUTINE authority on the database
- DBADM authority

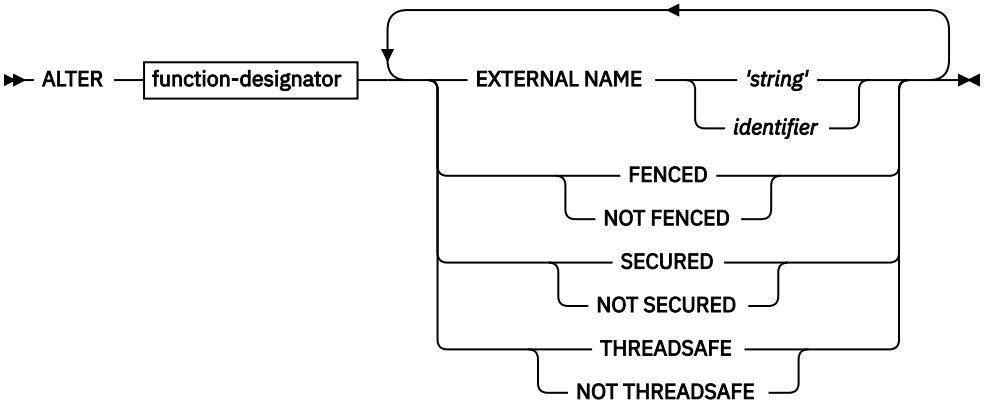
To alter a function to be fenced, no additional authorities or privileges are required.

To alter a function to be SECURED or NOT SECURED the privileges held by the authorization ID of the statement must include at least one of the following authorities:

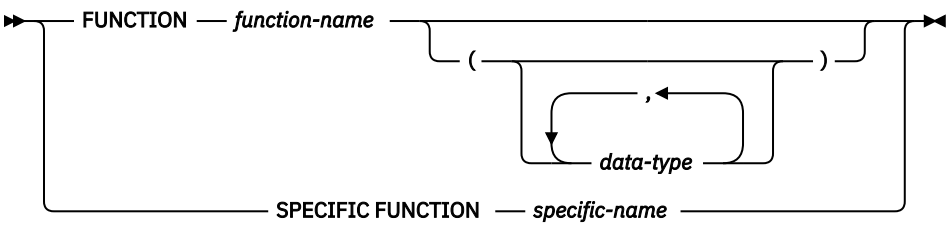
- SECADM authority
- CREATE_SECURE_OBJECT authority

If no other clauses are specified, then no other privileges are required to process the statement.

Syntax



function-designator



Description

function-designator
Uniquely identifies the function to be altered. For more information, see "Function, method, and procedure designators" on page 745.

EXTERNAL NAME *'string'* or *identifier*
Identifies the name of the user-written code that implements the function. This option can only be specified when altering external functions (SQLSTATE 42849).

FENCED or NOT FENCED
Specifies whether the function is considered safe to run in the database manager operating environment's process or address space (NOT FENCED), or not (FENCED). Most functions have the option of running as FENCED or NOT FENCED.

If a function is altered to be FENCED, the database manager insulates its internal resources (for example, data buffers) from access by the function. In general, a function running as FENCED will not perform as well as a similar one running as NOT FENCED.

CAUTION: Use of NOT FENCED for functions that were not adequately coded, reviewed, and tested can compromise the integrity of a Db2 database. Db2 databases take some precautions against many of the common types of inadvertent failures that might occur, but cannot guarantee complete integrity when NOT FENCED user-defined functions are used.

A function declared as NOT THREADSAFE cannot be altered to be NOT FENCED (SQLSTATE 42613).

If a function has any parameters defined AS LOCATOR, and was defined with the NO SQL option, the function cannot be altered to be FENCED (SQLSTATE 42613).

This option cannot be altered for LANGUAGE OLE, OLEDB, or CLR functions (SQLSTATE 42849).

This option cannot be altered for a function that is registered as a component routine of an aggregate interface function (SQLSTATE 42849).

SECURED or NOT SECURED
Specifies whether the function is considered secure for row and column access control.

NOT SECURED
Indicates that the function is not considered secure. When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled and column level access control is activated for its table (SQLSTATE 428HA). This rule applies to the non secure user-defined functions that are invoked anywhere in the statement.

SECURED
Indicates that the function is considered secure.

The function must be secure when it is referenced in a row permission or a column mask (SQLSTATE 428H8).

The function must be secure when it is referenced in a materialized query table and the materialized query table references any table that has row or column level access control activated (SQLSTATE 428H8).

This option cannot be altered for a function that is registered as a component routine of an aggregate interface function (SQLSTATE 42849).

THREADSAFE or NOT THREADSAFE
Specifies whether the function is considered safe to run in the same process as other routines (THREADSAFE), or not (NOT THREADSAFE).

If the function is defined with LANGUAGE other than OLE and OLEDB:

- If the function is defined as THREADSAFE, the database manager can invoke the function in the same process as other routines. In general, to be threadsafe, a function should not use any global or static data areas. Most programming references include a discussion of writing threadsafe routines. Both FENCED and NOT FENCED functions can be THREADSAFE.
- If the function is defined as NOT THREADSAFE, the database manager will never simultaneously invoke the function in the same process as another routine. Only a fenced function can be NOT THREADSAFE (SQLSTATE 42613).

This option may not be altered for LANGUAGE OLE or OLEDB functions (SQLSTATE 42849).

This option cannot be altered for a function that is registered as a component routine of an aggregate interface function (SQLSTATE 42849).

Notes

- It is not possible to alter a function that is in the following schema (SQLSTATE 42832):
  - SYSIBM
  - SYSFUN
  - SYSPROC
- Functions declared as LANGUAGE SQL, sourced functions, or template functions cannot be altered (SQLSTATE 42917).
- ***Altering a function from NOT SECURED to SECURED***: Normally users with SECADM authority do not have privileges to alter database objects such as user-defined functions and triggers. Typically they will examine the actions taken by a function, ensure it is secure, then grant the CREATE_SECURE_OBJECT authority to someone who has required privileges to alter the user-defined function to be secure. After the function is altered, they will revoke the CREATE_SECURE_OBJECT authority from the user who was granted this authority.

  The function is considered secure. The SECURED attribute is considered to be an assertion that declares the user has established a change control audit procedure for all changes to the user-defined function. The database manager assumes that such a control audit procedure is in place for all subsequent ALTER FUNCTION statements or changes to external packages.

  Packages and dynamically cached SQL statements that depend on the function might be invalidated because the secure attribute affects the access path selection for statements involving tables for which row or column level access control is activated and the function being replaced.

- ***Altering a function from SECURED to NOT SECURED***: The function is considered not secure. Packages and dynamically cached SQL statements that depend on the function might be invalidated because the secure attribute affects the access path selection for statements involving tables for which row or column level access control is activated.
- ***Invoking other user-defined functions in a secure function***: When a secure user-defined function is referenced in a data manipulation statement where a row or column access control enforced table is referenced, if the secure user-defined function invokes other user-defined functions, the database manager does not validate whether those nested user-defined functions are secure. If those nested functions can access sensitive data, the user with SECADM authority needs to ensure those functions are allowed to access those data and a change control audit procedure has been established for all changes to those functions.

Example

The function MAIL() has been thoroughly tested. To improve its performance, alter the function to be not fenced.

```
ALTER FUNCTION MAIL() NOT FENCED
```

ALTER HISTOGRAM TEMPLATE

The ALTER HISTOGRAM TEMPLATE statement is used to modify the template describing the type of histogram that can be used to override one or more of the default histograms of a service class or a work class.

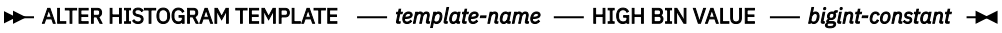
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include WLMADM or DBADM authority.

Syntax



Description

template-name
Names the histogram template. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The name must identify an existing histogram template at the current server (SQLSTATE 42704). The template name can be the default system histogram template SYSDEFAULTHISTOGRAM.

HIGH BIN VALUE ***bigint-constant***
Specifies the top value of the second to last bin (the last bin has an unbounded top value). The units depend on how the histogram is used. The maximum value is 268 435 456.

Rules

- A workload management (WLM)-exclusive SQL statement must be followed by a COMMIT or a ROLLBACK statement (SQLSTATE 5U021). WLM-exclusive SQL statements are:
  - CREATE HISTOGRAM TEMPLATE, ALTER HISTOGRAM TEMPLATE, or DROP (HISTOGRAM TEMPLATE)
  - CREATE SERVICE CLASS, ALTER SERVICE CLASS, or DROP (SERVICE CLASS)
  - CREATE THRESHOLD, ALTER THRESHOLD, or DROP (THRESHOLD)

  - CREATE WORK ACTION SET, ALTER WORK ACTION SET, or DROP (WORK ACTION SET)
  - CREATE WORK CLASS SET, ALTER WORK CLASS SET, or DROP (WORK CLASS SET)
  - CREATE WORKLOAD, ALTER WORKLOAD, or DROP (WORKLOAD)
  - GRANT (Workload Privileges) or REVOKE (Workload Privileges)
- A WLM-exclusive SQL statement cannot be issued within a global transaction (SQLSTATE 51041) such as, for example, an XA transaction.

Notes

- Only one uncommitted WLM-exclusive SQL statement at a time is allowed across all partitions. If an uncommitted WLM-exclusive SQL statement is executing, subsequent WLM-exclusive SQL statements will wait until the current WLM-exclusive SQL statement commits or rolls back.
- Changes are written to the system catalog, but do not take effect until they are committed, even for the connection that issues the statement.

Example

Change the high bin value of a histogram template named LIFETIMETEMP.

```
ALTER HISTOGRAM TEMPLATE LIFETIMETEMP
  HIGH BIN VALUE 90000
```

ALTER INDEX

The ALTER INDEX statement alters the definition of an index.

Invocation

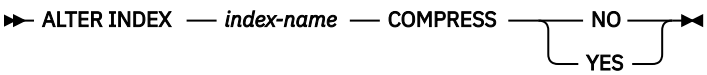
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- ALTERIN privilege on the schema of the index
- ALTER privilege on the table on which the index is defined
- CONTROL privilege on the index
- SCHEMAADM authority on the schema of the index
- DBADM authority

Syntax



Description

INDEX *index-name*
Identifies the index to be altered. The name must identify an index that exists at the current server (SQLSTATE 42704).

COMPRESS
Specifies whether index compression is to be enabled or disabled. The index must not be an MDC or ITC block index, catalog index, XML path index, index specification, or an index on a created temporary table or declared temporary table (SQLSTATE 56090).

NO
Specifies that index compression is disabled. A compressed index will remain compressed until the index is rebuilt via index reorganization or recreation.

YES
Specifies that index compression is enabled. An uncompressed index will remain uncompressed until the index is rebuilt via index reorganization or recreation.

Example

Alter index JOB_BY_DPT to be compressed index.

```
ALTER INDEX JOB_BY_DPT
COMPRESS YES
```

Related reference
“CREATE INDEX ” on page 1240
The CREATE INDEX statement is used to define an index on a database table.

Related information
BIND command

ALTER MASK

The ALTER MASK statement alters a column mask that exists at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privileges held by the authorization ID of the statement must include SECADM authority.

Syntax

Description

mask-name
Identifies the column mask to be altered. The name must identify a mask that exists at the current server (SQLSTATE 42704).

ENABLE

Enables the column mask. If column level access control is not currently activated on the table, the column mask will become effective when column level access control is activated on the table. If column level access control is currently activated on the table, the column mask becomes effective immediately and all packages and dynamically cached statements that reference the table are invalidated.

ENABLE is ignored if the column mask is already enabled.

DISABLE

Disables the column mask. If column level access control is not currently activated on the table, the column mask will remain ineffective when column level access control is activated on the table. If column level access control is currently activated on the table, the column mask becomes ineffective immediately and all packages and dynamically cached statements that reference the table are invalidated.

DISABLE is ignored if the column mask is already disabled.

Examples

- *Example 1*: Enable column mask M1.

```
ALTER MASK M1 ENABLE
```

- *Example 2*: Disable column mask M1.

```
ALTER MASK M1 DISABLE
```

ALTER METHOD

The ALTER METHOD statement modifies an existing method by changing the method body associated with the method.

Invocation

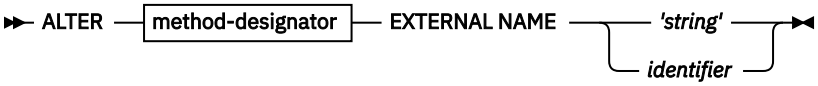
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

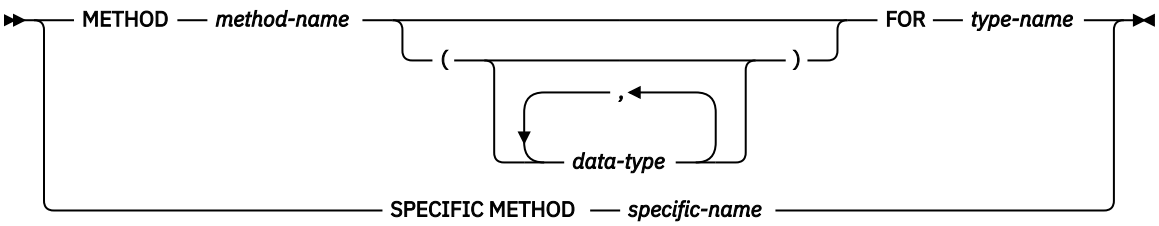
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- CREATE_EXTERNAL_ROUTINE authority on the database, and at least one of:
  - ALTERIN privilege on the schema of the type
  - SCHEMAADM privilege on the schema of the type
  - Owner of the type, as recorded in the OWNER column of the SYSCAT.DATATYPES catalog view
- DBADM authority

Syntax



method-designator



Description

method-designator
Uniquely identifies the method to be altered. For more information, see "Function, method, and procedure designators" on page 745.

EXTERNAL NAME *'string'* or *identifier*
Identifies the name of the user-written code that implements the method. This option can only be specified when altering external methods (SQLSTATE 42849).

Notes

- It is not possible to alter a method that is in the SYSIBM, SYSFUN, or SYSPROC schema (SQLSTATE 42832).
- Methods declared as LANGUAGE SQL cannot be altered (SQLSTATE 42917).
- Methods declared as LANGUAGE CLR cannot be altered (SQLSTATE 42849).
- The specified method must have a body before it can be altered (SQLSTATE 42704).

Example

Alter the method DISTANCE() in the structured type ADDRESS_T to use the library newaddresslib.

```
ALTER METHOD DISTANCE()
  FOR ADDRESS_T
  EXTERNAL NAME 'newaddresslib!distance2'
```

ALTER MODULE

The ALTER MODULE statement alters the definition of a module.

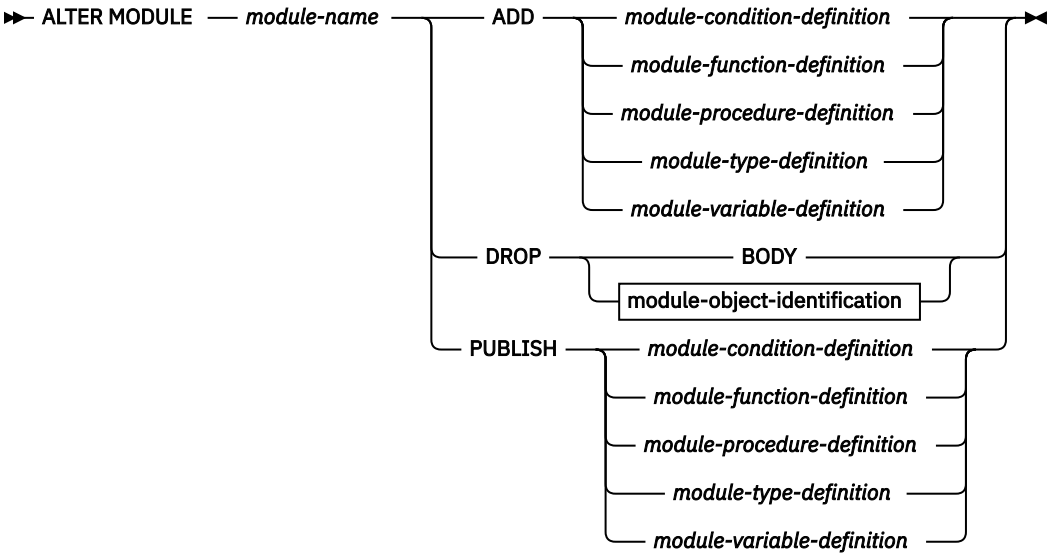
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

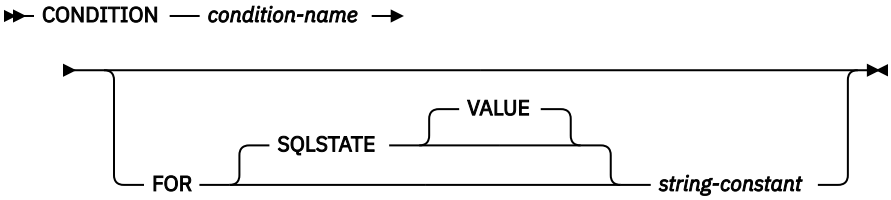
Authorization

The privileges held by the authorization ID of the statement must include ownership of the module and also include all of the privileges necessary to invoke the SQL statements that are specified within the ALTER MODULE statement.

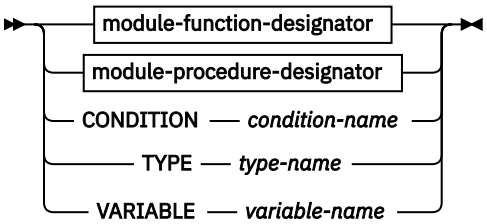
Syntax



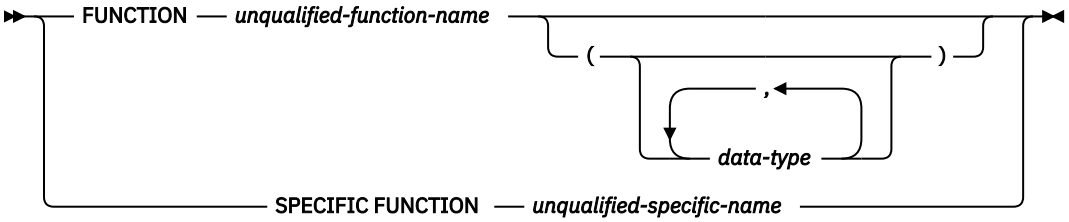
module-condition-definition



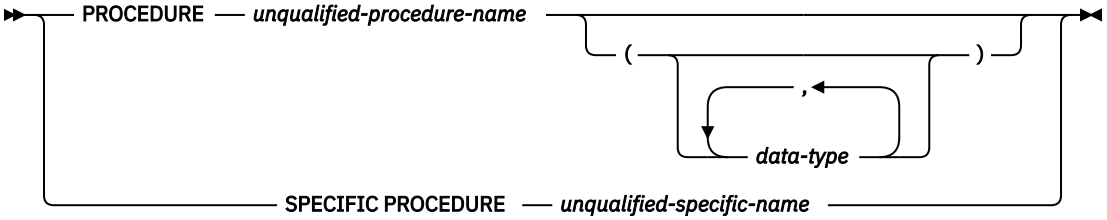
module-object-identification



module-function-designator



module-procedure-designator



Description

module-name
Identifies the module to be altered. The module-name must identify a module that exists at the current server (SQLSTATE 42704). The specified name must not be an alias for a module (SQLSTATE 560CT).

ADD
Adds an object to the module or adds the body to a routine definition that already exists in the module without a body. If adding a user-defined type or a global variable, the object must not identify a user-defined type or global variable that already exists in the module. If the user-defined type or global variable did not exist, it is added to the module for use within the module only.

If adding a routine and the specified routine does not exist, the routine is added. If adding a routine and the specified routine exists, the existing routine definition must not include a routine body (SQLSTATE 42723). This routine prototype is completely replaced by the new routine definition, including the routine attributes and the routine body, except that the published attribute is retained. The specified routine is considered to exist if one of the following conditions is true:

- There is a routine in the module with the same specific name and same routine name.
- The specified routine is a procedure and there is a procedure in the module with the same procedure name and the same number of parameters. The names and data types of the parameters do not need to match.
- The specified routine is a function and there is a function in the module with the same function name and the same number of parameters with matching data types. The length, precision, and scale of parameter data types are not compared and can be different when determining if the specified routine exists. The names of the parameters do not need to match.

module-condition-definition
Adds a module condition.

condition-name
Name of the condition. The name must not identify an existing condition in the module. The condition-name must be specified without any qualification (SQLSTATE 42601). The name of the condition must be unique within the module.

FOR SQLSTATE *string-constant*
Specifies the SQLSTATE that is associated with the condition. The string-constant must be specified as five characters enclosed in single quotation marks, and the SQLSTATE class (the first two characters) must not be '00'. This is an optional clause.

module-function-definition
The syntax to add a function is the same as the CREATE FUNCTION statement excluding the CREATE keyword and both the function-name and specific-name must be specified without any qualification (SQLSTATE 42601). If the function is unique within the module, a new function is added. If the function matches an existing function that does not include a body (SQL-routine-body or EXTERNAL NAME clause), then this function prototype is replaced by the new definition except that the published attribute is retained.

The module function definition must not specify the SOURCE clause, the TEMPLATE clause, or the LANGUAGE OLEDEB option (SQLSTATE 42613).

module-procedure-definition
The syntax to define the procedure is the same as the CREATE PROCEDURE statement excluding the CREATE keyword and both the procedure-name and specific-name must be specified without any qualification (SQLSTATE 42601). If the procedure signature is unique within the module, a new procedure is added. If the procedure matches an existing procedure that does not include a body (SQL-routine-body or EXTERNAL NAME clause), then this procedure prototype is replaced by the new definition except that the published attribute is retained. The name of the procedure can begin with "SYS_" only to add the module initialization procedure called SYS_INIT. See Notes for details.

module-type-definition
The syntax to define the user-defined type is the same as the CREATE TYPE statement excluding the CREATE keyword and the type-name must be specified without any qualification (SQLSTATE 42601). The name of the user-defined type must be unique within the module. A structured type cannot be defined in a module. Any generated functions required to support the type definition are also defined in the module. If the module user-defined type is published then so are the generated functions.

module-variable-definition
The syntax to define the variable is the same as the CREATE VARIABLE statement excluding the CREATE keyword and the variable-name must be specified without any qualification (SQLSTATE 42601). The name of the variable must be unique within the module.

DROP
Drops a specified part of a module. The module-object-identification syntax is used to identify the object to be dropped unless the body of the module is being dropped.

BODY
Drops the module body, which includes:

- all objects that are not published.
- the routine body of any published SQL routines
- the EXTERNAL reference for any published external routines.

PUBLISH
Adds a new object to the module and makes it available for use outside the module. In the case of routines, a routine prototype can be specified that does not include the executable body of the routine.

module-condition-definition
Adds a module condition that is available for use outside the module.

condition-name
Name of the condition. The name must not identify an existing condition in the module. The condition-name must be specified without any qualification (SQLSTATE 42601). The name of the condition must be unique within the module.

FOR SQLSTATE ***string-constant***
Specifies the SQLSTATE that is associated with the condition. The string-constant must be specified as five characters enclosed in single quotation marks, and the SQLSTATE class (the first two characters) must not be '00'. This is an optional clause.

module-function-definition
The syntax to define the function is the same as the CREATE FUNCTION statement excluding the CREATE keyword and both the function-name and specific-name must be specified without any qualification (SQLSTATE 42601). The definition of the function must include the function name, full specification of any parameters and the returns clause. Module user-defined data types that are not published are not candidates for the parameter data types or the RETURNS clause data type. Module variables that are not published are not candidates for the anchor object in an ANCHOR clause of a parameter data type or a returns data type. A function prototype can be specified by omitting the LANGUAGE clause (or specifying LANGUAGE SQL) and the SQL-routine-body. The function signature must be unique within the module. The name of the function must not begin with "SYS_" (SQLSTATE 42939). All SQL functions added to a module are processed as if a compound SQL (compiled) statement was used.

The module function definition can only specify the RETURNS TABLE clause when the SQL-routine-body is an compound SQL (compiled) statement that specifies NOT ATOMIC. The module function definition must not specify the SOURCE clause, the TEMPLATE clause, or the LANGUAGE OLEDEB option (SQLSTATE 42613).

module-procedure-definition
The syntax to define the procedure is the same as the CREATE PROCEDURE statement excluding the CREATE keyword and both the procedure-name and specific-name must be specified without

any qualification (SQLSTATE 42601). The definition of the procedure must include the procedure name and full specification of any parameters. Module user-defined data types that are not published are not candidates for the parameter data types. Module variables that are not published are not candidates for the anchor object in an ANCHOR clause of a parameter definition. A function prototype can be specified by omitting the LANGUAGE clause (or specifying LANGUAGE SQL) and the SQL-routine-body. The procedure signature must be unique within the module. The name of the procedure must not begin with "SYS_" (SQLSTATE 42939).

module-type-definition
The syntax to define the user-defined type is the same as the CREATE TYPE statement excluding the CREATE keyword and the type-name must be specified without any qualification (SQLSTATE 42601). Module user-defined data types that are not published are not candidates for any data type referenced in the module user-defined data type definition. Module variables that are not published are not candidates for the anchor object in an ANCHOR clause. The name of the user-defined type must not begin with "SYS_" (SQLSTATE 42939) and must be unique within the module. A structured type cannot be defined in a module. Any generated functions required to support the type definition are also defined in the module as published functions.

module-variable-definition
The syntax to define the variable is the same as the CREATE VARIABLE statement excluding the CREATE keyword and the variable-name must be specified without any qualification (SQLSTATE 42601). Module user-defined data types that are not published are not candidates for the any data type referenced in the variable definition. Module variables that are not published are not candidates for the anchor object in an ANCHOR clause. The name of the variable must not begin with "SYS_" (SQLSTATE 42939) and must be unique within the module.

module-object-identification
Identifies a unique module object.

module-function-designator
Uniquely identifies a single module function.

FUNCTION ***unqualified-function-name***
Identifies a particular function, and is valid only if there is exactly one function instance with the name unqualified-function-name in the module. The identified function can have any number of parameters defined for it. If no function by this name exists in the module, an error (SQLSTATE 42704) is raised. If there is more than one instance of the function in the module, an error (SQLSTATE 42725) is raised.

FUNCTION ***unqualified-function-name (data type,...)***
Provides the function signature, which uniquely identifies the function. The function resolution algorithm is not used.

unqualified-function-name
Specifies the name of the function.

(data-type,...)
Values must match the data types that were specified (in the corresponding position) when the function was originally defined. The number of data types, and the logical concatenation of the data types, is used to identify the specific function instance.

If a data type is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision, or scale for the parameterized data types. Instead, an empty set of parentheses can be coded to indicate that these attributes are to be ignored when looking for a data type match. FLOAT() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (REAL or DOUBLE). If length, precision, or scale is coded, the value must exactly match that specified when the function was defined.

A type of FLOAT(n) does not need to match the defined value for n, because 0 < n < 25 means REAL, and 24 < n < 54 means DOUBLE. Matching occurs on the basis of whether the type

is REAL or DOUBLE. If no function with the specified signature exists in the module, an error (SQLSTATE 42883) is raised.

SPECIFIC FUNCTION ***unqualified-specific-name***
Identifies a particular user-defined function, using the name that is specified or defaulted to at function definition time. The unqualified-specific-name must identify a specific function instance in the module; otherwise, an error is returned (SQLSTATE 42704).

module-procedure-designator
Uniquely identifies a single module procedure.

PROCEDURE ***unqualified-procedure-name***
Identifies a particular procedure, and is valid only if there is exactly one procedure instance with the name unqualified-procedure-name in the module. The identified procedure can have any number of parameters defined for it. If no procedure by this name exists in the module, an error is returned (SQLSTATE 42704). If there is more than one instance of the procedure in the module, an error is returned (SQLSTATE 42725).

PROCEDURE ***unqualified-procedure-name (data-type,...)***
Provides the procedure signature, which uniquely identifies the procedure. The procedure resolution algorithm is not used.

unqualified-procedure-name
Specifies the name of the procedure.

(data-type,...)
Values must match the data types that were specified (in the corresponding position) when the procedure was originally defined. The number of data types, and the logical concatenation of the data types, is used to identify the specific procedure instance.

If a data type is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision, or scale for the parameterized data types. Instead, an empty set of parentheses can be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (REAL or DOUBLE). If length, precision, or scale is coded, the value must exactly match that specified in when the procedure was defined.

A type of FLOAT(n) does not need to match the defined value for n, because 0 < n < 25 means REAL, and 24 < n < 54 means DOUBLE. Matching occurs on the basis of whether the type is REAL or DOUBLE.

If no procedure with the specified signature exists in the module, an error is returned (SQLSTATE 42883).

SPECIFIC PROCEDURE ***unqualified-specific-name***
Identifies a particular procedure, using the name that is specified or defaulted to at procedure definition time. The unqualified-specific-name must identify a specific procedure instance in the module; otherwise, an error is returned (SQLSTATE 42704).

TYPE ***type-name***
Identifies a user-defined type from the module. The type-name must be specified without any qualification (SQLSTATE 42601) and must identify a user-defined type that exists in the module (SQLSTATE 42704).

VARIABLE ***variable-name***
Identifies a global variable from the module. The variable-name must be specified without any qualification (SQLSTATE 42601) and must identify a global variable that exists in the module (SQLSTATE 42704).

CONDITION ***condition-name***
Identifies a condition from the module. The condition-name must be specified without any qualification and must identify a condition that exists in the module (SQLSTATE 42737).

Rules

- Names of objects in the module cannot begin with "SYS_" with the exception of specifically designated SYS_INIT procedure name (SQLSTATE 42939).
- **ALTER MODULE DROP FUNCTION**: If the function is referenced in the definition of a row permission or column mask, the function cannot be dropped (SQLSTATE 42893).
- **ALTER MODULE DROP VARIABLE**: If the variable is referenced in the definition of a row permission or column mask, the variable cannot be dropped (SQLSTATE 42893).
- **ALTER MODULE DROP BODY**: If the module is referenced in the definition of a row permission or column mask, the module cannot be dropped (SQLSTATE 42893).

Notes

- ***Module initialization*:** A module can have a special initialization procedure called SYS_INIT that is implicitly executed when the first reference is made to a module routine or module global variable. The SYS_INIT procedure must be implemented with no parameters, cannot return result sets, and can be an SQL or external procedure that cannot be published (SQLSTATE 428HP). If the SYS_INIT procedure fails, an error is returned for the statement that caused the module initialization (SQLSTATE 56098).
- ***Use of module conditions*:** A module condition can only be used with a SIGNAL statement, RESIGNAL statement or a handler declaration that is within a compound SQL (compiled) statement.
- ***Invalidation*:** If a routine prototype is replaced using the ADD action, all objects that depended on the published module routine are invalidated. If DROP BODY is issued, all objects dependent on published module routines are invalidated.
- ***Obfuscation*:** The ALTER MODULE ADD FUNCTION, ALTER MODULE ADD PROCEDURE, ALTER MODULE PUBLISH FUNCTION, and ALTER MODULE PUBLISH PROCEDURE statements can be submitted in obfuscated form. In an obfuscated statement, only the routine name and its parameters are readable. The rest of the statement is encoded in such a way that is not readable but can be decoded by the database server. Obfuscated statements can be produced by calling the DBMS_DDL.WRAP function.

Example

The following statements create a module named INVENTORY containing an associative array type, a variable of that data type, a procedure that adds elements to the array and a function that extracts elements from the array. Only the function and the procedure can be referenced from outside of the module based on the PUBLISH keyword in the corresponding ALTER MODULE statements. The data type and the variable can only be referenced by other objects in the module.

```
CREATE MODULE INVENTORY

ALTER MODULE INVENTORY ADD
TYPE ITEMLIST AS INTEGER ARRAY[VARCHAR(100)]

ALTER MODULE INVENTORY ADD
VARIABLE ITEMS ITEMLIST

ALTER MODULE INVENTORY PUBLISH
PROCEDURE UPDATE_ITEM(NAME VARCHAR(100), QUANTITY INTEGER)
BEGIN
SET ITEMS[NAME] = QUANTITY;
END

ALTER MODULE INVENTORY PUBLISH
FUNCTION CHECK_ITEM(NAME VARCHAR(100)) RETURNS INTEGER
RETURN ITEMS[NAME]
```

ALTER NICKNAME

The ALTER NICKNAME statement modifies the nickname information associated with a data source object (such as a table, view, or file).

This statement modifies the information that is stored in the federated database in the following ways:

- Altering the local column names for the columns of the data source object
- Altering the local data types for the columns of the data source object
- Adding, setting, or dropping nickname and column options
- Adding or dropping a primary key
- Adding or dropping one or more unique, referential, or check constraints
- Altering one or more referential or check constraint attributes
- Altering the caching of data at a federated server

Invocation

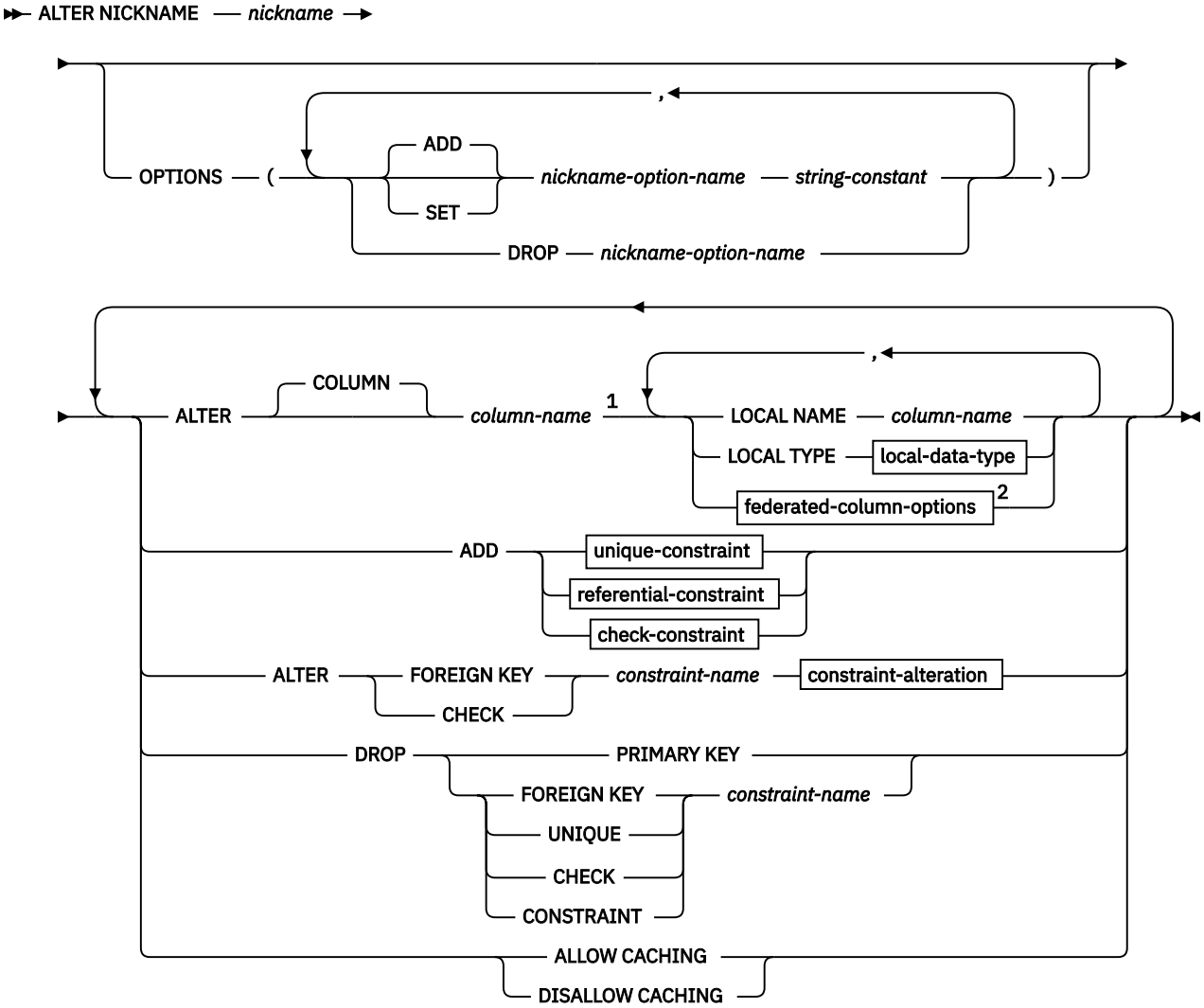
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

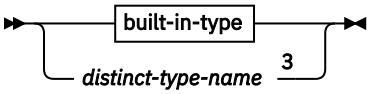
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- ALTER privilege on the nickname specified in the statement
- CONTROL privilege on the nickname specified in the statement
- ALTERIN privilege on the schema, if the schema name of the nickname exists
- SCHEMAADM authority on the schema, if the schema name of the nickname exists
- Owner of the nickname, as recorded in the OWNER column of the SYSCAT.TABLES catalog view
- DBADM authority

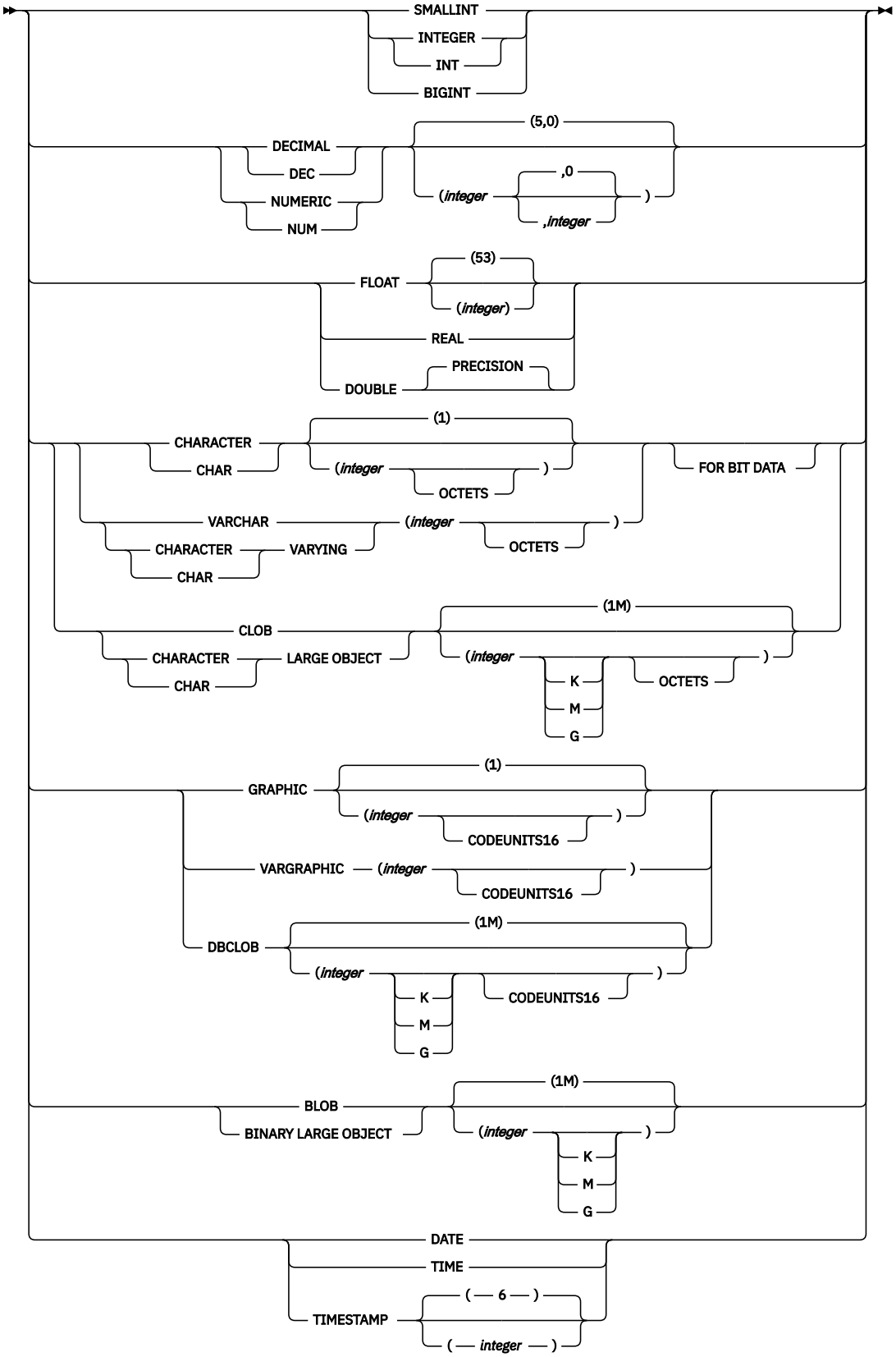
Syntax



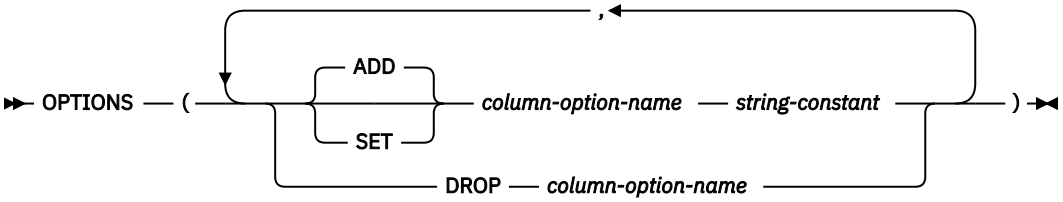
local-data-type



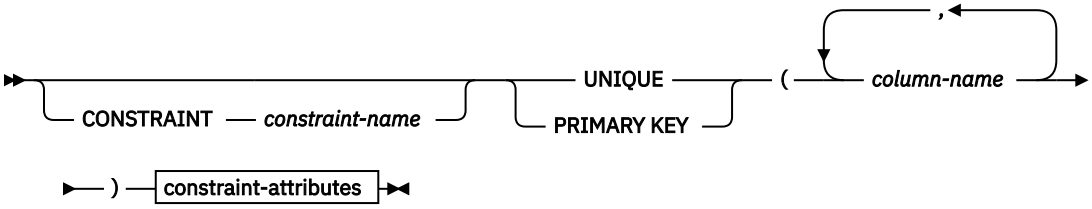
built-in-type



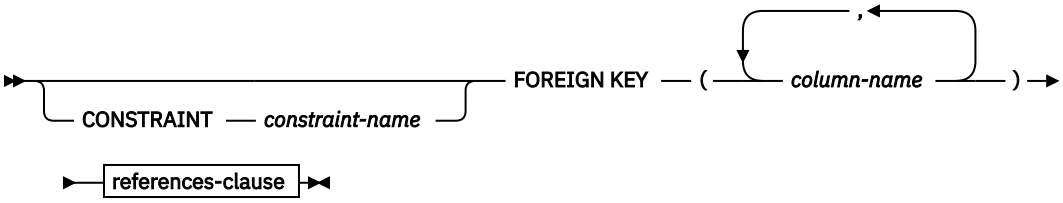
federated-column-options



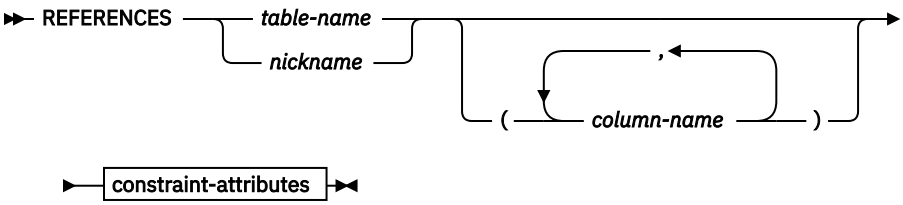
unique-constraint



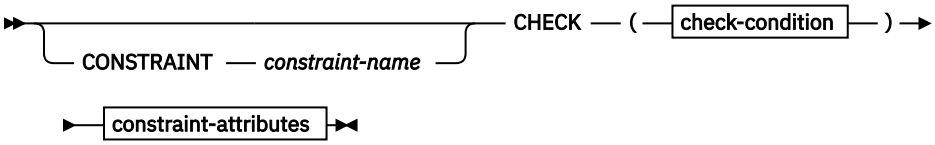
referential-constraint



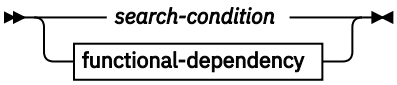
references-clause



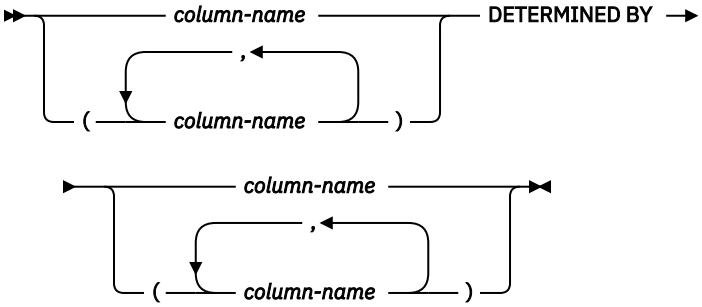
check-constraint



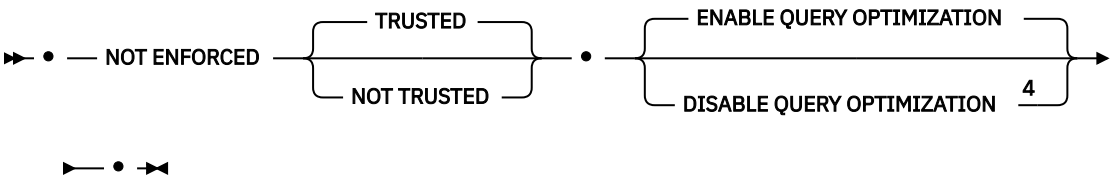
check-condition



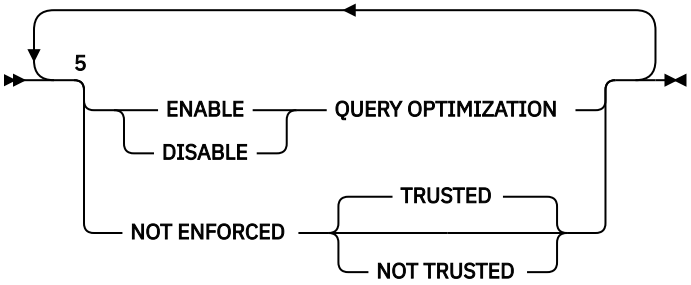
functional-dependency



constraint-attributes



constraint-alteration



Notes:

[1] You cannot specify both the ALTER COLUMN clause and an ADD, ALTER, or DROP informational constraint clause in the same ALTER NICKNAME statement.

[2] If you need to specify the federated-column-options clause in addition to the LOCAL NAME parameter, the LOCAL TYPE parameter, or both, you must specify the federated-column-options clause last.

[3] The specified distinct type cannot have any data type constraints and the source type cannot be an anchored data type (SQLSTATE 428H2).

[4] DISABLE QUERY OPTIMIZATION is not supported for a unique or primary key constraint.

[5] The same clause must not be specified more than once.

Description

nickname
Identifies the nickname for the data source object (such as a table, view, or file) that contains the column being altered. It must be a nickname described in the catalog.

OPTIONS
Indicates the nickname options that are added, set, or dropped when the nickname is altered.

ADD
Adds a nickname option.

SET
Changes the setting of a nickname option.

nickname-option-name
The nickname option that is to be added or set. Which options you can specify depends on the data source of the object for which a nickname is being created. For a list of data sources and the nickname options that apply to each, see Data source options.

string-constant
The nickname option setting as a character string constant enclosed in single quotation marks.

DROP *nickname-option-name*
Drops a nickname option.

ALTER COLUMN *column-name*
Names the column to be altered. The *column-name* is the federated server's current name for the column of the table or view at the data source. The *column-name* must identify an existing column of the nickname (SQLSTATE 42703). You cannot reference the same column name multiple times in the same ALTER NICKNAME statement (SQLSTATE 42711).

LOCAL NAME *column-name*
Specifies a new name, *column-name*, by which the federated server is to reference the column to be altered. The new name cannot be qualified, and the same name cannot be used for more than one column of the nickname (SQLSTATE 42711).

LOCAL TYPE *local-data-type*
Specifies a new local data type to which the data type of the column that is to be altered will map. The new type is denoted by *local-data-type*.

Some wrappers only support a subset of the SQL data types. For descriptions of specific data types, see the description of the "CREATE TABLE" statement.

built-in-type
See "CREATE TABLE" for the description of built-in data types.

OPTIONS
Indicates what column options are to be added, set, or dropped for the column specified after the COLUMN keyword.

ADD
Adds a column option.

SET
Changes the setting of a column option.

column-option-name
Names a column option that is to be added or set.

string-constant
Specifies the setting for *column-option-name* as a character string constant.

DROP *column-option-name*
Drops a column option.

ADD *unique-constraint*
Defines a unique constraint. See the description of the "CREATE NICKNAME" statement.

ADD *referential-constraint*
Defines a referential constraint. See the description of the "CREATE NICKNAME" statement.

ADD *check-constraint*
Defines a check constraint. See the description of the "CREATE NICKNAME" statement.

ALTER FOREIGN KEY *constraint-name*
Alters the constraint attributes of the referential constraint *constraint-name*. For a description of the constraint attributes, see the "CREATE NICKNAME" statement. The *constraint-name* must identify an existing referential constraint (SQLSTATE 42704).

ALTER CHECK *constraint-name*
Alters the constraint attributes of the check constraint *constraint-name*. The *constraint-name* must identify an existing check constraint (SQLSTATE 42704).

constraint-alteration
Provides options for changing the attributes associated with referential or check constraints.

ENABLE QUERY OPTIMIZATION
The constraint can be used for query optimization under appropriate circumstances.

DISABLE QUERY OPTIMIZATION
The constraint cannot be used for query optimization.

NOT ENFORCED
Specifies that the constraint is not enforced by the database manager during normal operations such as insert, update, or delete.

TRUSTED
The data can be trusted to conform to the constraint. TRUSTED must be used only if the data in the table is independently known to conform to the constraint. Query results might be unpredictable if the data does not actually conform to the constraint. This is the default option.

NOT TRUSTED
The data cannot be trusted to conform to the constraint. NOT TRUSTED is intended for cases where the data conforms to the constraint for most rows, but it is not independently known that all the rows or future additions will conform to the constraint. If a constraint is NOT TRUSTED and enabled for query optimization, then it will not be used to perform optimizations that depend on the data conforming completely to the constraint. NOT TRUSTED can be specified only for referential integrity constraints (SQLSTATE 42613).

DROP PRIMARY KEY
Drops the definition of the primary key and all referential constraints that are dependent upon this primary key. The nickname must have a primary key.

DROP FOREIGN KEY *constraint-name*
Drops the referential constraint *constraint-name*. The *constraint-name* must identify an existing referential constraint defined on the nickname.

DROP UNIQUE *constraint-name*
Drops the definition of the unique constraint *constraint-name* and all referential constraints that are dependent upon this unique constraint. The *constraint-name* must identify an existing unique constraint.

DROP CHECK *constraint-name*
Drops the check constraint *constraint-name*. The *constraint-name* must identify an existing check constraint defined on the nickname.

DROP CONSTRAINT *constraint-name*
Drops the constraint *constraint-name*. The *constraint-name* must identify an existing check constraint, referential constraint, primary key, or unique constraint defined on the nickname.

ALLOW CACHING or DISALLOW CACHING
Specifies whether the nickname can be referenced in a query that defines a materialized query table, which could be used to cache data from the data source at the federated server.

ALLOW CACHING
Specifies that the nickname can be referenced in a query that defines a materialized query table, which allows data from the data source to be cached in the materialized query table at the federated server. The refreshable options defined for the materialized query table specify how the cached data in the materialized query table is maintained.

DISALLOW CACHING
Specifies that the nickname cannot be referenced in a query that defines a materialized query table. DISALLOW CACHING cannot be specified for a nickname that is referenced in the fullselect of a materialized query table definition (SQLSTATE 42917).

Rules

- If a nickname is used in a view, SQL method, or SQL function, or informational constraints are defined on it, the ALTER NICKNAME statement cannot be used to change the local names or data types for the columns in the nickname (SQLSTATE 42893). The statement can be used, however, to add, set, or drop column options, nickname options, or informational constraints.
- If a nickname is referenced by a materialized query table definition, the ALTER NICKNAME statement cannot be used to change the local names, data types, column options, or nickname options (SQLSTATE 42893). Moreover, the statement cannot be used to disable caching (SQLSTATE 42917). The statement can be used, however, to add, alter, or drop informational constraints.
- A column option cannot be specified more than once in the same ALTER NICKNAME statement (SQLSTATE 42853). When a column option is enabled, reset, or dropped, any other column options that are in use are not affected.
- For relational nicknames, the ALTER NICKNAME statement within a given unit of work (UOW) cannot be processed under either of the following conditions (SQLSTATE 55007):
  - A nickname referenced in this statement has a cursor open on it in the same UOW

  - Either an INSERT, DELETE, or UPDATE statement is already issued in the same UOW against the nickname that is referenced in this statement
- For non-relational nicknames, the ALTER NICKNAME statement within a given unit of work (UOW) cannot be processed under any of the following conditions (SQLSTATE 55007):
  - A nickname referenced in this statement has a cursor open on it in the same UOW
  - A nickname referenced in this statement is already referenced by a SELECT statement in the same UOW
  - Either an INSERT, DELETE, or UPDATE statement has already been issued in the same UOW against the nickname that is referenced in this statement

Notes

- If the ALTER NICKNAME statement is used to change the local name for a column of a nickname, queries against that column must reference it by its new name.
- When the local specification of a column's data type is changed, the database manager invalidates any statistics (HIGH2KEY, LOW2KEY, and so on) gathered for that column.
- ***Caching and protected objects:*** For nicknames whose data source object is protected, specify `DISALLOW CACHING`. This ensures that each time the nickname is used, data for the appropriate authorization ID is returned from the data source at query execution time. This is done by restricting the nickname from being used in the definition of a materialized query table at the federated server, which might be being used to cache the nickname data.
- BINARY and VARBINARY types are not supported in a Federated system.

Examples

1. The nickname NICK1 references a Db2 for IBM i table called T1. Also, COL1 is the local name that references this table's first column, C1. Rename the local name for C1 from COL1 to NEWCOL.

```
ALTER NICKNAME NICK1
  ALTER COLUMN COL1
  LOCAL NAME NEWCOL
```

2. The nickname EMPLOYEE references a Db2 for z/OS table called EMP. Also, SALARY is the local name that references EMP_SAL, one of this table's columns. The column's data type, FLOAT, maps to the local data type, DOUBLE. Change the mapping so that FLOAT maps to DECIMAL (10, 5).

```
ALTER NICKNAME EMPLOYEE
  ALTER COLUMN SALARY
  LOCAL TYPE DECIMAL(10,5)
```

3. Indicate that in an Oracle table, a column with the data type of VARCHAR does not have trailing blanks. The nickname for the table is NICK2, and the local name for the column is COL1.

```
ALTER NICKNAME NICK2
  ALTER COLUMN COL1
  OPTIONS (ADD VARCHAR_NO_TRAILING_BLANKS 'Y')
```

4. Alter the fully qualified path for the table-structured file, drugdata1.txt, for the nickname DRUGDATA1. Use the FILE_PATH nickname option and change the path from the current value of '/user/pat/drugdata1.txt' to '/usr/kelly/data/drugdata1.txt'.

```
ALTER NICKNAME DRUGDATA1
  OPTIONS (SET FILE_PATH '/usr/kelly/data/drugdata1.txt')
```

ALTER PACKAGE

The ALTER PACKAGE statement alters bind options for a package at the current server without having to bind or rebind the package.

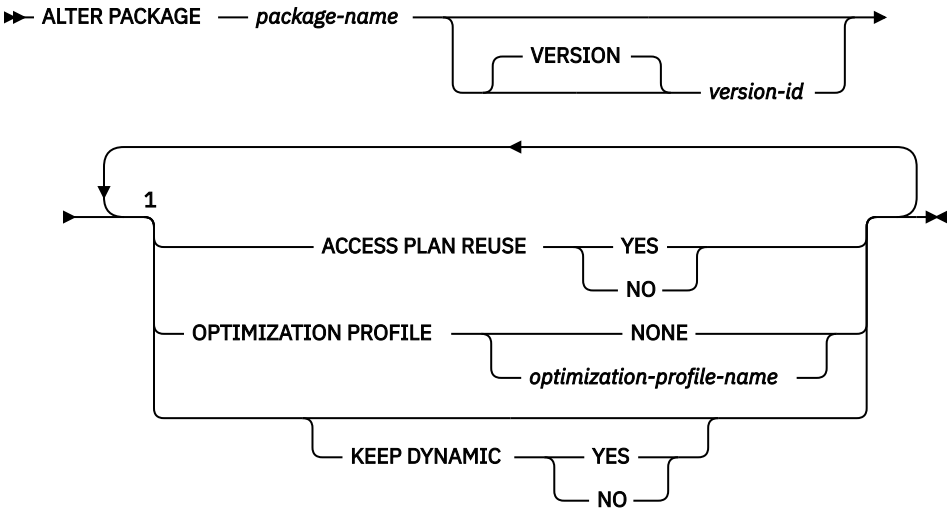
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- ALTERIN privilege on the schema
- BIND privilege on the package
- SCHEMAADM authority on the schema
- DBADM authority



Notes:

[1] The same clause must not be specified more than once.

Description

package-name

Identifies the package that is to be altered. The package name must identify a package that exists at the current server (SQLSTATE 42704).

VERSION *version-id*

Identifies which package version is to be altered. If a value is not specified, the version defaults to the empty string. If multiple packages with the same package name but different versions exist, only one package version can be altered in one invocation of the ALTER PACKAGE statement. Delimit the version identifier with double quotation marks when it:

- Is generated by the VERSION(AUTO) precompiler option
- Begins with a digit
- Contains lowercase or mixed-case letters

If the statement is invoked from an operating system command prompt, precede each double quotation mark delimiter with a back slash character to ensure that the operating system does not strip the delimiters.

ACCESS PLAN REUSE
Indicates whether the query compiler should attempt to reuse the access plans for static statements in the package during future implicit and explicit rebinds.

NO
Specifies not to reuse access plans.

YES
Specifies to attempt to reuse access plans.

OPTIMIZATION PROFILE
Indicates what, if any, optimization profile to associate with the package.

NONE
Associates no optimization profile with the package. If an optimization profile is already associated with the package, the association is removed.

optimization-profile-name
Associates the optimization profile *optimization-profile-name* with the package. The optimization profile is a two-part name. If the specified *optimization-profile-name* is unqualified, the value of the CURRENT DEFAULT SCHEMA special register is used as the implicit qualifier. If an optimization profile is already associated with the package, the association is replaced with *optimization-profile-name*.

While the ALTER PACKAGE statement removes the current copy of the package from the Db2 package cache, it does not invalidate the package and does not cause an implicit rebind to take place. This means that although dynamic SQL is affected by the changes made by the statement, query execution plans for static statements are not be affected until the next implicit or explicit rebind.

KEEP DYNAMIC

Starting with Db2 Version 9.8 Fix Pack 2, you can modify the value of the KEEPDYNAMIC bind option for a package without requiring a fresh bind operation, thereby avoiding unnecessary recompilation until the next bind operation occurs. This option controls how long the statement text and section associated with a prepared statement are kept in the SQL context. It takes effect after all applications that are using the package have completed the transactions that were running when the **ALTER PACKAGE** statement was executed.

YES

Instructs the SQL context to keep the statement text and section associated with prepared statements indefinitely. Dynamic SQL statements are kept across transactions. All packages bound with KEEPDYNAMIC YES are by default compatible with the existing package cache behavior.

NO

Instructs the SQL context to remove the statement text and section associated with prepared statements at the end of each unit of work. The executable versions of prepared statements and the statement text in packages bound with the KEEP DYNAMIC NO option are removed from the SQL context at transaction boundaries. The client, driver, or application needs to prepare any dynamic SQL statement it wishes to reuse in a new unit of work again.

For remote applications that use an IBM non-embedded API, once you have ensured that statements will be prepared in new transactions, you can use this option so that WLB will not be disallowed solely based on the KEEP DYNAMIC behavior. However even with this option, WLB may be disallowed for other reasons.

SELECT statements issued by cursors with the WITH HOLD option are disassociated from the SQL context at the next transaction boundary where the cursor is closed. As a result, workload balancing is allowed as long as there are no executable versions of prepared statements associated with the application in the SQL context.

Note: Workload balancing is not restricted for dynamic SQL applications that use IBM non-embedded APIs, such as JDBC, .NET, or CLI/ODBC, to run SQL within the common client packages. These interfaces implicitly re-prepare SQL statements before executing them in transactions where their connection might have been moved to a new executable version of prepared statements.

Notes

- **Catalog view values may not reflect the settings that were in effect for the package:** Because this statement does not trigger a rebind of the package, the settings for a package as shown in the SYSCAT.PACKAGES catalog view might not reflect what was actually in effect during the last BIND or REBIND. If the ALTER_TIME is greater than the LAST_BIND_TIME, then this might be the case.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with the BIND and REBIND commands. These alternatives are non-standard and should not be used.
  - APREUSE can be specified in place of ACCESS PLAN REUSE.
  - OPTPROFILE can be specified in place of OPTIMIZATION PROFILE.
  - KEEPDYNAMIC can be specified in place of KEEP DYNAMIC.

Examples

Example 1: Enable access plan reuse for package TRUUVERT.EMPADMIN.

```
ALTER PACKAGE TRUUVERT.EMPADMIN ACCESS PLAN REUSE YES
```

Example 2: Assume access plan reuse has been enabled for package TRUUVERT.EMPADMIN. Assume also that optimization profile AYYANG.INDEXHINTS contains a statement profile for a specific statement within the package. Associate the optimization profile with this package so that it will override the reuse of the access plan for the statement.

```
ALTER PACKAGE TRUUVERT.EMPADMIN OPTIMIZATION PROFILE AYYANG.INDEXHINTS
```

Dynamic statements will be affected after the statement commits; static statements will be affected at the next rebind. When the package is rebound, the query compiler will attempt to reuse the access plans for all static statements in the package, with the exception of the statement identified by the optimization profile. When recompiling this statement, the query compiler will instead attempt to apply the statement profile.

Example 3: The following statement will result in no optimization profile being associated with package TRUUVERT.EMPADMIN.

```
ALTER PACKAGE TRUUVERT.EMPADMIN OPTIMIZATION PROFILE NONE
```

ALTER PERMISSION

The ALTER PERMISSION statement alters a row permission that exists at the current server.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privileges held by the authorization ID of the statement must include SECADM authority.

Syntax

Description

permission-name
This is the name of the row permission to be altered. The name must identify a row permission that already exists at the current server (SQLSTATE 42704). The name must not identify a default row permission that is created implicitly by the database manager (SQLSTATE 428H9).

ENABLE

Enables the row permission. If row level access control is not currently activated on the table, the row permission will become effective when row level access control is activated on the table. If row level access control is currently activated on the table, the row permission becomes effective immediately and all packages and dynamic cached statements that reference the table are invalidated.

ENABLE is ignored if the row permission is already defined as enabled.

DISABLE

Disables the row permission. If row level access control is not currently activated on the table, the row permission will remain ineffective when row level access control is activated on the table. If row level access control is currently activated on the table, the row permission becomes ineffective immediately and all packages and dynamic cached statements that reference the table are invalidated.

DISABLE is ignored if the row permission is already defined as disabled.

Examples

- *Example 1*: Enable permission P1.

```
ALTER PERMISSION P1 ENABLE
```

- *Example 2*: Disable permission P1.

```
ALTER PERMISSION P1 DISABLE
```

ALTER PROCEDURE (external)

The ALTER PROCEDURE (External) statement modifies an existing external procedure by changing the properties of the procedure.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- ALTERIN privilege on the schema of the procedure
- Owner of the procedure, as recorded in the OWNER column of the SYSCAT.ROUTINES catalog view
- SCHEMAADM authority on the schema of the procedure
- DBADM authority

To alter the EXTERNAL NAME of a procedure, the privileges held by the authorization ID of the statement must also include at least one of the following authorities:

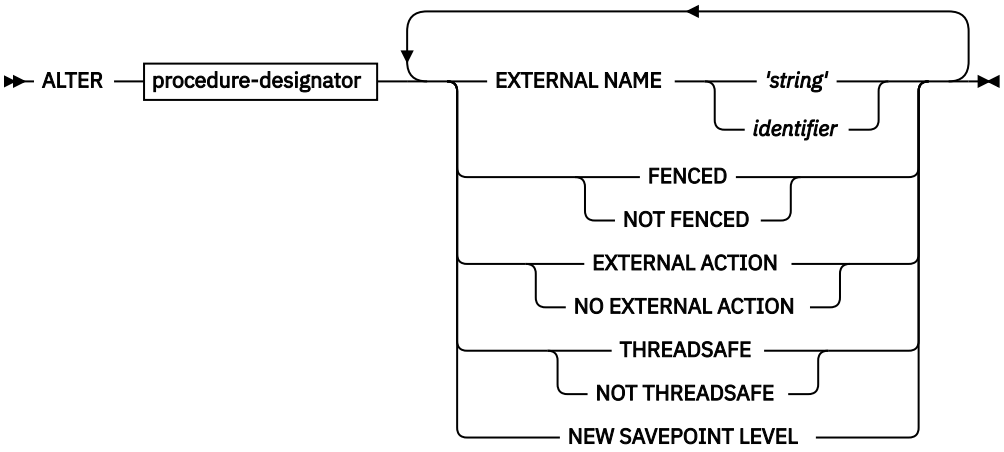
- CREATE_EXTERNAL_ROUTINE authority on the database
- DBADM authority

To alter a procedure to be not fenced, the privileges held by the authorization ID of the statement must also include at least one of the following authorities:

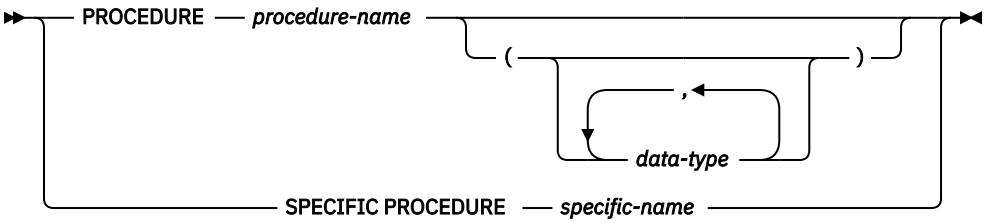
- CREATE_NOT_FENCED_ROUTINE authority on the database
- DBADM authority

To alter a procedure to be fenced, no additional authorities or privileges are required.

Syntax



procedure-designator



Description

procedure-designator
Identifies the procedure to alter. The *procedure-designator* must identify a procedure that exists at the current server. The owner of the procedure and all privileges on the procedure are preserved. For more information, see "Function, method, and procedure designators" on page 745.

EXTERNAL NAME *'string'* or *identifier*
Identifies the name of the user-written code that implements the procedure.

FENCED or NOT FENCED
Specifies whether the procedure is considered safe to run in the database manager operating environment's process or address space (NOT FENCED), or not (FENCED). Most procedures have the option of running as FENCED or NOT FENCED.

If a procedure is altered to be FENCED, the database manager insulates its internal resources (for example, data buffers) from access by the procedure. In general, a procedure running as FENCED will not perform as well as a similar one running as NOT FENCED.

CAUTION: Use of NOT FENCED for procedures that were not adequately coded, reviewed, and tested can compromise the integrity of a Db2 database. Db2 databases take some

precautions against many of the common types of inadvertent failures that might occur, but cannot guarantee complete integrity when NOT FENCED stored procedures are used.

A procedure declared as NOT THREADSAFE cannot be altered to be NOT FENCED (SQLSTATE 42613).

If a procedure has any parameters defined AS LOCATOR, and was defined with the NO SQL option, the procedure cannot be altered to be FENCED (SQLSTATE 42613).

This option cannot be altered for LANGUAGE OLE or CLR procedures (SQLSTATE 42849).

This option cannot be altered for a procedure that is registered as a component routine of an aggregate interface function (SQLSTATE 42849).

EXTERNAL ACTION or NO EXTERNAL ACTION
Specifies whether the procedure takes some action that changes the state of an object not managed by the database manager (EXTERNAL ACTION), or not (NO EXTERNAL ACTION). If NO EXTERNAL ACTION is specified, the system can use certain optimizations that assume the procedure has no external impact.

This option cannot be altered for a procedure that is registered as a component routine of an aggregate interface function (SQLSTATE 42849).

THREADSAFE or NOT THREADSAFE
Specifies whether the procedure is considered safe to run in the same process as other routines (THREADSAFE), or not (NOT THREADSAFE).

If the procedure is defined with LANGUAGE other than OLE:

- If the procedure is defined as THREADSAFE, the database manager can invoke the procedure in the same process as other routines. In general, to be threadsafe, a procedure should not use any global or static data areas. Most programming references include a discussion of writing threadsafe routines. Both FENCED and NOT FENCED procedures can be THREADSAFE.
- If the procedure is defined as NOT THREADSAFE, the database manager will never invoke the procedure in the same process as another routine. Only a fenced procedure can be NOT THREADSAFE (SQLSTATE 42613).

This option cannot be altered for LANGUAGE OLE procedures (SQLSTATE 42849).

This option cannot be altered for a procedure that is registered as a component routine of an aggregate interface function (SQLSTATE 42849).

NEW SAVEPOINT LEVEL
Specifies that a new savepoint level is to be created for the procedure. A savepoint level refers to the scope of reference for any savepoint-related statement, as well as to the name space used for comparison and reference of any savepoint names.

The savepoint level for a procedure can only be altered to NEW SAVEPOINT LEVEL.

This option cannot be altered for a procedure that is registered as a component routine of an aggregate interface function (SQLSTATE 42849).

Rules

- It is not possible to alter a procedure that is in the following schema (SQLSTATE 42832):
  - SYSIBM
  - SYSFUN
  - SYSPROC

Example

Alter the procedure PARTS_ON_HAND() to be not fenced.

```
ALTER PROCEDURE PARTS_ON_HAND() NOT FENCED
```

ALTER PROCEDURE (sourced)

The ALTER PROCEDURE (Sourced) statement modifies an existing sourced procedure by changing the data type of one or more parameters of the sourced procedure.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

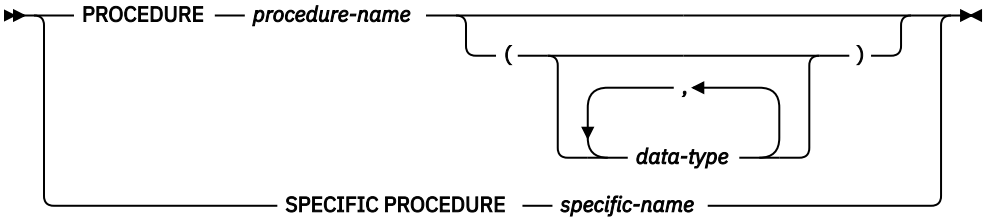
Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- ALTERIN privilege on the schema of the procedure
- Owner of the procedure, as recorded in the OWNER column of the SYSCAT.ROUTINES catalog view
- SCHEMAADM authority on the schema of the procedure
- DBADM authority

Syntax

procedure-designator



parameter-alteration

Description

procedure-designator
Uniquely identifies the procedure to be altered. The identified procedure must be a sourced procedure (SQLSTATE 42849). For more information, see "Function, method, and procedure designators" on page 745.

parameter-name
Identifies the parameter to be altered. The *parameter-name* must identify an existing parameter of the procedure (SQLSTATE 42703). The name must not identify a parameter that is otherwise being altered in the same ALTER PROCEDURE statement (SQLSTATE 42713).

data-type
Specifies the new local data type of the parameter. SQL data type specifications and abbreviations that are valid for the *data-type* definition of a CREATE TABLE statement can be specified. BLOB, CLOB, DBCLOB, DECFLOAT, XML, REFERENCE, and user-defined types are not supported (SQLSTATE 42815).

Example

Assume that federated procedure FEDEMPLOYEE has been created for a remote Oracle procedure named 'EMPLOYEE'. The data type of an input parameter named SALARY maps to a DOUBLE(8) in Db2. Alter the data type of this parameter to DECIMAL(5,2).

```
ALTER PROCEDURE FEDEMPLOYEE
  ALTER PARAMETER SALARY
  SET DATA TYPE DECIMAL(5,2)
```

ALTER PROCEDURE (SQL)

The ALTER PROCEDURE (SQL) statement modifies an existing SQL procedure by changing the properties of the procedure.

Invocation

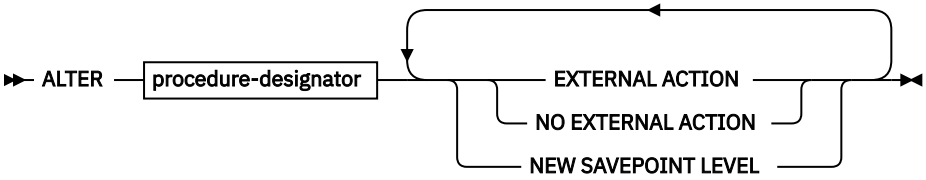
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

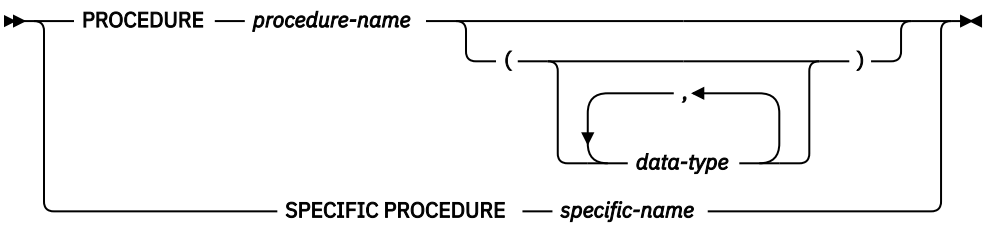
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- ALTERIN privilege on the schema of the procedure
- Owner of the procedure, as recorded in the OWNER column of the SYSCAT.ROUTINES catalog view
- SCHEMAADM authority on the schema of the procedure
- DBADM authority

Syntax



procedure-designator



Description

procedure-designator

Identifies the procedure to alter. The *procedure-designator* must identify a procedure that exists at the current server. The owner of the procedure and all privileges on the procedure are preserved. For more information, see "Function, method, and procedure designators" on page 745.

EXTERNAL ACTION or NO EXTERNAL ACTION

Specifies whether the procedure takes some action that changes the state of an object not managed by the database manager (EXTERNAL ACTION), or not (NO EXTERNAL ACTION). If NO EXTERNAL

ACTION is specified, the system can use certain optimizations that assume the procedure has no external impact.

NEW SAVEPOINT LEVEL
Specifies that a new savepoint level is to be created for the procedure. A savepoint level refers to the scope of reference for any savepoint-related statement, as well as to the name space used for comparison and reference of any savepoint names.

The savepoint level for a procedure can only be altered to NEW SAVEPOINT LEVEL.

Rules

- It is not possible to alter a procedure that is in the following schema (SQLSTATE 42832):
  - SYSIBM
  - SYSFUN
  - SYSPROC

Example

Alter the procedure MEDIAN_RESULT_SET to indicate that it has no external action.

```
ALTER PROCEDURE MEDIAN_RESULT_SET(DOUBLE)
  NO EXTERNAL ACTION
```

ALTER SCHEMA

The ALTER SCHEMA statement modifies the data capture attribute of an existing schema.

Invocation

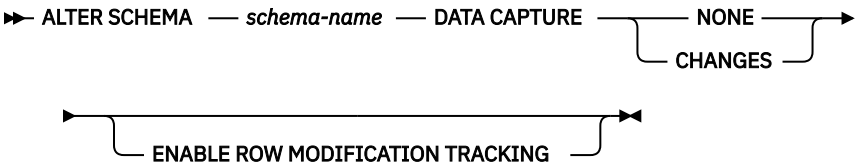
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- Owner of the schema, as recorded in the OWNER column of SYSCAT.SCHEMATA catalog view
- DBADM authority

Syntax



Description

schema-name
Identifies the schema to be altered. The *schema-name* must identify a schema that exists at the current server (SQLSTATE 42704).

DATA CAPTURE
Indicates whether extra information for data replication is to be written to the log.

NONE
Indicates that no extra information for data replication will be logged.

CHANGES
Indicates that extra information regarding SQL changes to this schema will be written to the log. This option is required if this schema will be replicated and a replication capture program is used to capture changes for this schema from the log.

ENABLE ROW MODIFICATION TRACKING
Indicates tables created in the schema are to be enabled for logical backup. This only applies to columnar organized tables. For a list of restrictions, see Schema enabled for row modification tracking.

Notes

- Altering the DATA CAPTURE attribute at the schema level causes newly created tables to inherit the DATA CAPTURE attribute from the schema if one is not specified at the table level. Altering the DATA CAPTURE attribute at the schema level does not affect the DATA CAPTURE attribute of existing tables within that schema. If the DATA CAPTURE attribute is changed and any existing tables do not match the new attribute, a warning is returned (SQLSTATE 01696).
- To find the list of tables that have the DATA CAPTURE attribute set to CHANGES, issue the following query:

```
SELECT TABNAME, TABSCHEMA FROM SYSCAT.TABLES
  WHERE TYPE IN ('T','S','L')
    AND DATACAPTURE <> 'N'
```

- To find the list of tables that have the DATA CAPTURE attribute set to NONE, issue the following query:

```
SELECT TABNAME, TABSCHEMA FROM SYSCAT.TABLES
  WHERE TYPE IN ('T','S','L')
    AND DATACAPTURE = 'N'
```

Related information
Schema enabled for row modification tracking

ALTER SECURITY LABEL COMPONENT

The ALTER SECURITY LABEL COMPONENT statement modifies a security label component.

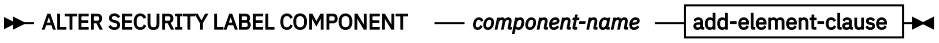
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

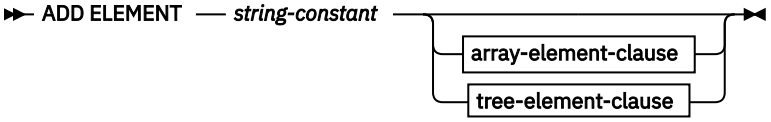
Authorization

The privileges held by the authorization ID of the statement must include SECADM authority.

Syntax

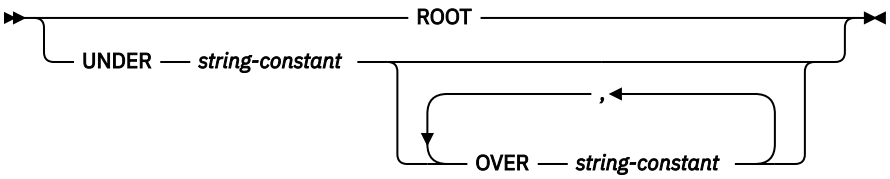


add-element-clause



array-element-clause

tree-element-clause



Description

component-name
Specifies the name of the security label component to be altered. The named component must exist at the current server (SQLSTATE 42704).

ADD ELEMENT
Specifies the element to be added to the security label component. If *array-element-clause* and *tree-element-clause* are not specified, the element is added to a set component.

string-constant
The string constant value to be added to the set of valid values for the security label component. The value cannot be the same as any other value in the set of valid values for the security label component (SQLSTATE 42713).

BEFORE or AFTER
For an array component, specifies where the element is to be added in the ordered set of element values for the security label component.

BEFORE
The element to be added is to be ranked immediately before the identified existing element.

AFTER
The element to be added is to be ranked immediately after the identified existing element.

string-constant
Specifies a string constant value of an existing element in the array component (SQLSTATE 42704).

ROOT or UNDER
For a tree component, specifies where the element is to be added in the tree structure of node element values for the security label component.

ROOT
The element to be added is to be considered the root node of the tree.

UNDER *string-constant*
The element to be added is an immediate child of the element identified by the *string-constant*. The *string-constant* value must be an existing element in the tree component (SQLSTATE 42704).

OVER *string-constant*,...
The element to be added is an immediate child of every element identified by the list of *string-constant* values. Each *string-constant* value must be an existing element in the tree component (SQLSTATE 42704).

Rules

- Element names cannot contain any of these characters (SQLSTATE 42601):
  - Opening parenthesis - (
  - Closing parenthesis -)
  - Comma - ,

  - Colon - :
- An element name can have no more than 32 bytes (SQLSTATE 42622).
- If a security label component is a set or a tree, no more than 64 elements can be part of that component.
- If the component is an array, it might or might not be possible to arrive at an array whose total number of elements matches the total number of elements that could be specified when creating a security label component of type array (65 535). The database manager assigns an encoded value to the new element from within the interval into which the new element is added. Depending on the pattern followed when adding elements to an array component, the number of possible values that can be assigned from within a particular interval might be quickly exhausted if several elements are inserted into that interval.
- BEFORE and AFTER must only be specified for a security label component that is an array (SQLSTATE 42613).
- ROOT and UNDER must only be specified for a security label component that is a tree (SQLSTATE 42613).

Notes

- For a set component, there is no order to the elements in the set.

Examples

- *Example 1:* Add the element 'High classified' to the LEVEL security label array component between the elements 'Secret' and 'Classified'.

```
ALTER SECURITY LABEL COMPONENT LEVEL
  ADD ELEMENT 'High classified' BEFORE 'Classified'
```

- *Example 2:* Add the element 'Funding' to the COMPARTMENTS security label set component.

```
ALTER SECURITY LABEL COMPONENT COMPARTMENTS
  ADD ELEMENT 'Funding'
```

- *Example 3:* Add the elements 'ENGINE' and 'TOOLS' to the GROUPS security label array component. The following diagram shows where these new elements are to be placed.

```
                    PROJECT
                 ___|_____
                |         |
           ENGINE         TOOLS
        ___|_____
       |        |
    TEST      DEVELOPMENT
           ____|____
          |         |
       CURRENT    FIELD


ALTER SECURITY LABEL COMPONENT GROUPS
  ADD ELEMENT 'TOOLS' UNDER 'PROJECT'

ALTER SECURITY LABEL COMPONENT GROUPS
  ADD ELEMENT 'ENGINE' UNDER 'PROJECT'
    OVER 'TEST', 'DEVELOPMENT'
```

ALTER SECURITY POLICY

The ALTER SECURITY POLICY statement modifies a security policy.

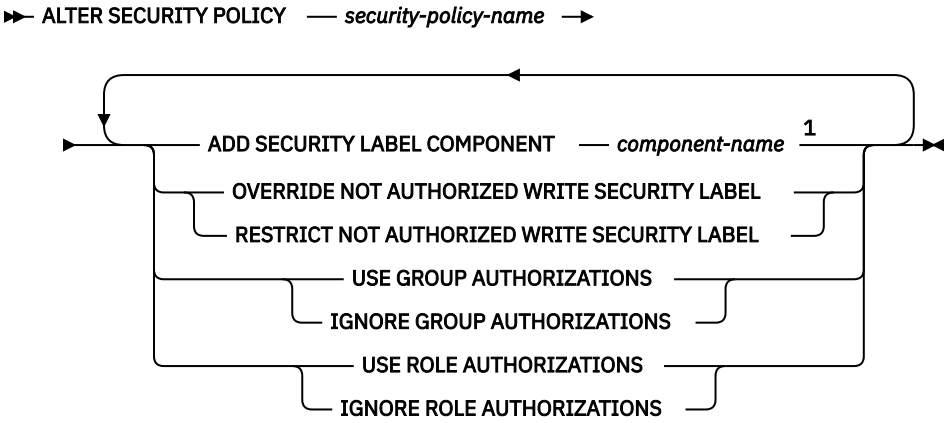
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include SECADM authority.

Syntax



Notes:

[1] Only the ADD SECURITY LABEL COMPONENT clause can be specified more than once.

Description

security-policy-name
Specifies the name of the security policy to be altered. The name must identify an existing security policy at the current server (SQLSTATE 42710).

ADD SECURITY LABEL COMPONENT *component-name*
Adds a security label component to the security policy. The same security component must not be specified more than once for the security policy (SQLSTATE 42713). The security policy cannot currently be in use by a table (SQLSTATE 42893).

OVERRIDE NOT AUTHORIZED WRITE SECURITY LABEL or RESTRICT NOT AUTHORIZED WRITE SECURITY LABEL
Specifies the action taken when a user is not authorized to write the explicitly specified security label that is provided in the INSERT or UPDATE statement issued against a table that is protected with this security policy. A user's security label and exemption credentials determine the user's authorization to write an explicitly provided security label.

OVERRIDE NOT AUTHORIZED WRITE SECURITY LABEL
Indicates that the value of the user's security label, rather than the explicitly specified security label, is used for write access during an insert or update operation.

RESTRICT NOT AUTHORIZED WRITE SECURITY LABEL
Indicates that the insert or update operation will fail if the user is not authorized to write the explicitly specified security label that is provided in the INSERT or UPDATE statement (SQLSTATE 42519).

USE GROUP AUTHORIZATION or IGNORE GROUP AUTHORIZATION
Specifies whether or not security labels and exemptions granted to groups, directly or indirectly, are considered for any access attempt.

USE GROUP AUTHORIZATION
Indicates that any security labels or exemptions granted to groups, directly or indirectly, are considered.

IGNORE GROUP AUTHORIZATION
Indicates that any security labels or exemptions granted to groups are not considered.

USE ROLE AUTHORIZATION or IGNORE ROLE AUTHORIZATION
Specifies whether or not security labels and exemptions granted to roles, directly or indirectly, are considered for any access attempt.

USE ROLE AUTHORIZATION
Indicates that any security labels or exemptions granted to roles, directly or indirectly, are considered.

IGNORE ROLE AUTHORIZATION
Indicates that any security labels or exemptions granted to roles are not considered.

Rules

- If a user does not directly hold a security label for write access, an error is returned in the following situations (SQLSTATE 42519):
  - A value for the row security label column is not explicitly provided as part of the SQL statement
  - The OVERRIDE NOT AUTHORIZED WRITE SECURITY LABEL option is in effect for the security policy, and the user is not allowed to write a data object with the provided security label

Notes

- New components are logically added at the end of the existing security label definition contained by the modified policy. Existing security labels defined for this security policy are modified to contain the new component as part of their definition with no element in their value for this component.
- ***Cache invalidation when changing NOT AUTHORIZED WRITE SECURITY LABEL:*** Changing the NOT AUTHORIZED WRITE SECURITY LABEL to a new value will cause the invalidation of any cached dynamic or static SQL statements that are dependent on any table that is protected by the security policy being altered.
- Because the session authorization ID is the focus authorization ID for label-based access control, security labels granted to groups or to roles that are accessible through groups are eligible for consideration for all types of SQL statements, including static SQL.
- If more than one security label or exemption is available to a user with associated groups or roles at the time of a read or write access attempt, those security labels and exemptions will be evaluated for eligibility based on the following rules:
  - If the security policy enables only role authorizations for consideration, all security labels and exemptions granted to roles of which the user authorization ID is a direct or indirect member will be considered. Security labels and exemptions granted to roles for which membership is only accessible through the groups associated with the user authorization ID will not be considered.
  - If the security policy enables only group authorizations for consideration, all security labels and exemptions granted to groups associated with the user authorization ID will be considered. Security labels and exemptions granted to roles for which membership is only accessible through the groups associated with the user authorization ID will not be considered.
  - If the security policy enables both group and role authorizations for consideration, any security labels and exemptions granted to roles accessible to the user indirectly through groups associated with the user authorization ID will be considered.

  - Role authorizations that are accessible to the user only through PUBLIC will not be considered at any time.
- If more than one security label is eligible for consideration during an access attempt, the values provided for each security label are merged at the individual component level to form a security label that reflects the combination of all available values at each component piece of the security policy. This is the security label value that will be used for the access attempt.

  The mechanisms for combining security labels vary by component type. The components of the resultant security label are as follows:

  - Set components contain the union of all unique values encountered in the eligible security labels
  - Array components contain the highest order element encountered in the eligible security labels
  - Tree components contain the union of all unique values encountered in the eligible security labels
- If more than one exemption is eligible for consideration during an access attempt, all found exemptions are applied to the access attempt.

Examples

- *Example 1:* Alter a security policy named DATA_ACCESS to add a new component named REGION.

```
ALTER SECURITY POLICY DATA_ACCESS
  ADD COMPONENT REGION
```

- *Example 2:* Alter a security policy named DATA_ACCESS to allow access through security labels granted to roles.

```
ALTER SECURITY POLICY DATA_ACCESS
  USE ROLE AUTHORIZATIONS
```

- *Example 3:* Show the eligible security labels that would be considered depending on the settings for group or role authorizations in a security policy. The security policy SECUR_POL has an array component and a set component, consisting of the following elements:

  Array = {TS, S, C, U}
  Set = {A, B, X, Y}

  The following security labels are defined for SECUR_POL:

  Security label L1 = C:A
  Security label L2 = S:B
  Security label L3 = TS:X
  Security label L4 = U:Y

  User Paul is a member of role R1 and group G1. Group G1 is a member of role R2. Security label L1 is granted to Paul. Security label L2 is granted to role R1. Security label L3 is granted to group G1. Security label L4 is granted to role R2. The following table shows what security labels would be considered for any access attempt by Paul, depending on the different possible settings of the security policy SECUR_POL.

Table 124. Security labels considered as a function of security policy settings

	Roles Enabled	**Roles Disabled**
Groups Enabled	L1, L2, L3, L4	L1, L3
Groups Disabled	L1, L2	L1

  The following table shows the value of the combined security label for any access attempt by Paul, depending on the different settings of the security policy SECUR_POL.

Table 125. Combined security labels as a function of security policy settings		
	Roles Enabled	**Roles Disabled**
Groups Enabled	TS:(A, B, X, Y)	TS:(A, X)
Groups Disabled	S:(A, B)	C:A

ALTER SEQUENCE

The ALTER SEQUENCE statement can be used to change a sequence.

A sequence can be changed in the following ways:

- Restarting the sequence
- Changing the increment between future sequence values
- Setting or eliminating the minimum or maximum values
- Changing the number of cached sequence numbers
- Changing the attribute that determines whether the sequence can cycle or not
- Changing whether sequence numbers must be generated in order of request

Invocation

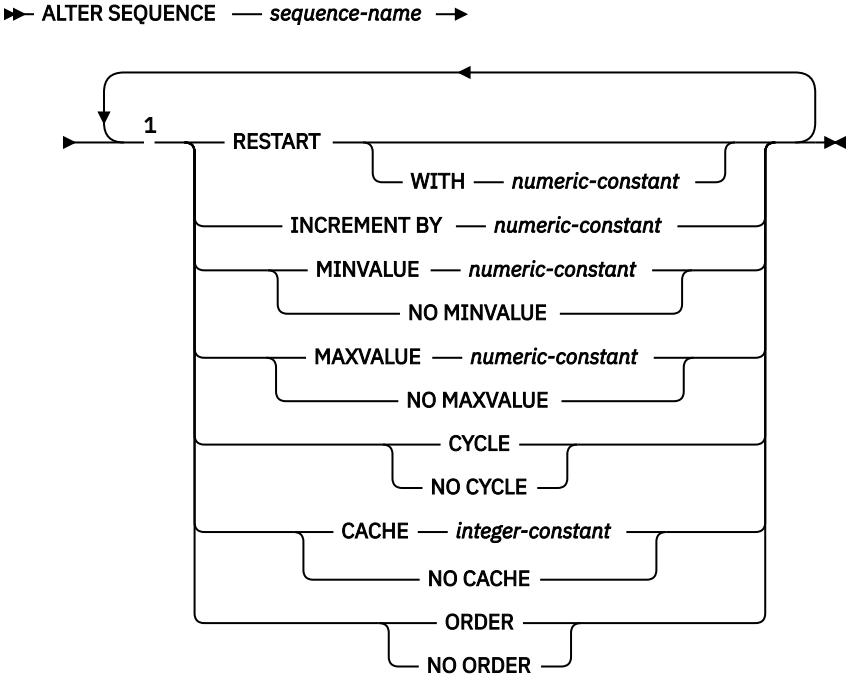
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- ALTER privilege on the sequence to be altered
- ALTERIN privilege on the schema implicitly or explicitly specified
- SCHEMAADM authority on the schema implicitly or explicitly specified
- DBADM authority

Syntax



Notes:

[1] The same clause must not be specified more than once.

Description

sequence-name
Identifies the sequence that is to be changed. The name, including the implicit or explicit schema qualifier, must uniquely identify an existing sequence at the current server. If no sequence by this name exists in the explicitly or implicitly specified schema, an error (SQLSTATE 42704) is returned. *sequence-name* must not be a sequence generated by the system for an identity column (SQLSTATE 428FB).

RESTART
Restarts the sequence. If *numeric-constant* is not specified, the sequence is restarted at the value specified implicitly or explicitly as the starting value on the CREATE SEQUENCE statement that originally created the sequence.

WITH *numeric-constant*
Restarts the sequence with the specified value. This value can be any positive or negative value that could be assigned to a column of the data type associated with the sequence (SQLSTATE 42815), without nonzero digits existing to the right of the decimal point (SQLSTATE 428FA).

INCREMENT BY *numeric-constant*
Specifies the interval between consecutive values of the sequence. This value can be any positive or negative value that could be assigned to a column of the data type associated with the sequence (SQLSTATE 42815). The value must not exceed the value of a large integer constant (SQLSTATE 42820) and must not contain nonzero digits to the right of the decimal point (SQLSTATE 428FA).

If this value is negative, then this is a descending sequence. If this value is 0 or positive, this is an ascending sequence after the ALTER statement.

MINVALUE or NO MINVALUE
Specifies the minimum value at which a descending sequence either cycles or stops generating values, or an ascending sequence cycles to after reaching the maximum value.

MINVALUE *numeric-constant*
Specifies the numeric constant that is the minimum value. This value can be any positive or negative value that could be assigned to a column of the data type associated with the sequence (SQLSTATE 42815), without nonzero digits existing to the right of the decimal point (SQLSTATE 428FA), but the value must be less than or equal to the maximum value (SQLSTATE 42815).

NO MINVALUE
For an ascending sequence, the value is the original starting value. For a descending sequence, the value is the minimum value of the data type associated with the sequence.

MAXVALUE or NO MAXVALUE
Specifies the maximum value at which an ascending sequence either cycles or stops generating values, or a descending sequence cycles to after reaching the minimum value.

MAXVALUE *numeric-constant*
Specifies the numeric constant that is the maximum value. This value can be any positive or negative value that could be assigned to a column of the data type associated with the sequence (SQLSTATE 42815), without nonzero digits existing to the right of the decimal point (SQLSTATE 428FA), but the value must be greater than or equal to the minimum value (SQLSTATE 42815).

NO MAXVALUE
For an ascending sequence, the value is the maximum value of the data type associated with the sequence. For a descending sequence, the value is the original starting value.

CYCLE or NO CYCLE
Specifies whether the sequence should continue to generate values after reaching either its maximum or minimum value. The boundary of the sequence can be reached either with the next value landing exactly on the boundary condition, or by overshooting the value.

CYCLE
Specifies that values continue to be generated for this sequence after the maximum or minimum value has been reached. If this option is used, after an ascending sequence reaches its maximum value, it generates its minimum value; or after a descending sequence reaches its minimum value, it generates its maximum value. The maximum and minimum values for the sequence determine the range that is used for cycling.

When CYCLE is in effect, then duplicate values can be generated for the sequence.

NO CYCLE
Specifies that values will not be generated for the sequence once the maximum or minimum value for the sequence has been reached.

CACHE or NO CACHE
Specifies whether to keep some preallocated values in memory for faster access. This is a performance and tuning option.

CACHE *integer-constant*
Specifies the maximum number of sequence values that are preallocated and kept in memory. Preallocating and storing values in the cache reduces synchronous I/O to the log when values are generated for the sequence.

In the event of a system failure, all cached sequence values that have not been used in committed statements are lost (that is, they will never be used).The maximum number of sequence values that can be lost is calculated as follows:

- If ORDER is specified, the maximum is the value specified for the CACHE option.
- In a multi-partition or Db2 pureScale environment, the maximum is the value specified for the CACHE option times the number of members that generate new identity values.

The minimum value is 2 (SQLSTATE 42815).

NO CACHE
Specifies that values of the sequence are not to be preallocated. It ensures that there is not a loss of values in the case of a system failure, shutdown or database deactivation. When this option is

specified, the values of the sequence are not stored in the cache. In this case, every request for a new value for the sequence results in synchronous I/O to the log.

ORDER or NO ORDER
Specifies whether the sequence numbers must be generated in order of request.

ORDER
Specifies that the sequence numbers are generated in order of request.

NO ORDER
Specifies that the sequence numbers do not need to be generated in order of request.

Notes

- Only future sequence numbers are affected by the ALTER SEQUENCE statement.
- The data type of a sequence cannot be changed. Instead, drop and re-create the sequence specifying the required data type for the new sequence.
- All cached values are lost when a sequence is altered.
- After restarting a sequence or changing to CYCLE, it is possible for sequence numbers to be duplicate values of ones generated by the sequence previously.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - A comma can be used to separate multiple sequence options.
  - NOMINVALUE, NOMAXVALUE, NOCYCLE, NOCACHE, and NOORDER can be specified in place of NO MINVALUE, NO MAXVALUE, NO CYCLE, NO CACHE, and NO ORDER, respectively

Example

A possible reason for specifying RESTART without a numeric value would be to reset the sequence to the START WITH value. In this example, the goal is to generate the numbers from 1 up to the number of rows in the table and then inserting the numbers into a column added to the table using temporary tables. Another use would be to get results back where all the resulting rows are numbered:

```
ALTER SEQUENCE ORG_SEQ RESTART
SELECT NEXT VALUE FOR ORG_SEQ, ORG.* FROM ORG
```

ALTER SERVER

The ALTER SERVER statement is used to modify the definition or configuration of a data source.

This statement can be used to make the following changes:

- Modify the definition of a specific data source, or the definition of a category of data sources.
- Make changes in the configuration of a specific data source, or the configuration of a category of data sources-changes that will persist over multiple connections to the federated database.

In this statement, the word SERVER and the parameter names that start with *server-* refer only to data sources in a federated system. They do not refer to the federated server in such a system, or to DRDA application servers.

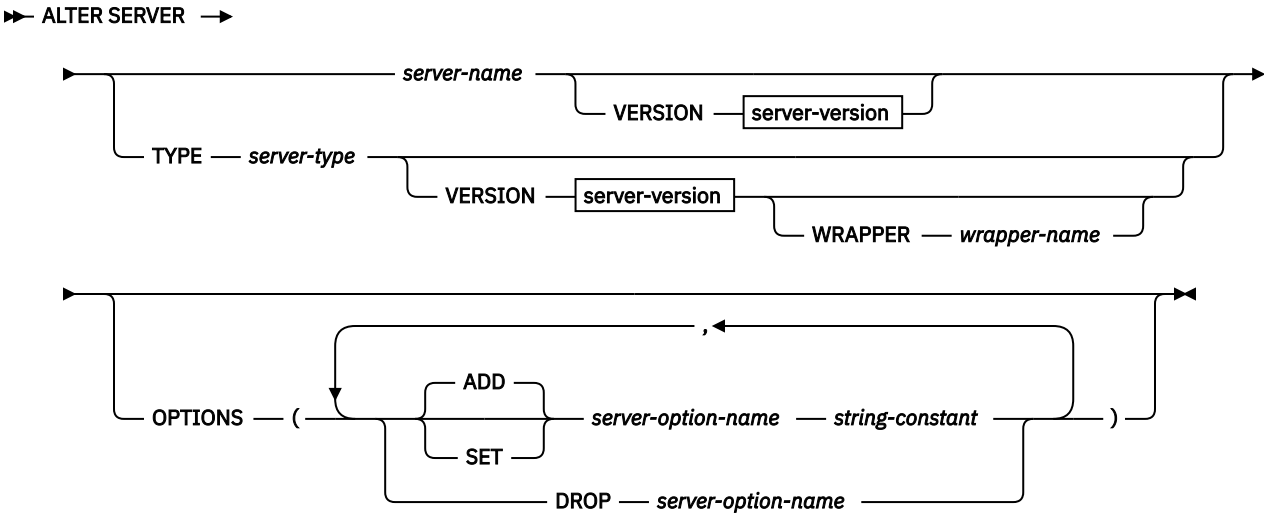
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

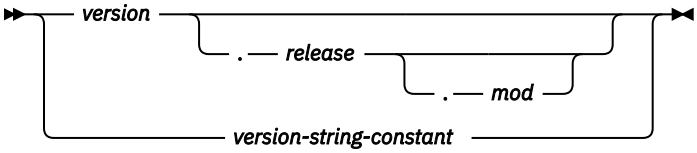
Authorization

The privileges held by the authorization ID of the statement must include DBADM authority.

Syntax



server-version



Description

server-name
Identifies the federated server's name for the data source to which the changes being requested are to apply. The data source must be one that is described in the catalog.

VERSION
After *server-name*, VERSION and its parameter specify a new version of the data source that *server-name* denotes.

version
Specifies the version number. The value must be an integer.

release
Specifies the number of the release of the version denoted by *version.* The value must be an integer.

mod
Specifies the number of the modification of the release denoted by *release.* The value must be an integer.

version-string-constant
Specifies the complete designation of the version. The *version-string-constant* can be a single value (for example, '8i'); or it can be the concatenated values of *version*, *release* and, if applicable, *mod* (for example, '8.0.3').

TYPE *server-type*
Specifies the type of data source to which the changes being requested are to apply.

VERSION
After *server-type*, VERSION and its parameter specify the version of the data sources for which server options are to be enabled, reset, or dropped.

WRAPPER *wrapper-name*
Specifies the name of the wrapper that the federated server uses to interact with data sources of the type and version denoted by *server-type* and *server-version*. The wrapper must be listed in the catalog.

OPTIONS
Indicates what server options are to be enabled, reset, or dropped for the data source denoted by *server-name*, or for the category of data sources denoted by *server-type* and its associated parameters.

ADD
Enables a new server option.

SET
Changes the setting of a server option.

server-option-name
The server option that is to be added or reset. Which options you can specify depends on the data source of the object for which a wrapper is being created. For a list of data sources and the server options that apply to each, see Data source options.

string-constant
The server option setting as a character string constant enclosed in single quotation marks.

DROP *server-option-name*
Drops a server option.

Notes

- A server option cannot be specified more than once in the same ALTER SERVER statement (SQLSTATE 42853). When a server option is enabled, reset, or dropped, any other server options that are in use are not affected.
- An ALTER SERVER statement within a given unit of work (UOW) cannot be processed (SQLSTATE 55007) under either of the following conditions:
  - The statement references a single data source, and the UOW already includes one of the following:
    - A SELECT statement that references a nickname for a table or view within this data source
    - An open cursor on a nickname for a table or view within this data source
    - Either an INSERT, DELETE, or UPDATE statement issued against a nickname for a table or view within this data source
  - The statement references a category of data sources (for example, all data sources of a specific type and version), and the UOW already includes one of the following:
    - A SELECT statement that references a nickname for a table or view within one of these data sources
    - An open cursor on a nickname for a table or view within one of these data sources
    - Either an INSERT, DELETE, or UPDATE statement issued against a nickname for a table or view within one of these data sources
- If the server option is set to one value for a type of data source, and set to another value for an instance of this type, the second value overrides the first one for the instance. For example, assume that PLAN_HINTS is set to 'Y' for server type ORACLE, and to 'N' for an Oracle data source named DELPHI. This configuration causes plan hints to be enabled at all Oracle data sources except DELPHI.
- You can only alter set or alter drop server options for a category of data sources that was enabled by a prior alter add server option operation (SQLSTATE 42704).
- When altering the server version, no verification occurs to ensure that the specified server version matches the remote server version. Specifying an incorrect server version can result in SQL errors when you access nicknames that belong to the database server definition. This is most likely when you specify a server version that is later than the remote server version. In that case, when you access nicknames that belong to the server definition, the database server might send SQL that the remote server does not recognize.
- Server option HOST and NODE cannot be dropped at the same time (SQLSTATE 428EG).

- Both HOST and NODE keywords are included in the server option list, NODE is used even if NODE specified an unrecognized data source.
- PORT option is not mandatory. When PORT is dropped, ODBC wrapper uses default PORT number according to remote data source. The default PORT number is listed in CREATE SERVER.

Examples

- *Example 1:* Ensure that when authorization IDs are sent to your Oracle 8.0.3 data sources, the case of the IDs will remain unchanged. Also, assume that the local federated server CPU is twice as fast as the data source CPU. Inform the optimizer of this statistic.

```
   ALTER SERVER
     TYPE ORACLE
     VERSION 8.0.3
     OPTIONS
        (ADD FOLD_ID 'N',
        SET CPU_RATIO '2.0')
```

- *Example 2:* Indicate that the Documentum data source called DCTM_SVR_ASIA has been changed to Version 4.

```
   ALTER SERVER DCTM_SVR_ASIA
     VERSION 4
```

- *Example 3:* Drop the NODE option, add new HOST option. ODBC driver uses DSN-less mode instead of DSN connection to access a Hive server.

```
   ALTER SERVER HIVE
OPTIONS (add HOST ‘9.123.111.214’, DROP NODE)
```

ALTER SERVICE CLASS

The ALTER SERVICE CLASS statement alters the definition of a service class.

Invocation

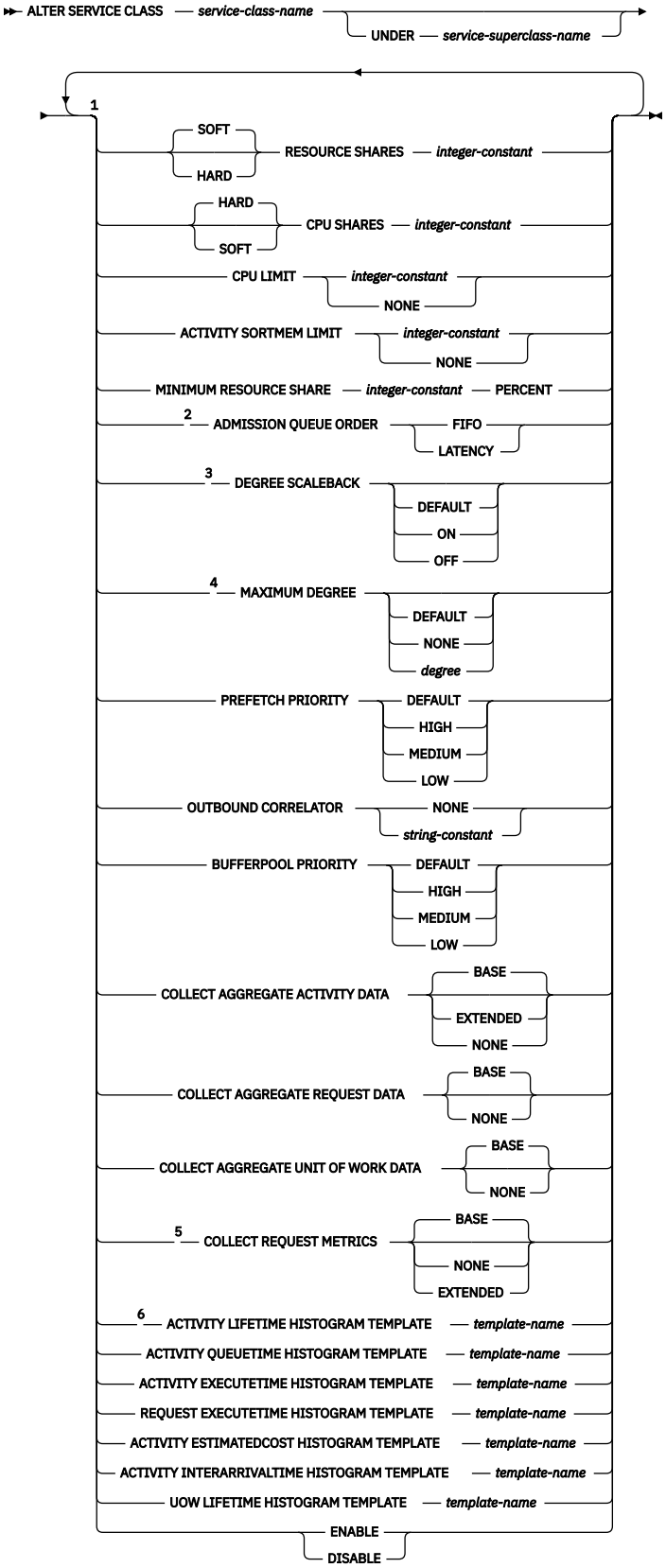
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

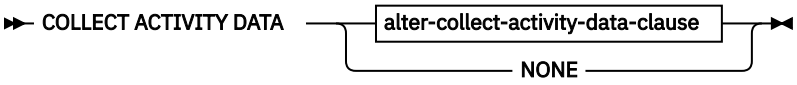
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- SQLADM authority, only if every alteration clause is a COLLECT clause
- WLMADM authority
- DBADM authority

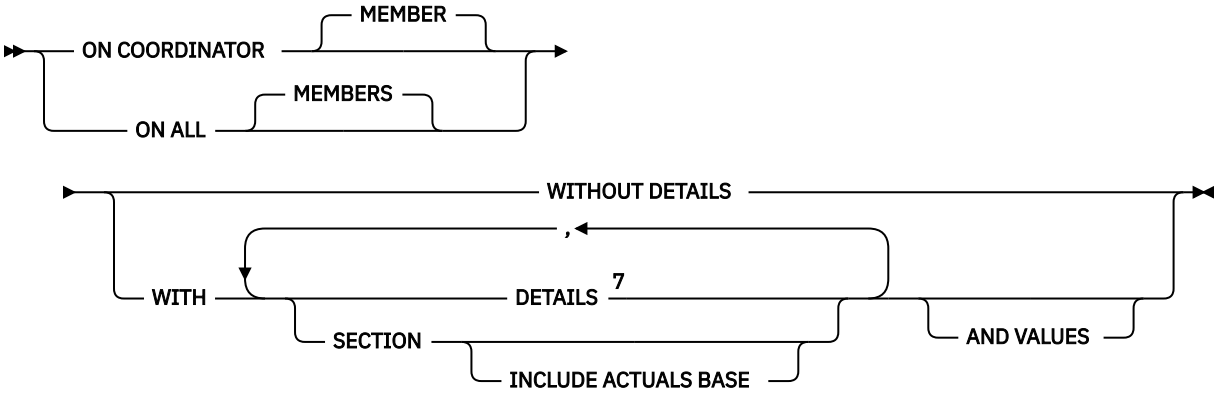
Syntax



collect-activity-data-clause



alter-collect-activity-data-clause



Notes:

[1] The same clause must not be specified more than once (SQLSTATE 42613).
[2] The ADMISSION QUEUE ORDER clause is valid only for a service subclass (SQLSTATE 5U043).
[3] The DEGREE SCALEBACK DEFAULT option is valid only for a service subclass (SQLSTATE 5U043).
[4] The MAXIMUM DEGREE DEFAULT option is valid only for a service subclass (SQLSTATE 5U043).
[5] The COLLECT REQUEST METRICS clause is valid only for a service superclass (SQLSTATE 05U44).
[6] The REQUEST EXECUTETIME AND UOW LIFETIME HISTOGRAM TEMPLATE clauses are valid only for a service subclass (SQLSTATE 05U43).
[7] The DETAILS keyword is the minimum to be specified, followed by the option separated by a comma.

Description

service-class-name
Identifies the service class that is to be altered. This is a one-part name. It is an SQL identifier (either ordinary or delimited).The *service-class-name* must identify a service class that exists in the database (SQLSTATE 42704). To alter a service subclass, the *service-superclass-name* must be specified using the UNDER clause.

UNDER *service-superclass-name*
This clause is used only for altering a service subclass. The *service-superclass-name* identifies the service superclass of the service subclass and must identify a service superclass that exists in the database (SQLSTATE 42704).

RESOURCE SHARES
Specifies the number of shares of resources to which this service class is entitled, and whether the service class is allowed to exceed this number when other service classes in the same scope are not using their full entitlements. This value affects the amount of work the workload manager (WLM) adaptive admission control allows into the system.

HARD
The service class is not allowed to exceed its resource share entitlement.

SOFT
The service class is allowed to exceed its resource share entitlement when other service classes are not using their full entitlements.

Valid values are integers 1 - 65535.

Note: To use resource shares with WLM, you must enable the **wlm_admission_ctrl** configuration parameter.

CPU SHARES
Specifies the number of CPU shares that the workload manager (WLM) dispatcher allocates to this service class when work is executing within this service class, and whether the service class is allowed to exceed this number when other service classes in the same scope are not using their full entitlement.

HARD
The service class is not allowed to exceed its CPU share entitlement.

SOFT
The service class is allowed to exceed its CPU share entitlement when other service classes are not using their full entitlements.

Valid values are integers 1 - 65535.

Note: To use CPU shares with WLM dispatcher, you must enable the **wlm_disp_cpu_shares** database manager configuration parameter.

CPU LIMIT
Specifies the maximum percentage of the CPU resources that the WLM dispatcher can assign to this service class. Valid values for the *integer-constant* are integers between 1 and 100. You can also specify CPU LIMIT NONE to indicate that there is no CPU limit.

ACTIVITY SORTMEM LIMIT

Specifies the maximum percentage of the configured shared sort memory (**SHEAPTHRES_SHR**) that individual queries executing in the service class are allowed to consume. Queries requiring more memory than the configured limit will have individual per-operator **SORTHEAP** values reduced at runtime. Memory requests that exceed the limit will be throttled. Valid values for the integer-constant are integers ranging between 10 and 100. You can also specify NONE to indicate there is no activity sort memory limit. The default is NONE.

The effective sort memory limit for a query will be the most restrictive of the limit defined at the subclass, superclass and database via the **ACT_SORTMEM_LIMIT** database configuration parameter. The sort memory limit applied to an activity is determined when the activity is first admitted for execution. The applied sort memory limit will not change if a query is remapped at runtime to a different service subclass.

The activity sort memory limit will only be enforced for queries that are managed by the adaptive workload manager. If the adaptive workload manager is disabled (**WLM_ADMISSION_CTRL** database config parameter is set to NO,) or a query bypasses the adaptive workload manager, no sort memory limit is applied to the query regardless of which service class it runs in.

Note: Setting an activity sort memory limit too low may result in reduced performance for queries.

MINIMUM RESOURCE SHARE *integer-constant* PERCENT
Specifies the percentage of entitled resources managed by WLM adaptive admission control that is held in reserve for the service class when other service classes exceed their admission resource entitlement. Valid values for the *integer-constant* are integers 0 - 100.

ADMISSION QUEUE ORDER
Specifies the queue order for activities queued by WLM adaptive admission control.

FIFO
Requests are queued in a first-in first-out order. This is the default.

LATENCY
The position of a request in the queue is based on its estimated execution time (that is, its latency) relative to the amount of time that has elapsed since it joined the queue.

DEGREE SCALEBACK
Specifies whether work running in this service class may have its degree scaled back. Queries set to DEGREE ANY may have their actual runtime degree scaled back by the database manager based on current CPU loads.

Scaling back the degree for service classes running simple queries may result in less contention and improved throughput. Disabling degree scale back for service classes with complex queries can

help ensure more consistent and predictable response times. A setting of DEFAULT means a service subclass inherits its DEGREE SCALEBACK setting from the parent superclass. The DEFAULT setting is only applicable to service subclasses. The default setting for a service superclass is ON. The default value for a service subclass is DEFAULT.

MAXIMUM DEGREE
Specifies the maximum runtime degree of parallelism for activities running in this service class. The MAXIMUM DEGREE DEFAULT option is only valid for a service subclass (SQLSTATE 5U043).

DEFAULT
This service subclass should inherit its maximum degree value from its parent superclass. This setting is only applicable to service subclass.

NONE
This service class does not specify a maximum runtime degree for assigned applications. The actual runtime degree is determined as the lower of the value of **max_querydegree** configuration parameter, the value set by SET RUNTIME DEGREE command, the SQL statement compilation degree and the MAXIMUM DEGREE value set on the Workload.

degree
Specifies the maximum degree of parallelism for this service class. Valid values are 1 to 32767. The actual runtime degree is determined as the lower of this degree, the value of **max_querydegree** configuration parameter, the value set by SET RUNTIME DEGREE command, the SQL statement compilation degree and the MAXIMUM DEGREE set on the Workload.

PREFETCH PRIORITY
This parameter controls the priority with which agents in the service class can submit their prefetch requests:

HIGH
Prefetch requests are submitted to the high priority queue.

MEDIUM
Prefetch requests are submitted to the medium low priority queue.

LOW
Prefetch requests are submitted to the low priority queue.

DEFAULT
The default value is DEFAULT, which is internally mapped to MEDIUM for service superclasses. If DEFAULT is specified for a service subclass, it inherits the PREFETCH PRIORITY of its parent superclass.

Other values are not valid (SQLSTATE 42615).

Prefetchers empty the priority queue in order from high to low. Agents in the service class submit their prefetch requests at the PREFETCH PRIORITY level when the next activity begins. If PREFETCH PRIORITY is altered after a prefetch request is submitted, the request priority does not change. PREFETCH PRIORITY cannot be altered for a default subclass (SQLSTATE 5U032).

OUTBOUND CORRELATOR
Specifies whether or not to associate threads from this service class to an external workload manager service class.

If OUTBOUND CORRELATOR is set to a *string-constant* for the service superclass and OUTBOUND CORRELATOR NONE is set for a service subclass, the service subclass inherits the OUTBOUND CORRELATOR of its parent.

OUTBOUND CORRELATOR NONE
For a service superclass, specifies that there is no external workload manager service class association with this service class, and for a service subclass, specifies that the external workload manager service class association is the same as its parent.

OUTBOUND CORRELATOR *string-constant*
Specifies the *string-constant* that is to be used as a correlator to associate threads from this service class to an external workload manager service class. The external workload manager must

be active (SQLSTATE 5U030). The external workload manager should be set up to recognize the value of the specified string constant.

BUFFERPOOL PRIORITY
This parameter controls the bufferpool priority (HIGH, MEDIUM, or LOW) of pages fetched by activities in this service class. If DEFAULT is specified for a service subclass, it inherits the BUFFERPOOL PRIORITY from its parent superclass. Other values are not valid (SQLSTATE 42615).

Pages fetched by activities in a service class with higher bufferpool priority are less likely to be swapped out than pages fetched by activities in a service class with lower bufferpool priority. BUFFERPOOL PRIORITY cannot be altered for a default subclass (SQLSTATE 5U032).

COLLECT ACTIVITY DATA
Specifies that information about each activity that executes in this service class is to be sent to any active activities event monitor when the activity completes.

alter-collect-activity-data-clause

ON COORDINATOR MEMBER
Specifies that activity data is to be collected only at the coordinator member of the activity.

ON ALL MEMBERS
Specifies that activity data is to be collected at all members where the activity is processed. On remote members, a record for the activity may be captured multiple times as the activity comes and goes on those members. If the AND VALUES clause is specified, activity input values will be collected only for the members of the coordinator.

WITHOUT DETAILS
Specifies that data about each activity that executes in the service class is to be sent to any active activities event monitor, when the activity completes execution. Details about statement, compilation environment, and section environment data are not sent.

WITH DETAILS
Specifies that statement and compilation environment data is to be sent to any active activities event monitor, for those activities that have them. Section environment data is not sent.

SECTION
Specifies that statement, compilation environment, section environment data, and section actuals are to be sent to any active activities event monitor for those activities that have them. DETAILS must be specified if SECTION is specified. Section actuals will be collected on any partition where the activity data is collected.

INCLUDE ACTUALS BASE
Specifies that section actuals should also be collected on any partition where the activity data is collected. For section actuals to be collected, either INCLUDE ACTUALS clause must be specified or the **section_actuals** database configuration parameter must be set.

The effective setting for the collection of section actuals is the combination of the INCLUDE ACTUALS clause (specified on the WORK ACTION, SERVICE CLASS, or WORKLOAD), the **section_actuals** database configuration parameter, and the <collectsectionactuals> setting specified on the WLM_SET_CONN_ENV routine. For example, if INCLUDE ACTUALS BASE is specified, yet the **section_actuals** database configuration parameter value is NONE and <collectsectionactuals> is set to NONE, then the effective setting for the collection of section actuals is BASE.

BASE specifies that the following actuals should be enabled and collected during the activity's execution:

- Basic operator cardinality counts
- Statistics for each object referenced (DML statements only)

AND VALUES
Specifies that input data values are to be sent to any active activities event monitor, for those activities that have them. This data does not include SQL statements that are compiled by using the REOPT ALWAYS bind option.

NONE
Specifies that activity data should not be collected for each activity that executes in this service class.

COLLECT AGGREGATE ACTIVITY DATA
Specifies that aggregate activity data should be captured for this service class and sent to the statistics event monitor, if one is active. This information is collected periodically on an interval that is specified by the **wlm_collect_int** database configuration parameter. The default is COLLECT AGGREGATE ACTIVITY DATA BASE.

BASE
Specifies that basic aggregate activity data should be captured for this service class and sent to the statistics event monitor, if one is active. Basic aggregate activity data includes:

- Estimated activity cost high watermark
- Rows returned high watermark
- Temporary table space usage high watermark

  Note: Only activities that have an SQLTEMPSPACE threshold applied to them participate in this high watermark.
- Activity life time histogram
- Activity queue time histogram
- Activity execution time histogram

EXTENDED
Specifies that all aggregate activity data should be captured for this service class and sent to the statistics event monitor, if one is active. This includes all basic aggregate activity data plus:

- Activity data manipulation language (DML) estimated cost histogram
- Activity DML inter-arrival time histogram

NONE
Specifies that no aggregate activity data should be captured for this service class.

COLLECT AGGREGATE REQUEST DATA
Specifies that aggregate request data should be captured for this service class and sent to the statistics event monitor, if one is active. This information is collected periodically on an interval specified by the **wlm_collect_int** database configuration parameter. The default is COLLECT AGGREGATE REQUEST DATA NONE. The COLLECT AGGREGATE REQUEST DATA clause is valid only for a service subclass.

BASE
Specifies that basic aggregate request data should be captured for this service class and sent to the statistics event monitor, if one is active.

NONE
Specifies that no aggregate request data should be captured for this service class.

COLLECT AGGREGATE UNIT OF WORK DATA
Specifies that aggregate unit of work data is to be captured for this service class and sent to the statistics event monitor, if one is active. This information is collected periodically on an interval that is specified by the **wlm_collect_int** database configuration parameter. The default, when COLLECT AGGREGATE UNIT OF WORK DATA is specified, is COLLECT AGGREGATE UNIT OF WORK DATA BASE.

BASE
Specifies that basic aggregate unit of work data is to be captured for this service class and sent to the statistics event monitor, if one is active. Basic aggregate unit of work data includes:

- Unit of work lifetime histogram

NONE
Specifies that no aggregate unit of work data is to be collected for this service class.

COLLECT REQUEST METRICS
Specifies that monitor metrics should be collected for any request submitted by a connection that is associated with the specified service superclass and sent to the statistics and unit of work event monitors, if active. The default is COLLECT REQUEST METRICS NONE. The COLLECT REQUEST METRICS clause is only valid for a service superclass (SQLSTATE 50U44).

Note: The effective request metrics collection setting is the combination of the attribute specified by the COLLECT REQUEST METRICS clause on the service superclass associated with the connection submitting the request, and the **mon_req_metrics** database configuration parameter. If either the service superclass attribute or the configuration parameter has a value other than NONE, metrics will be collected for the request.

BASE
Specifies that basic metrics will be collected for any request submitted by a connection associated with the service superclass.

EXTENDED
Specifies that basic metrics will be collected for any request submitted by a connection associated with the service superclass. In addition, specifies that the values for the following monitor elements should be determined with additional granularity:

- **total_section_time**
- **total_section_proc_time**
- **total_routine_user_code_time**
- **total_routine_user_code_proc_time**
- **total_routine_time**

NONE
Specifies that no metrics will be collected for any request submitted by a connection associated with the service superclass.

ACTIVITY LIFETIME HISTOGRAM TEMPLATE
Specifies the template that describes the histogram used to collect statistical data about the duration, in milliseconds, of database activities running in the service class during a specific interval. This time includes both time queued and time executing. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

ACTIVITY QUEUETIME HISTOGRAM TEMPLATE
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, that database activities running in the service class are queued during a specific interval. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

ACTIVITY EXECUTETIME HISTOGRAM TEMPLATE
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, that database activities running in the service class are executing during a specific interval. This time does not include the time spent queued. Activity execution time is collected in this histogram at the coordinator member only. The time does not include idle time. Idle time is the time between the execution of requests belonging to the same activity when no work is being done. An example of idle time is the time between the end of opening a cursor and the start of fetching from that cursor. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

REQUEST EXECUTETIME HISTOGRAM TEMPLATE
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, that database requests running in the service class are executing during a specific interval. This time does not include the time spent queued. Request execution time is collected in this histogram on each member where the request executes. This information is only

collected when the COLLECT AGGREGATE REQUEST DATA clause is specified with the BASE option. This clause is only valid for a service subclass.

ACTIVITY ESTIMATEDCOST HISTOGRAM TEMPLATE
Specifies the template that describes the histogram used to collect statistical data about the estimated cost, in timerons, of DML activities running in the service class. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified with the EXTENDED option.

ACTIVITY INTERARRIVALTIME HISTOGRAM TEMPLATE
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, between the arrival of one DML activity and the arrival of the next DML activity. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified with the EXTENDED option.

UOW LIFETIME HISTOGRAM TEMPLATE
Specifies the template that describes the histogram used to collect statistical data about the duration, in milliseconds, of units of work running in the service class during a specific interval. The default is SYSDEFAULTHISTOGRAM. This information is collected only when the COLLECT AGGREGATE UNIT OF WORK DATA clause is specified with the BASE option.

ENABLE or DISABLE
Specifies whether or not connections and activities can be mapped to the service class.

ENABLE
Connections and activities can be mapped to the service class.

DISABLE
Connections and activities cannot be mapped to the service class. New connections or activities that are mapped to a disabled service class will be rejected (SQLSTATE 5U028). When a service superclass is disabled, its service subclasses are also disabled. When the service superclass is re-enabled, its service subclasses return to states that are defined in the system catalog. A default service class cannot be disabled (SQLSTATE 5U032).

Rules

- A workload management (WLM)-exclusive SQL statement must be followed by a COMMIT or a ROLLBACK statement (SQLSTATE 5U021). WLM-exclusive SQL statements are:
  - CREATE HISTOGRAM TEMPLATE, ALTER HISTOGRAM TEMPLATE, or DROP (histogram template)
  - CREATE SERVICE CLASS, ALTER SERVICE CLASS, or DROP (service class)
  - CREATE THRESHOLD, ALTER THRESHOLD, or DROP (threshold)
  - CREATE WORK ACTION SET, ALTER WORK ACTION SET, or DROP (work action set)
  - CREATE WORK CLASS SET, ALTER WORK CLASS SET, or DROP (work class set)
  - CREATE WORKLOAD, ALTER WORKLOAD, or DROP (workload)
  - GRANT (workload privileges) or REVOKE (workload privileges)
- A WLM-exclusive SQL statement cannot be issued within a global transaction (SQLSTATE 51041) such as, for example, an XA transaction.

Notes

- Only one uncommitted WLM-exclusive SQL statement at a time is allowed across all members. If an uncommitted WLM-exclusive SQL statement is executing, subsequent WLM-exclusive SQL statements will wait until the current WLM-exclusive SQL statement commits or rolls back.
- Changes are written to the system catalog, but do not take effect until after a COMMIT statement, even for the connection that issues the statement.
- After the ALTER SERVICE CLASS statement is committed, changes to PREFETCH PRIORITY, OUTBOUND CORRELATOR, and COLLECT take effect for the next new activity in the service class. Existing activities in the service class continue to complete their work using the old settings.

- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - o DATABASE PARTITION can be specified in place of MEMBER, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
  - o DATABASE PARTITIONS can be specified in place of MEMBERS, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.

Examples

- *Example 1:* Alter the amount of CPU that can be consumed by work running in service superclass PETSALES to a maximum of 50%.

```
ALTER SERVICE CLASS PETSALES CPU LIMIT 50
```

- *Example 2:* Alter service superclass BARNSALES and add an outbound correlator 'osLowPriority'. Threads running in the service superclass and its service subclasses will have the outbound correlator 'osLowPriority' associated with them.

```
ALTER SERVICE CLASS BARNSALES OUTBOUND CORRELATOR 'osLowPriority'
```

ALTER STOGROUP

The ALTER STOGROUP statement is used to alter the definition of a storage group.

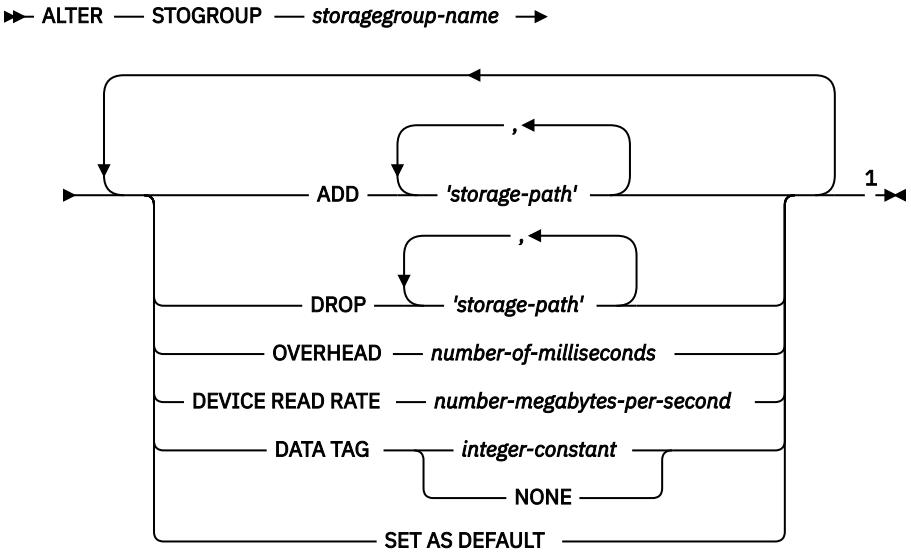
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include SYSCTRL or SYSADM authority.

Syntax



Notes:

[1] Each clause can be specified only once.

Description

storagegroup-name
Identifies the storage group to be altered; *storagegroup-name* must identify a storage group that exists at the current server (SQLSTATE 42704). This is a one-part name.

ADD
Specifies that one or more new storage paths are to be added to the specified storage group.

storage-path
A string constant that specifies containers the location where automatic storage table spaces are to be created. The format of the string depends on the operating system, as illustrated in the following table:

Operating system	Format of storage path string
Linux AIX	An absolute path
Windows	The letter name of a drive

The string can include database partition expressions to specify database partition number information in the storage path. For predictable performance, ensure the storage paths added to a storage group have similar media characteristics.

The maximum length of a storage path is 175 characters (SQLSTATE 54036).

A storage path being added must be valid according to the naming rules for paths, and must be accessible (SQLSTATE 57019). Similarly, in a partitioned database environment, the storage path must exist and be accessible on every database partition (SQLSTATE 57019).

DROP
Specifies that one or more storage paths are to be removed from the given storage group. If table spaces are actively using a storage path being dropped, then the state of the storage path is changed from "In Use" to "Drop Pending" and future use of the storage path will be prevented.

The DROP *storage-path* clause is not supported in a Db2 pureScale environment (SQLSTATE 56038).

storage-path
A string constant that specifies the storage path from which storage groups are to be dropped. The format of the string depends on the operating system, as illustrated in the following table:

Operating system	Format of storage path string
Linux AIX	An absolute path
Windows	The letter name of a drive

The string can include database partition expressions to specify database partition number information in the storage path.

A storage path being dropped must currently exist in the storage group (SQLSTATE 57019) and cannot already be in the "Drop Pending" state (SQLSTATE 55073).

OVERHEAD *number-of-milliseconds*
Specifies the I/O controller usage and disk seek and latency time. This value is used to determine the cost of I/O during query optimization. The value of *number-of-milliseconds* is any numeric literal (integer, decimal, or floating point). If this value is not the same for all storage paths, set the value to a numeric literal which represents the average for all storage paths that belong to the storage group.

DEVICE READ RATE *number-megabytes-per-second*
Represents the device specification for the read transfer rate in megabytes per second. This value is used to determine the cost of I/O during query optimization. The value of *number-megabytes-per-second* is any numeric literal (integer, decimal, or floating point). If this value is not the same for all

storage paths, set the value to a numeric literal which represents the average for all storage paths that belong to the storage group.

DATA TAG *integer-constant* or DATA TAG NONE
Specifies a tag for the data in a given storage group. This value can be used as part of a WLM configuration in a work class definition or referenced within a threshold definition. For more information, see the CREATE WORK CLASS SET, ALTER WORK CLASS SET, CREATE THRESHOLD, and ALTER THRESHOLD statements.

integer-constant
Valid values for *integer-constant* are integers from 1 to 9.

NONE
If NONE is specified, there is no data tag.

SET AS DEFAULT
Specifies that the storage group being altered is designated as the default storage group. There can be only one storage group designated as the default storage group. There is no affect to the existing table spaces using that storage group. The designated default storage group is used by automatic storage table spaces when no storage group is specified at table space creation and a database managed table space is converted to automatic storage managed during redirected restore.

Rules

- A storage group must have at least one storage path. Dropping all storage paths from the storage group is not permitted (SQLSTATE 428HH).
- The ALTER STOGROUP statement cannot be executed while a database partition server is being added (SQLSTATE 55071).
- A storage group can have up to 128 defined storage paths (SQLSTATE 5U009).
- A transaction can have at most one ALTER STOGROUP statement per storage group. In the case of the default storage group, there can be at most one ALTER DATABASE statement or one ALTER STOGROUP statement on the default storage group (SQLSTATE 25502).

Notes

- ***Adding new storage paths:*** When adding new storage paths:
  - Existing REGULAR and LARGE table spaces using this storage group will not initially use these new paths. The database manager might choose to create new table space containers on these paths only if an out-of-space condition occurs. You can issue ALTER TABLESPACE REBALANCE statements for existing table spaces to stripe them over the newly added storage paths.
  - Existing temporary table spaces managed by automatic storage do not automatically use new storage paths. The database must be stopped normally then restarted for containers in these table spaces to use the new storage path or paths. As an alternative, the temporary table spaces can be dropped and re-created. When created, these table spaces automatically use all storage paths that have sufficient free space.
- ***Calculation of free space:*** When free space is calculated for a storage path on a database partition, the database manager checks for the existence of the following directories or mount points within the storage path, and will use the first one that is found.

```
<storage path>/<instance name>/NODE####/<database name>
<storage path>/<instance name>/NODE####
<storage path>/<instance name>
<storage path>
```

Where:

  - `<storage path>` is a storage path associated with the database.
  - `<instance name>` is the instance under which the database resides.
  - `NODE####` corresponds to the database partition number (for example, NODE0000 or NODE0001).

– <database name> is the name of the database.

- ***Isolating multiple database partitions under one storage path:*** File systems can be mounted at a point beneath the storage path, and the database manager will recognize that the actual amount of free space available for table space containers might not be the same amount that is associated with the storage path directory itself.

  Consider an example in which two logical database partitions exist on one physical computer, and there is a single storage path (/dbdata). Each database partition will use this storage path, but you might want to isolate the data from each partition within its own file system. In this case, a separate file system can be created for each partition and it can be mounted at /dbdata/<instance>/NODE####. When creating containers on the storage path and determining free space, the database manager will not retrieve free space information for /dbdata, but instead will retrieve it for the corresponding /dbdata/<instance>/NODE#### directory.
- ***Dropping a storage path that is in use by one or more table spaces:*** When dropping a storage path that is in use by one or more table spaces, the state of the path changes from "In Use" to "Drop Pending". Future growth on the path will not occur.

  Before the path can be fully removed from the storage group, each affected table space must be rebalanced (using the REBALANCE clause of the ALTER TABLESPACE statement) so that its container data is moved off the storage path. Rebalance is supported only for REGULAR and LARGE table spaces. Drop and re-create temporary table spaces to have their containers removed from the dropped path. When the path is no longer in use by any table space, it will be physically removed from the database.

  For a partitioned database environment, the path is maintained independently on each partition. When a path is no longer in use on a given database partition, it will be physically removed from that partition. Other partitions might still show the path as being in the "Drop Pending" state. The list of automatic storage table spaces using drop pending storage paths can be determined by issuing the following SQL statement:

```
SELECT DISTINCT TBSP_NAME, TBSP_ID, TBSP_CONTENT_TYPE
  FROM TABLE(MON_GET_TABLESPACE(NULL,-2)) AS T
  WHERE TBSP_PATHS_DROPPED = 1
```

- ***Dropping a storage path that was added to a storage group multiple times:*** It is possible for a given storage path to be added to a storage group multiple times. When using the DROP clause, specifying that particular path once will drop all instances of the path from the storage group.

Examples

1. Add drives D and E to the storage group named COMPLIANCE.

```
ALTER STOGROUP COMPLIANCE ADD 'D:\', 'E:\'
```

2. Add storage paths to the storage group named COMPLIANCE.

```
ALTER STOGROUP COMPLIANCE ADD '/db/filesystem3', '/db/filesystem4'
```

3. Change the data tag for the OPERATIONAL storage group and designate it as the default storage group.

```
ALTER STOGROUP OPERATIONAL DATA TAG 3 SET AS DEFAULT
```

4. Add a storage path that uses a database partition expression to differentiate the storage paths on each of the database partitions.

```
ALTER STOGROUP TESTDATA ADD '/dataForPartition $N'
```

5. Remove paths /db/filesystem1 and /db/filesystem2 from storage group TESTDATA.

```
ALTER STOGROUP TESTDATA DROP '/db/filesystem1', '/db/filesystem2'
```

ALTER TABLE

The ALTER TABLE statement alters the definition of a table.

Invocation

This statement can be embedded in an application program or issued by using dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges that are held by the authorization ID of the statement must include at least one of the following authorities:

- ALTER privilege on the table to be altered
- CONTROL privilege on the table to be altered
- ALTERIN privilege on the schema of the table
- SCHEMAADM authority on the schema of the table
- DBADM authority

To create or drop a foreign key, the privileges that are held by the authorization ID of the statement must include one of the following authorities on the parent table:

- REFERENCES privilege on the table
- REFERENCES privilege on each column of the specified parent key
- CONTROL privilege on the table
- SCHEMAADM authority on the schema of the table
- DBADM authority

To drop the primary key or a unique constraint on table T, the privileges of the authorization ID of the statement must include at least one of the following authorities on every table that is a dependent of T's parent key:

- ALTER privilege on the table
- CONTROL privilege on the table
- ALTERIN privilege on the schema of the table
- SCHEMAADM authority on the schema of the table
- DBADM authority

To alter a table to become a materialized query table (by using a fullselect), the privileges that are held by the authorization ID of the statement must include at least one of the following authorities:

- CONTROL privilege on the table
- SCHEMAADM authority on the schema of the table
- DBADM authority

and at least one of the following authorities on each table or view that is identified in the fullselect (excluding group privileges):

- All of the following:
  - SELECT privilege on the table or view or SELECTIN privilege on the schema of the table or view
  - ALTER privilege on the table or view or ALTERIN privilege (including group privileges) on the schema of the table or view
- CONTROL privilege on the table or view
- DATAACCESS on the schema of the table or view

- DATAACCESS authority on the database

To alter a table so that it is no longer a materialized query table, the privileges of the authorization ID of the statement must include at least one of the following authorities on each table or view that is identified in the fullselect that is used to define the materialized query table:

- ALTER privilege on the table or view
- CONTROL privilege on the table or view
- ALTERIN privilege on the schema of the table or view
- SCHEMAADM authority on the schema of the table
- DBADM authority

To add a column of type DB2SECURITYLABEL to a table, the privileges of the authorization ID of the statement must include at least a security label from the security policy that is associated with the table.

To remove the security policy from a table, the privileges that are held by the authorization ID of the statement must include SECADM authority.

To alter a table to attach a data partition, the privileges of the authorization ID of the statement must also include at least one of the following authorities on the source table:

- SELECT privilege on the table or SELECTIN privilege on the schema containing the source table and DROPIN privilege on the schema of the table
- CONTROL privilege on the table
- DATAACCESS authority on the schema of the table
- DATAACCESS authority on the database

and at least one of the following authorities on the target table:

- All of the following:
  - ALTER privilege on the table or ALTERIN privilege on the schema of the table
  - INSERT privilege on the table or INSERTIN privilege on the schema of the table
- CONTROL privilege on the table
- DATAACCESS authority on the schema of the table
- DATAACCESS authority on the database

To alter a table to detach a data partition, the privileges of the authorization ID of the statement must also include at least one of the following authorities on the target table of the detached partition:

- CREATETAB authority on the database, and USE privilege on the table spaces used by the table, and one of:
  - IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the new table does not exist
  - CREATEIN privilege on the schema, if the schema name of the new table refers to an existing schema
  - SCHEMAADM authority on the schema of the table and USE privilege on the table spaces that are used by the table
- DBADM authority

and at least one of the following authorities on the source table:

- All of the following:
  - SELECT privilege on the table or SELECTIN privilege on the schema of the table
  - ALTER privilege on the table or ALTERIN privilege on the schema of the table
  - DELETE privilege on the table or DELETEIN privilege on the schema of the table
- CONTROL privilege on the table
- DATAACCESS authority on the schema of the table

- DATAACCESS authority on the database

To alter a table to activate not logged initially with empty table, the privileges that are held by the authorization ID of the statement must include at least one of the following authorities:

- All of the following:
  - ALTER privilege on the table or ALTERIN privilege on the schema of the table
  - DELETE privilege on the table or DELETEIN privilege on the schema of the table
- CONTROL privilege on the table
- SCHEMAADM authority on the schema of the table
- DBADM authority

To alter a table that is protected by a security policy to activate not logged initially with empty table, the privileges that are held by the authorization ID of the statement must include at least one of the following authorities:

- CONTROL privilege on the table
- SCHEMAADM authority on the schema of the table
- DBADM authority

To alter a table to ACTIVATE and DEACTIVATE row and column access control, the privileges that are held by the authorization ID of the statement must include the SECADM authority.

To alter a table with ACTIVATE NOT LOGGED INITIALLY WITH EMPTY TABLE, if that table activated row access control, the privileges that are held by the authorization ID of the statement must include at least one of the following authorities:

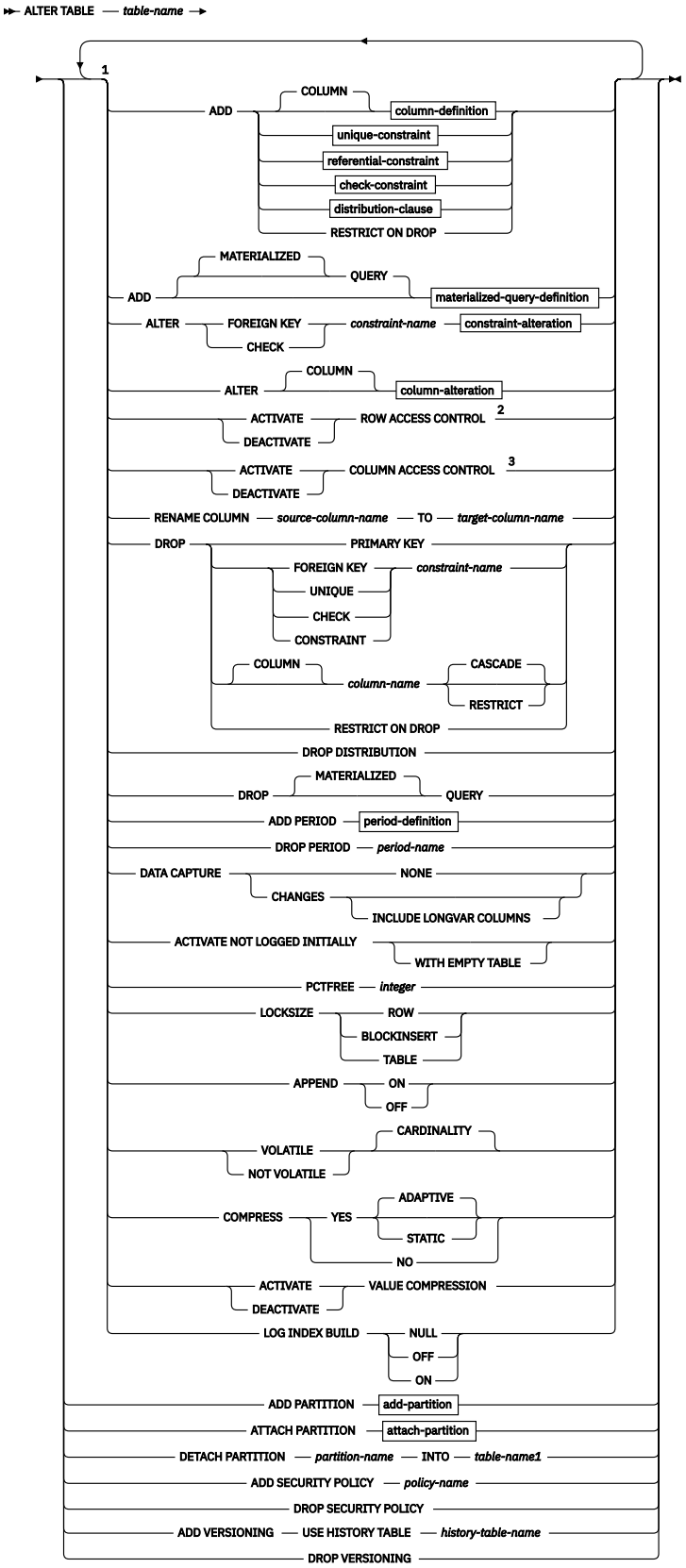
- CONTROL privilege on the table
- SCHEMAADM authority on the schema of the table
- DBADM authority

When altering system period temporal tables, one of the following authorities is required by the authorization ID of the statement:

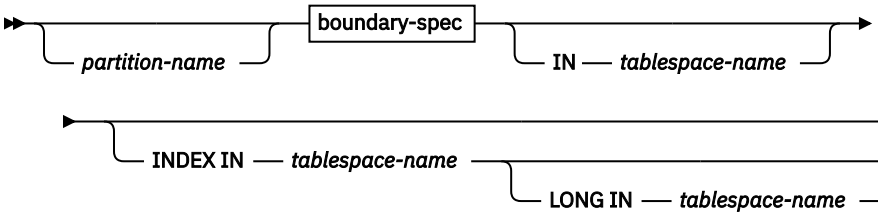
- ALTER privilege on the history table
- CONTROL privilege on the history table
- ALTERIN privilege on the schema of the history table
- SCHEMAADM authority on the schema of the history table
- DBADM authority

This condition applies when altering a table to become a system-period temporal table (with the ADD VERSIONING clause). It also applies when altering a system-period temporal table where one or more changes result in changes to the associated history table.

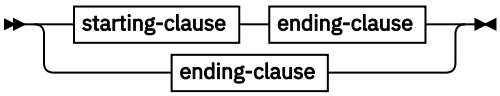
Syntax



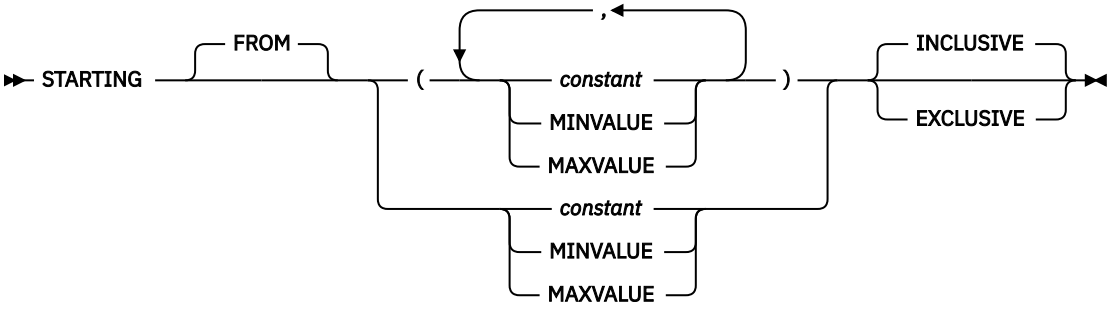
add-partition



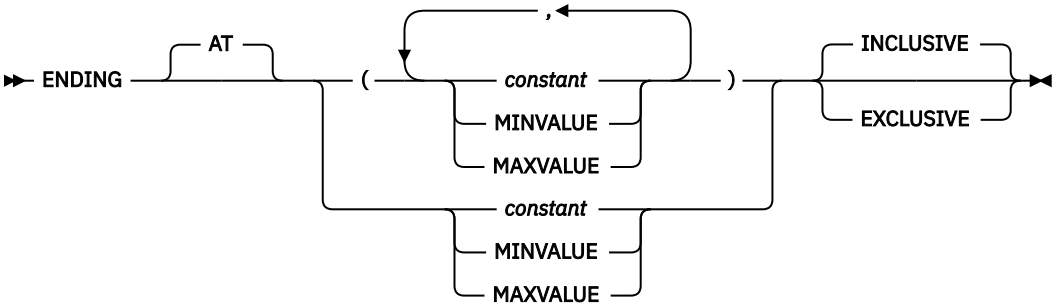
boundary-spec



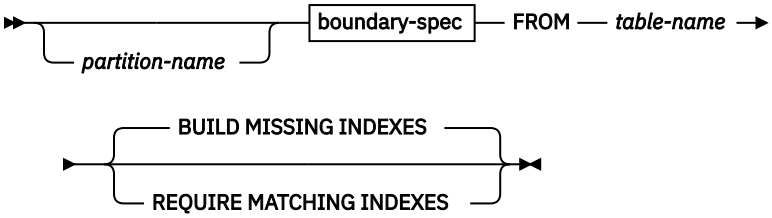
starting-clause



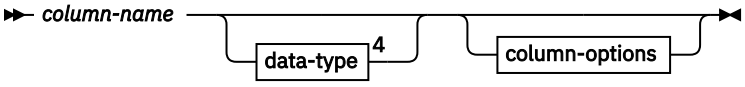
ending-clause



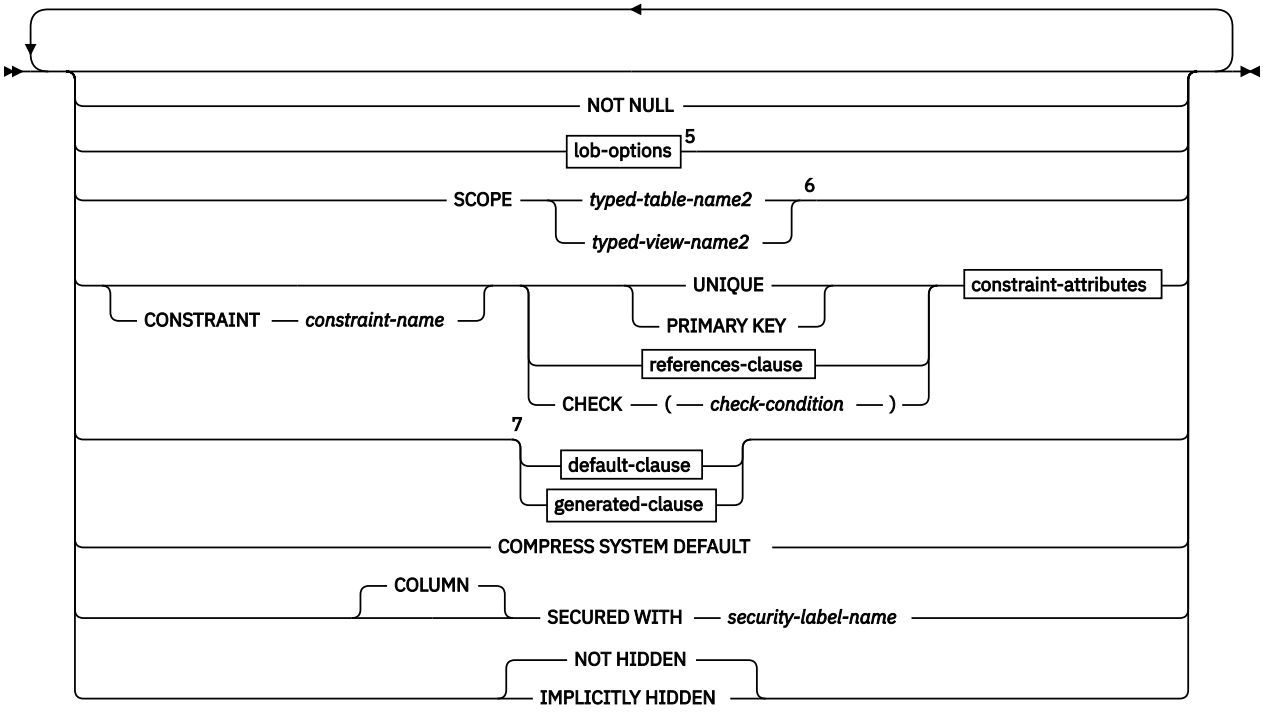
attach-partition



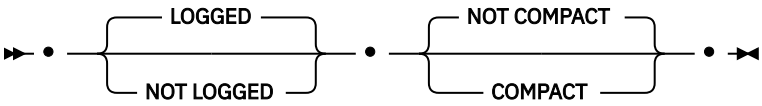
column-definition



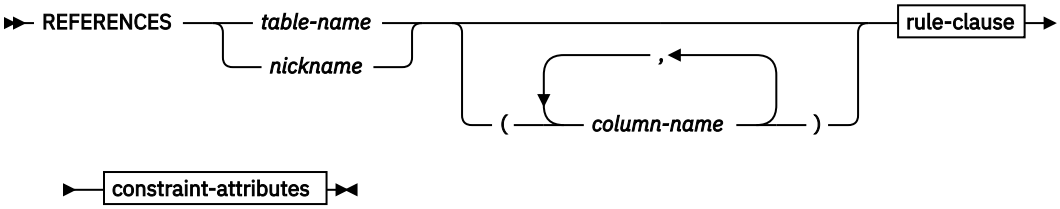
column-options



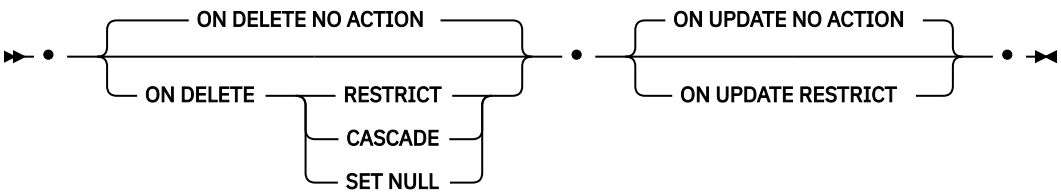
lob-options



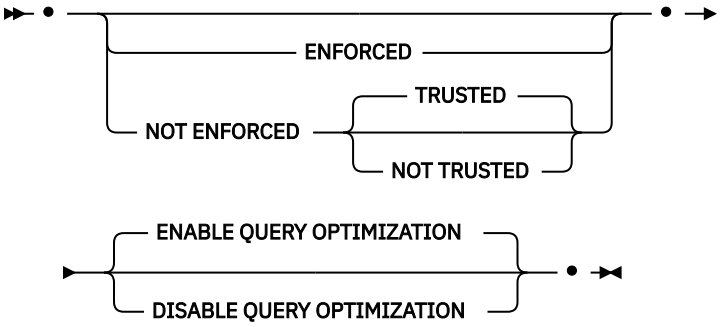
references-clause



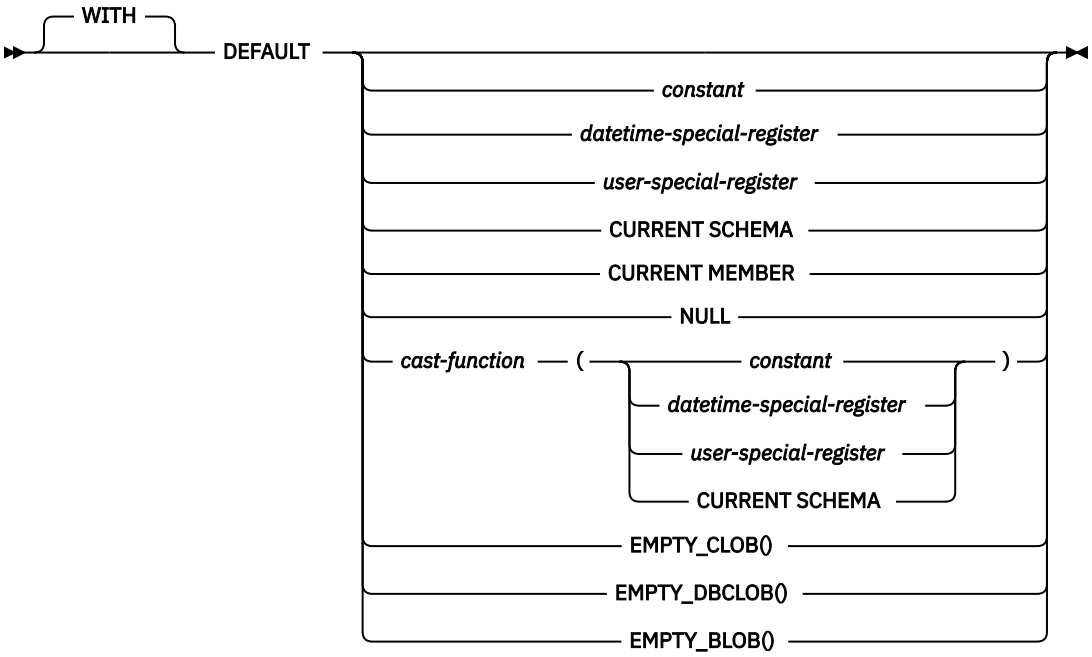
rule-clause



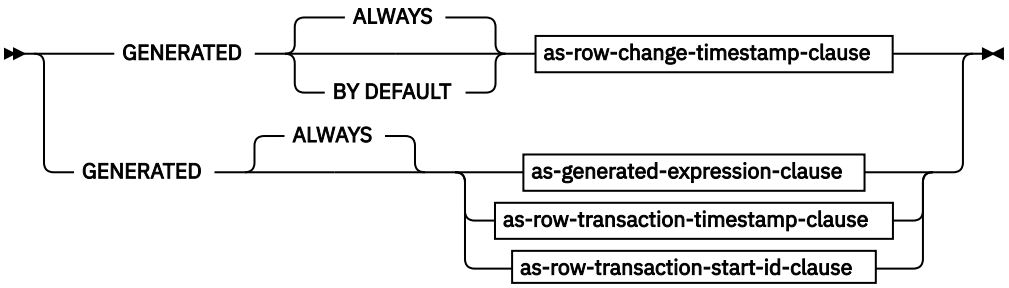
Constraint-attributes



default-clause



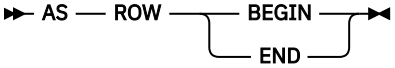
generated-clause



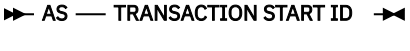
as-row-change-timestamp-clause

as-generated-expression-clause

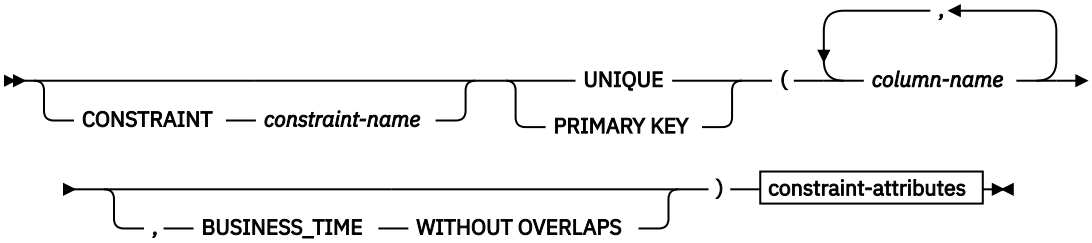
as-row-transaction-timestamp-clause



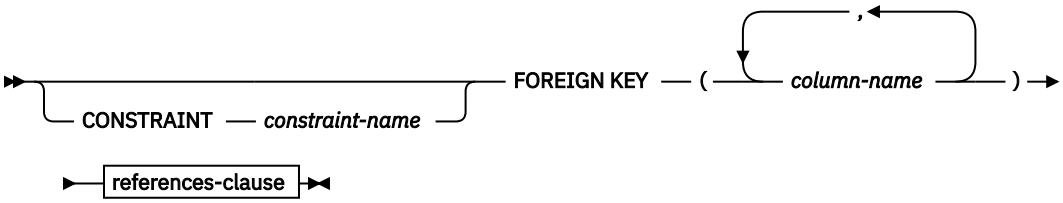
as-row-transaction-start-id-clause



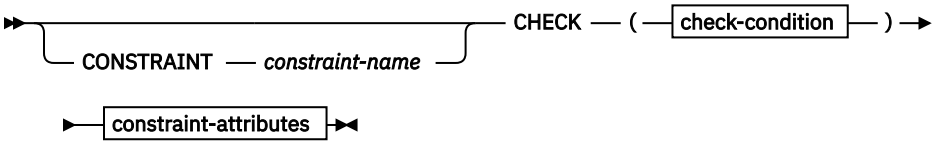
unique-constraint



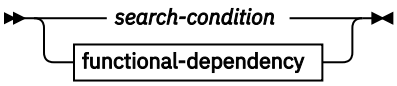
referential-constraint



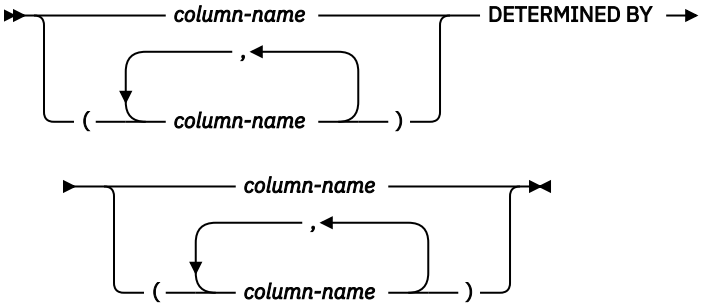
check-constraint



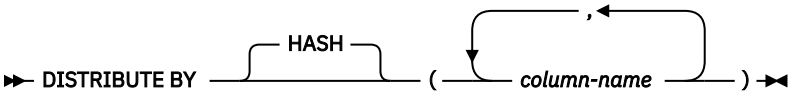
check-condition



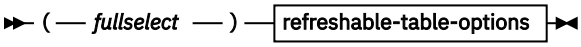
functional-dependency



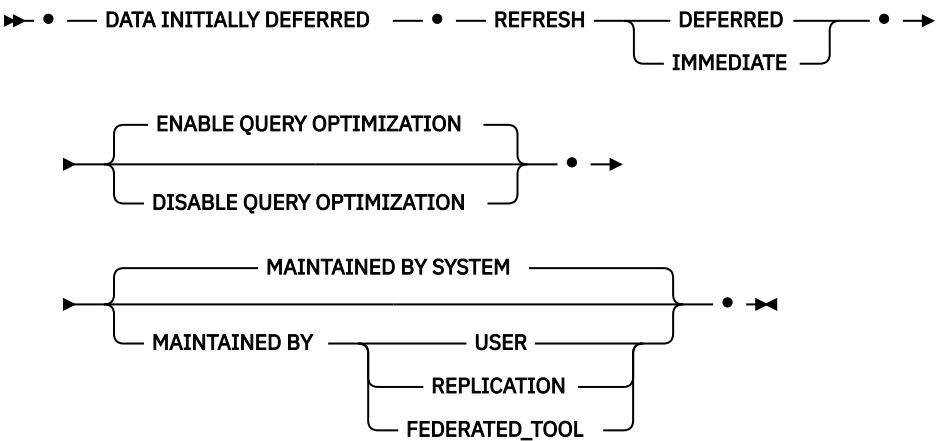
distribution-clause



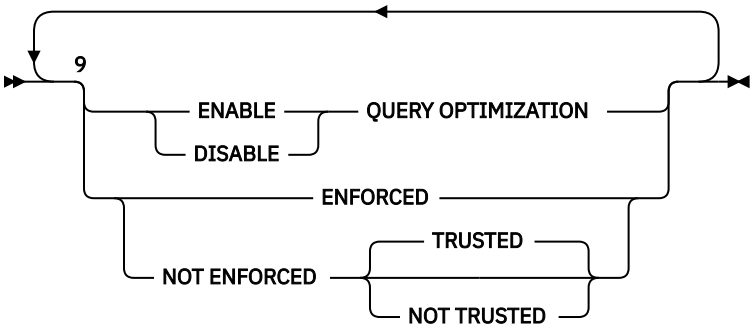
materialized-query-definition



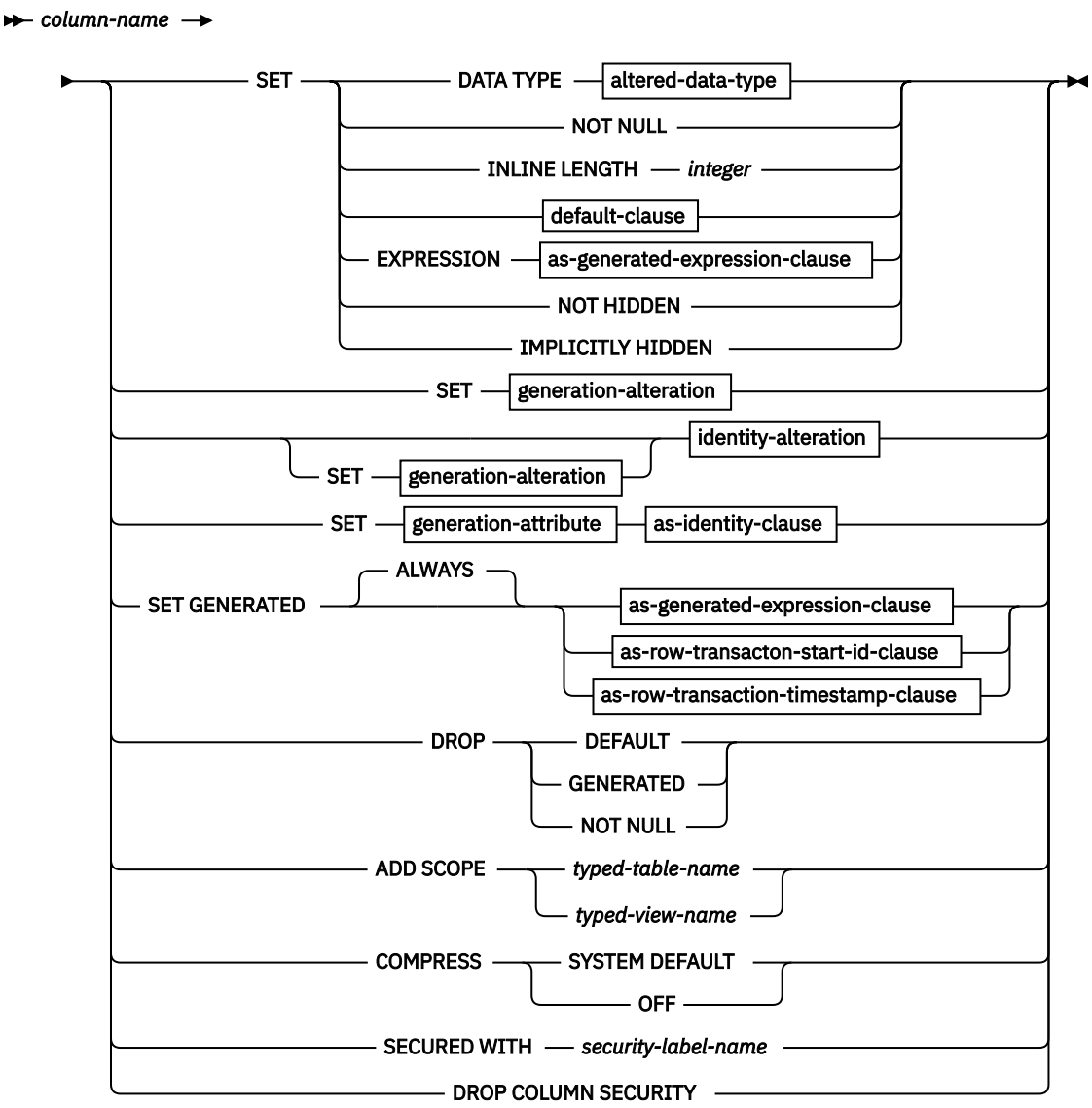
refreshable-table-options



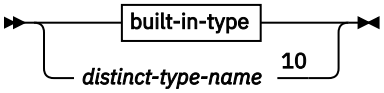
constraint-alteration



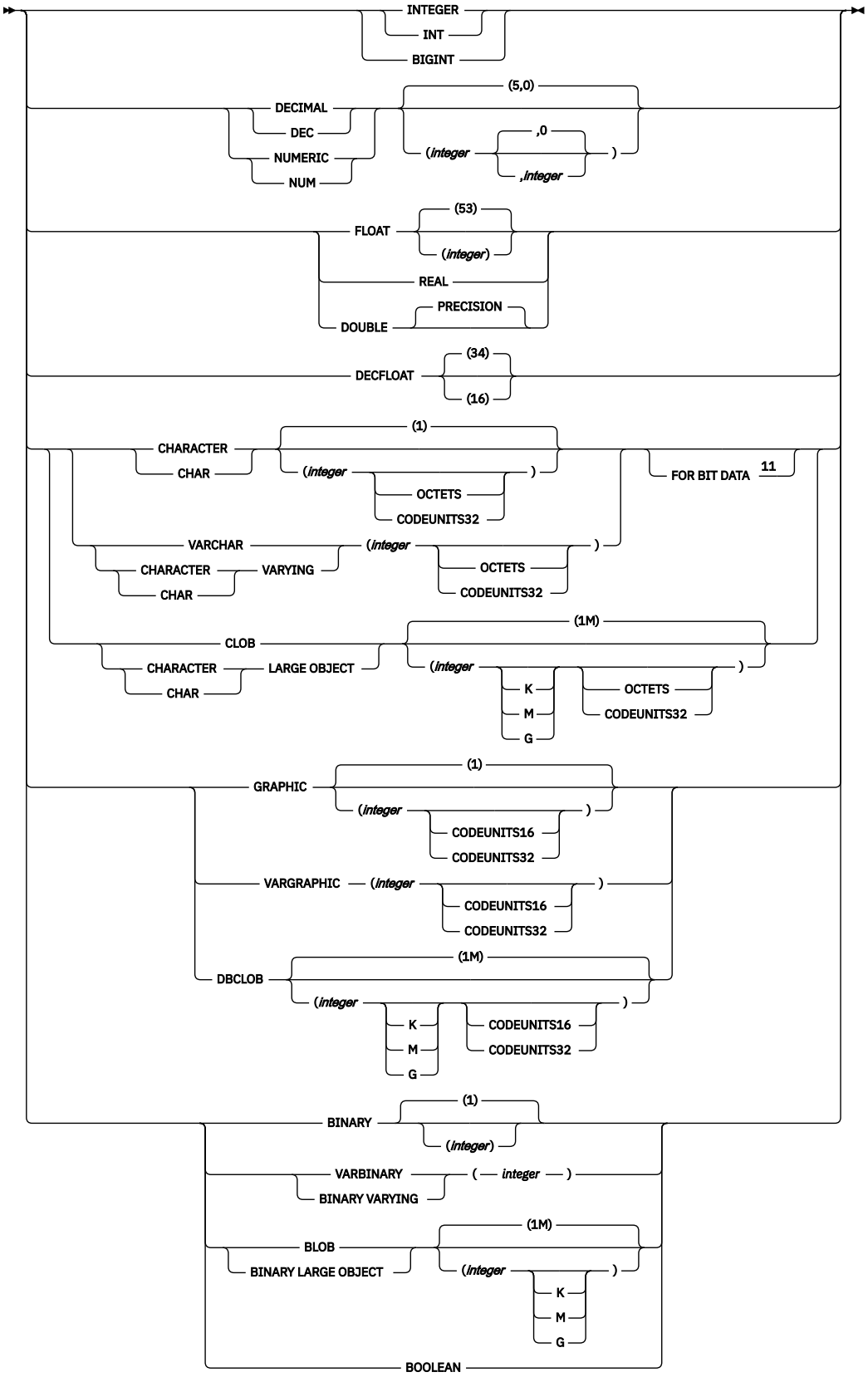
column-alteration



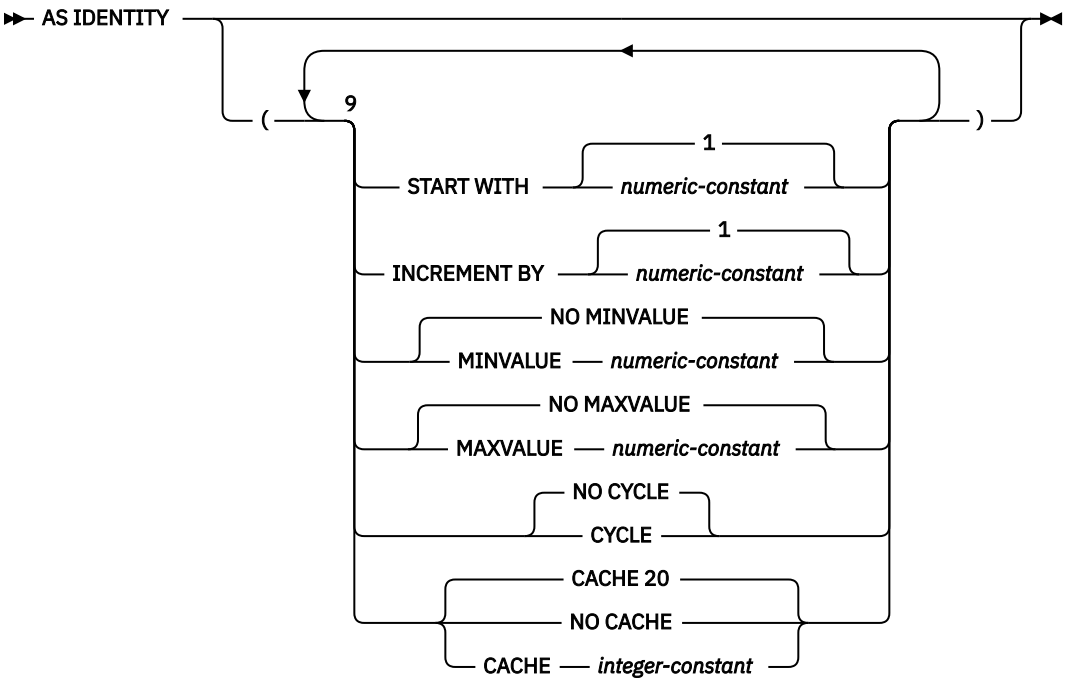
altered-data-type



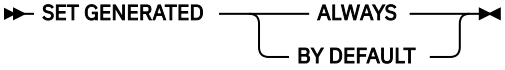
built-in-type



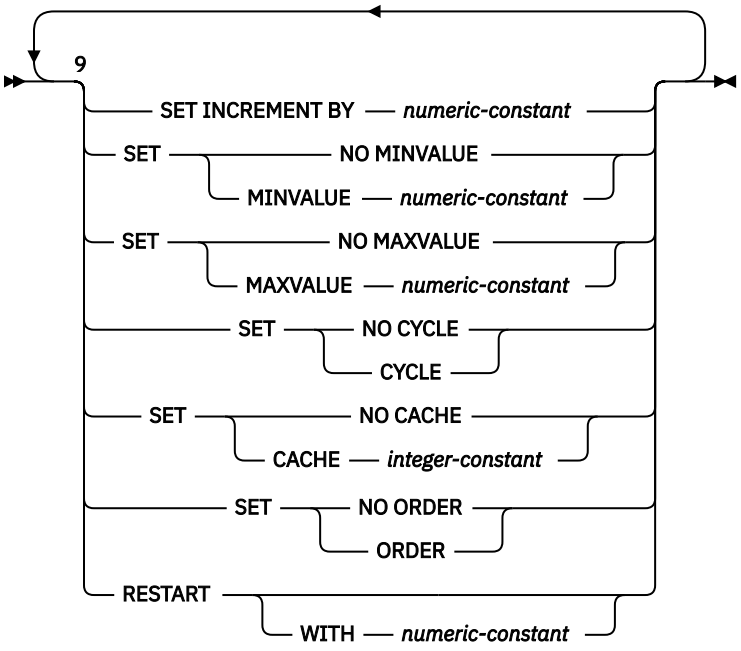
as-identity-clause



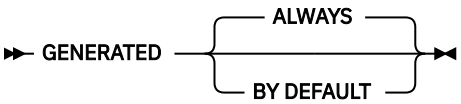
generation-alteration



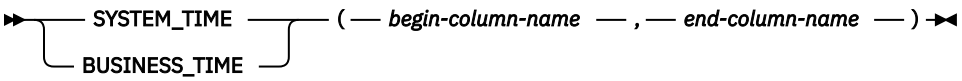
identity-alteration



generation-attribute



period-definition



Notes:

[1] The same clause must not be specified more than once (SQLSTATE 42614).
[2] If an ACTIVATE or DEACTIVATE clause is specified for row access control, no other clause except ACTIVATE or DEACTIVATE column access control can be specified in the same ALTER TABLE statement (SQLSTATE 42613).
[3] If an ACTIVATE or DEACTIVATE clause is specified for column access control, no other clause except ACTIVATE or DEACTIVATE row access control can be specified in the same ALTER TABLE statement (SQLSTATE 42613).
[4] If the first column option chosen is *generated-clause*, *data-type* can be omitted; it is computed by the generation expression.
[5] The *lob-options* clause applies to large object types (CLOB, DBCLOB, and BLOB), and to distinct types that are based on large object types only.
[6] The SCOPE clause applies to the REF type only.
[7] The default-clause and generated-clause cannot both be specified for the same column definition (SQLSTATE 42614).
[8] Data type is optional for a row change timestamp column if the first column-option specified is a generated-clause. The data type default is TIMESTAMP(6). Data type is optional for row-begin, row-end, and transaction-start-ID columns if the first column-option is a generated-clause; the data type default is TIMESTAMP(12).
[9] The same clause must not be specified more than once.
[10] The specified distinct type cannot have any data type constraints and the source type cannot be an anchored data type (SQLSTATE 428H2).
[11] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

Description

table-name
The *table-name* must identify a table that exists at the current server. It cannot be a nickname (SQLSTATE 42809) and must not be a view, a catalog table, a created temporary table, or a declared temporary table (SQLSTATE 42995).

If *table-name* identifies a materialized query table, alterations are limited to adding or dropping the materialized query, starting the ACTIVATING NOT LOGGED INITIALLY clause, adding or dropping RESTRICT ON DROP, modifying data capture, pctfree, locksize, append, volatile, data row compression, value compression, and activating or deactivating row and column access control.

If *table-name* identifies a shadow table, alterations can include adding or dropping primary keys in addition to the alterations that are supported for a materialized query table.

If *table-name* identifies a range-clustered table, alterations are limited to adding, changing, or dropping constraints, activating not logged initially, adding or dropping RESTRICT ON DROP, changing locksize, data capture, or volatile, and setting column default values.

ADD *column-definition*
Adds a column to the table.

A column cannot be added to the following tables:

- A history table for a system-period temporal table (SQLSTATE 428HZ)
- A typed table (SQLSTATE 428DH)
- A table whose compression dictionary is being created in an asynchronous background operation (SQL20054N)

For all existing rows in the table, the value of the new column is set to its default value. The new column is the last column of the table; that is, if initially *n* columns exist, the added column is column *n*+1.

Adding the new column must not cause the total byte count of all columns to exceed the maximum record size.

If the table is a column-organized table, a LOB column can not be added to an existing table.

Note: This restriction has been removed starting from Db2version 11.5.1

If the table is a system-period temporal table, the column is added to the associated history table as well.

If the added column is a generated column that is based on an expression, the expression must not reference a column for which a column mask is defined (SQLSTATE 42621).

If a column is added to a table on which a mask or a permission is defined, or to a table that is referenced in the definition of a mask or a permission, that mask or permission is invalidated. Access to a table that activates column access control and a defined invalid mask on it is blocked until the invalid mask is either disabled, dropped, or re-created (SQLSTATE 560D0). Access to a table that activates row access control and a defined invalid row permission on it is blocked until the invalid permission is either disabled, dropped, or re-created (SQLSTATE 560D0).

column-name
Is the name of the column to be added to the table. The name cannot be qualified. Existing column names or period names in the table cannot be used (SQLSTATE 42711).

data-type
Is one of the data types that are listed under "CREATE TABLE".

NOT NULL
Prevents the column from containing null values. The *default-clause* must also be specified (SQLSTATE 42601).

lob-options
Specifies options for LOB data types. See *lob-options* in "CREATE TABLE".

SCOPE
Specify a scope for a reference type column.

typed-table-name2
The name of a typed table. The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-table-name2* (SQLSTATE 428DM). No checking is done of the default value for *column-name* to ensure that the value references an existing row in *typed-table-name2*.

typed-view-name2
The name of a typed view. The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-view-name2* (SQLSTATE 428DM). No checking is done of the default value for *column-name* to ensure that the values reference an existing row in *typed-view-name2*.

CONSTRAINT *constraint-name*
Names the constraint. A *constraint-name* must not identify a constraint that was already specified within the same ALTER TABLE statement, or as the name of any other existing constraint on the table (SQLSTATE 42710).

If the constraint name is not specified by the user, an 18-byte long identifier unique within the identifiers of the existing constraints that are defined on the table is generated by the system. (The identifier consists of "SQL" followed by a sequence of 15 numeric characters that are generated by a timestamp-based function).

When used with a PRIMARY KEY or UNIQUE constraint:

- the *constraint-name* might be used as the name of an index that is created to support the constraint. See Notes for details on index names that are associated with unique constraints.
- you can use an existing index that is defined with RANDOM columns for the key index if applicable. If there is more than one index that can satisfy the primary or unique requirement,

then the index that is chosen cannot be predicted. An existing primary or unique key index cannot be altered to be an index with random ordering. If a primary or unique key index with random ordering is required, a suitable index must first be defined with the RANDOM keyword. Then, the table must be altered to add the primary or unique key.

PRIMARY KEY
Provides a shorthand method of defining a primary key that is composed of a single column. Thus, if PRIMARY KEY is specified in the definition of column C, the effect is the same as if the PRIMARY KEY(C) clause were specified as a separate clause. The column cannot contain null values, so the NOT NULL attribute must also be specified (SQLSTATE 42831).

See PRIMARY KEY within the *unique-constraint* description.

UNIQUE
Provides a shorthand method of defining a unique key that is composed of a single column. Thus, if UNIQUE is specified in the definition of column C, the effect is the same as if the UNIQUE(C) clause were specified as a separate clause.

See UNIQUE within the *unique-constraint* description.

references-clause
Provides a shorthand method of defining a foreign key that is composed of a single column. Thus, if a references-clause is specified in the definition of column C, the effect is the same as if that references-clause were specified as part of a FOREIGN KEY clause in which C is the only identified column.

See *references-clause* in "CREATE TABLE".

rule-clause
See *rule-clause* in "CREATE TABLE".

CHECK (*check-condition*)
Provides a shorthand method of defining a check constraint that applies to a single column. See *check-condition* in "CREATE TABLE".

constraint-attributes
See *constraint-attributes* in "CREATE TABLE".

default-clause
Specifies a default value for the column.

WITH
An optional keyword.

DEFAULT
Provides a default value when a value is not supplied on INSERT or is specified as DEFAULT on INSERT or UPDATE. If a specific default value is not specified following the DEFAULT keyword, the default value depends on the data type of the column as shown in Table 126 on page 835. If a column is defined as an XML or structured type, then a DEFAULT clause cannot be specified.

If a column is defined that uses a distinct type, then the default value of the column is the default value of the source data type cast to the distinct type.

Table 126. Default Values (when no value specified)

Data Type	Default Value
Numeric	0
Fixed-length character string	A string of blanks
Varying-length character string	A string of length 0
Fixed-length graphic string	A string of double-byte blanks
Varying-length graphic string	A string of length 0

Table 126. Default Values (when no value specified) (continued)

Data Type	Default Value
Fixed-length binary string	A string of hexadecimal zeros
Varying-length binary string	A string of length 0
Date	For existing rows, a date corresponding to January 1, 0001. For added rows, the current date.
Time	For existing rows, a time corresponding to 0 hours, 0 minutes, and 0 seconds. For added rows, the current time.
Timestamp	For existing rows, a date corresponding to 01 January 0001, and a time corresponding to 0 hours, 0 minutes, 0 seconds, and 0 microseconds. For added rows, the current time stamp.
Binary string (blob)	A string of length 0
Boolean	FALSE

Omission of DEFAULT from a *column-definition* results in the use of the null value as the default for the column.

Specific types of values that can be specified with the DEFAULT keyword are as follows.

constant
Specifies the constant as the default value for the column. The specified constant must:

- Represent a value that might be assigned to the column in accordance with the rules of assignment as described in Chapter 3
- Not be a floating-point constant unless the column is defined with a floating-point data type
- Be a numeric constant or a decimal floating-point special value if the data type of the column is decimal floating-point. Floating-point constants are first interpreted as DOUBLE and then converted to decimal floating-point. For DECFLOAT(16) columns, decimal constants must have a precision value less than or equal to 16.
- Not have nonzero digits beyond the scale of the column data type if the constant is a decimal constant (for example, 1.234 cannot be the default for a DECIMAL(5,2) column)
- The constant must be expressed with no more than 254 bytes including the quotation characters, any introducer character such as the X for a hexadecimal constant, and characters from the fully qualified function name and parentheses when the constant is the argument of a *cast-function*.

datetime-special-register
Specifies the value of the datetime special register (CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP) at the time of INSERT, UPDATE, or LOAD as the default for the column. The data type of the column must be the data type that corresponds to the special register specified (for example, data type must be DATE when CURRENT DATE is specified). For existing rows, the value is the current date, current time, or current time stamp when the ALTER TABLE statement is processed.

user-special-register
Specifies the value of the user special register (CURRENT USER, SESSION_USER, SYSTEM_USER) at the time of INSERT, UPDATE, or LOAD as the default for the column. The data type of the column must be a character string with a length not less than the length attribute of a user special register. USER can be specified in place of SESSION_USER and

CURRENT_USER can be specified in place of CURRENT USER. For existing rows, the value is the CURRENT USER, SESSION_USER, or SYSTEM_USER of the ALTER TABLE statement.

CURRENT SCHEMA
Specifies the value of the CURRENT SCHEMA special register at the time of INSERT, UPDATE, or LOAD as the default for the column. If CURRENT SCHEMA is specified, the data type of the column must be a character string with a length greater than or equal to the length attribute of the CURRENT SCHEMA special register. For existing rows, the value of the CURRENT SCHEMA special register at the time the ALTER TABLE statement is processed.

CURRENT MEMBER
Specifies the value of the CURRENT MEMBER special register at the time of INSERT, UPDATE, or LOAD as the default for the column. If CURRENT MEMBER is specified, the data type of the column must allow assignment from an integer. For existing rows, the value of the CURRENT MEMBER special register at the time the ALTER TABLE statement is processed.

NULL
Specifies NULL as the default for the column. If NOT NULL was specified, DEFAULT NULL must not be specified within the same column definition.

cast-function
This form of a default value can be used with columns defined as a distinct type, BLOB, or datetime (DATE, TIME, or TIMESTAMP) data type. For distinct type only, except for distinct types based on BLOB or datetime types, the name of the function must match the name of the distinct type for the column. If qualified with a schema name, it must be the same as the schema name for the distinct type. If not qualified, the schema name from function resolution must be the same as the schema name for the distinct type. For a distinct type based on a datetime type, where the default value is a constant, a function must be used and the name of the function must match the name of the source type of the distinct type with an implicit or explicit schema name of SYSIBM. For other datetime columns, the corresponding datetime function can also be used. For a BLOB or a distinct type based on BLOB, a function must be used and the name of the function must be BLOB with an implicit or explicit schema name of SYSIBM.

constant
Specifies one constant as the argument. The constant must conform to the rules of a constant for the source type of the distinct type or for the data type if not a distinct type. If the *cast-function* is BLOB, the constant must be a string constant.

datetime-special-register
Specifies CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP. The source type of the distinct type of the column must be the data type that corresponds to the specified special register.

user-special-register
Specifies CURRENT USER, SESSION_USER, or SYSTEM_USER. The data type of the source type of the distinct type of the column must be a string data type with a length of at least 8 bytes. If the *cast-function* is BLOB, the length attribute must be at least 8 bytes.

CURRENT SCHEMA
Specifies the value of the CURRENT SCHEMA special register. The data type of the source type of the distinct type of the column must be a character string with a length greater than or equal to the length attribute of the CURRENT SCHEMA special register. If the *cast-function* is BLOB, the length attribute must be at least 8 bytes.

EMPTY_CLOB(), EMPTY_DBCLOB(), or EMPTY_BLOB()
Specifies a zero-length string as the default for the column. The column must have the data type that corresponds to the result data type of the function.

If the value specified is not valid, an error (SQLSTATE 42894) is returned.

generated-clause

Specifies a generated value for the column. This clause must not be specified with *default-clause* in a column definition (SQLSTATE 42623). A generated column cannot be added to a system-period temporal table (SQLSTATE 428HZ). For details on column generation, see "CREATE TABLE".

GENERATED

Specifies that the database manager generates values for the column. GENERATED must be specified whether the column is to be considered an identity column, row change timestamp column, row-begin column, row-end column, transaction start-ID column, or generated expression column.

If the column is nullable, the null value is assigned as the value for the column in existing rows. Otherwise, the value for the column in existing rows depends on the definition of the column:

- ROW CHANGE TIMESTAMP uses a date that corresponds to January 1, 0001 and a time that corresponds to 0 hours, 0 minutes, 0 seconds, and 0 fractional seconds
- ROW BEGIN uses a date that corresponds to January 1, 0001 and a time that corresponds to 0 hours, 0 minutes, 0 seconds, and 0 fractional seconds
- ROW END uses a date that corresponds to December 30, 9999, and a time that corresponds to 0 hours, 0 minutes, 0 seconds, and 0 fractional seconds
- TRANSACTION START ID uses a date that corresponds to January 1, 0001, and a time that corresponds to 0 hours, 0 minutes, 0 seconds, and 0 fractional seconds
- Expressions use the value that is derived from the expression

ALWAYS

Specifies that the database manager always generates a value for the column when a row is inserted or updated and a value must be generated. The result of the expression is stored in the table. GENERATED ALWAYS is the recommended option unless data propagation or unload and reload operations are running. GENERATED ALWAYS is the default for generated columns.

BY DEFAULT

Specifies that the database manager generates a value for the column when a row is inserted into the table, or updated, specifying DEFAULT for the column, unless an explicit value is specified. BY DEFAULT can be specified with *as-row-change-timestamp-clause* only. BY DEFAULT is the recommended option when you use data propagation or running unload and reload operations.

FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP

Specifies that the column is a timestamp column with values that are generated by the database manager. A value is generated for the column in each row that is inserted, and for any row in which any column is updated. The value that is generated for a ROW CHANGE TIMESTAMP column is a timestamp that corresponds to the insert or update time for that row. If multiple rows are inserted or updated with a single statement, the value of the ROW CHANGE TIMESTAMP column might be different for each row.

A table can have one ROW CHANGE TIMESTAMP column (SQLSTATE 428C1) only. If *data-type* is specified, it must be TIMESTAMP or TIMESTAMP(6) (SQLSTATE 42842). A ROW CHANGE TIMESTAMP column cannot have a DEFAULT clause (SQLSTATE 42623). NOT NULL must be specified for a ROW CHANGE TIMESTAMP column (SQLSTATE 42831).

AS (*generation-expression*)

Specifies that the definition of the column is based on an expression. Requires the table to be placed in set integrity pending no access state, by using the SET INTEGRITY statement with the OFF NO ACCESS option. After the ALTER TABLE statement, the SET INTEGRITY statement with the IMMEDIATE CHECKED and FORCE GENERATED options must be used to update and check all the values in that column against the new expression. For details on specifying a column with a *generation-expression*, see "CREATE TABLE".

AS ROW BEGIN

Specifies that the value is assigned by the database manager whenever a row is inserted into the table or any column in the row is updated. The value is generated by using a reading of the time-of-day clock during execution of the first of the following events in the transaction:

- A data change statement that requires a value to be assigned to the row-begin or transaction start-ID column in a table
- A deletion of a row in a system-period temporal table

For a system-period temporal table, the database manager ensures uniqueness of the generated values for a row-begin column across transactions. The timestamp value might be adjusted to ensure that rows that are inserted into an associated history table have the end timestamp value greater than the begin timestamp value (SQLSTATE 01695). This can happen when a conflicting transaction is updating the same row in the system-period temporal table. The database configuration parameter **systime_period_adj** must be set to Yes for this adjustment to the timestamp value to occur otherwise an error is returned (SQLSTATE 57062). If multiple rows are inserted or updated within a single SQL transaction and an adjustment is not needed, the values for the row-begin column are the same for all the rows and are unique from the values that are generated for the column for another transaction. A row-begin column is required as the begin column of a SYSTEM_TIME period, which is the intended use for this type of generated column.

A table can have only one row-begin column (SQLSTATE 428C1). If *data-type* is not specified, the column is defined as a TIMESTAMP(12). If *data-type* is specified, it must be TIMESTAMP(12) (SQLSTATE 42842). The column must be defined as NOT NULL (SQLSTATE 42831). A row-begin column cannot be updated.

AS ROW END

Specifies that a value for the data type of the column is assigned by the database manager whenever a row is inserted or any column in the row is updated. The assigned value is TIMESTAMP '9999-12-30-00.00.00.000000000000'.

A row-end column is required as the second column of a SYSTEM_TIME period, which is the intended use for this type of generated column.

A table can have only one row-end column (SQLSTATE 428C1). If *data-type* is not specified, the column is defined as TIMESTAMP(12). If *data-type* is specified, it must be TIMESTAMP(12) (SQLSTATE 42842). The column must be defined as NOT NULL (SQLSTATE 42831). A row-end column cannot be updated.

AS TRANSACTION START ID

Specifies that the value is assigned by the database manager whenever a row is inserted into the table or any column in the row is updated. The database manager assigns a unique timestamp value per transaction or the null value. The null value is assigned to the transaction start-ID column if the column is nullable and if a row-begin column in the table for which the value did not need to be adjusted exists. Otherwise, the value is generated by using a reading of the time-of-day clock during execution of the first of the following events in the transaction:

- A data change statement that requires a value to be assigned to the row-begin or transaction start-ID column in a table
- A deletion of a row in a system-period temporal table

If multiple rows are inserted or updated within a single SQL transaction, the values for the transaction start-ID column are the same for all the rows and are unique from the values that are generated for the column for another transaction.

A transaction start-ID column is required for a system-period temporal table, which is the intended use for this type of generated column.

A table can have only one transaction start-ID column (SQLSTATE 428C1). If *data-type* is not specified, the column is defined as TIMESTAMP(12). If *data-type* is specified, it must be TIMESTAMP(12). A transaction start-ID column cannot be updated.

COMPRESS SYSTEM DEFAULT
Specifies that system default values (that is, the default values used for the data types when no specific values are specified) are to be stored in minimal space. If the VALUE COMPRESSION clause is not specified, a warning is returned (SQLSTATE 01648) and system default values are not stored in minimal space.

Allowing system default values to be stored in this manner causes a slight performance penalty during insert and update operations on the column because of extra checking that is done.

The base data type must not be a DATE, TIME, TIMESTAMP, XML, or structured data type (SQLSTATE 42842). If the base data type is a varying-length string, this clause is ignored. String values of length 0 are automatically compressed if a table is set with VALUE COMPRESSION.

COLUMN SECURED WITH *security-label-name*
Identifies a security label that exists for the security policy that is associated with the table. The name must not be qualified (SQLSTATE 42601). The table must be associated with a security policy (SQLSTATE 55064). The table must not be a system-period temporal table.

NOT HIDDEN or IMPLICITLY HIDDEN
Specifies whether the column is to be defined as hidden. The hidden attribute determines whether the column is included in an implicit reference to the table, or whether it can be explicitly referenced in SQL statements. The default is NOT HIDDEN.

NOT HIDDEN
Specifies that the column is included in implicit references to the table, and that the column can be explicitly referenced.

IMPLICITLY HIDDEN
Specifies that the column is not visible in SQL statements unless the column is explicitly referenced by name. For example, assuming that a table includes a column that is defined with the IMPLICITLY HIDDEN clause, the result of a SELECT * does not include the implicitly hidden column. However, the result of a SELECT that explicitly refers to the name of an implicitly hidden column includes that column in the result table.

ADD *unique-constraint*
Defines a unique or primary key constraint. A primary key or unique constraint cannot be added to a table that is a subtable (SQLSTATE 429B3). If the table is a supertable at the top of the hierarchy, the constraint applies to the table and all its subtables.

CONSTRAINT *constraint-name*
Names the primary key or unique constraint. For more information, see *constraint-name* in “CREATE TABLE ” on page 1351.

UNIQUE (*column-name*, ... BUSINESS_TIME WITHOUT OVERLAPS)
Defines a unique key that is composed of the identified columns and periods. The identified columns must be defined as NOT NULL. Each *column-name* must identify a column of the table and the same column must not be identified more than once. The name cannot be qualified. The number of identified columns plus two times the number of identified periods must not exceed 64, and the sum of their stored lengths must not exceed the index key length limit for the page size. For column stored lengths, see "Byte Counts" in "CREATE TABLE". For key length limits, see "SQL and XML limits". No LOB, distinct type based on any of these types, or structured type can be used as part of a unique key, even if the length attribute of the column is small enough to fit within the index key length limit for the page size (SQLSTATE 54008). The set of columns in the unique key cannot be the same as the set of columns of the primary key or another unique key (SQLSTATE 01543). If LANGLEVEL is SQL92E or MIA, an error is returned, SQLSTATE 42891. Any existing values in the set of identified columns must be unique (SQLSTATE 23515).

A check is run to determine whether an existing index matches the unique key definition (ignoring any INCLUDE columns in the index). An index definition matches if it identifies the same set of columns without regard to the order of the columns or the direction (ASC/DESC/RANDOM)

specifications. However, for partitioned tables, non-unique partitioned indexes whose columns are not a superset of the table-partitioning key columns are not considered matching indexes.

If a matching index definition is found, the description of the index is changed to indicate that it is required by the system and it is changed to unique after ensuring uniqueness if it was a non-unique index. If the table has more than one matching index, an existing unique index is selected. If multiple unique indexes exist, the selection is arbitrary with one exception:

- For partitioned tables, matching unique partitioned indexes are favored over matching unique nonpartitioned indexes or matching non-unique indexes (partitioned or nonpartitioned).

If no matching index is found, a unique bidirectional index is automatically created for the columns, as described in CREATE TABLE. See Notes for details on index names that are associated with unique constraints.

BUSINESS_TIME WITHOUT OVERLAPS
For a constraint, BUSINESS_TIME indicates the period name in this table. The period must exist in the table (SQLSTATE 42727).

BUSINESS_TIME WITHOUT OVERLAPS specifies that overlapping periods for BUSINESS_TIME are not allowed, and that values for the rest of the keys must be unique regarding any period of BUSINESS_TIME. When BUSINESS_TIME WITHOUT OVERLAPS is specified, the end column and begin column of the period BUSINESS_TIME (in this order of the columns) is automatically added to the index key in ascending order and enforce that no overlaps in time exist. The columns that are used to defined BUSINESS_TIME must not be specified as part of the constraint (SQLSTATE 428HW).

When a partition is attached to a range partitioned application-period temporal table that has a partitioned BUSINESS_TIME WITHOUT OVERLAPS index, the source table must have an index that matches the partitioned BUSINESS_TIME WITHOUT OVERLAPS index. Additionally, the PERIODNAME and PERIODPOLICY attributes on the indexes must also match.

PRIMARY KEY (*column-name*, ... BUSINESS_TIME WITHOUT OVERLAPS)
Defines a primary key that is composed of the identified columns. Each *column-name* must identify a column of the table, and the same column must not be identified more than once. The name cannot be qualified. The number of identified columns must not exceed 64, and the sum of their stored lengths must not exceed the index key length limit for the page size. For column stored lengths, see "Byte Counts" in "CREATE TABLE". For key length limits, see "SQL limits". The table must not have a primary key and the identified columns must be defined as NOT NULL. No LOB, distinct type based on any of these types, or structured type can be used as part of a primary key, even if the length attribute of the column is small enough to fit within the index key length limit for the page size (SQLSTATE 54008). The set of columns in the primary key cannot be the same as the set of columns in a unique key (SQLSTATE 01543). (If LANGLEVEL is SQL92E or MIA, an error is returned, SQLSTATE 42891). Any existing values in the set of identified columns must be unique (SQLSTATE 23515). *column-name* must not be the name of a row change timestamp, or a begin or end column of the period (SQLSTATE 428HW).

A check runs to determine whether an existing index matches the primary key definition (ignoring any INCLUDE columns in the index). An index definition matches if it identifies the same set of columns without regard to the order of the columns or the direction (ASC/DESC/RANDOM) specifications. However, for partitioned tables, non-unique partitioned indexes whose columns are not a superset of the table-partitioning key columns are not considered matching indexes.

If a matching index definition is found, the description of the index is changed to indicate that it is the primary index, as required by the system, and it is changed to unique after uniqueness is ensured if it was a non-unique index. If the table has more than one matching index, an existing unique index is selected. If multiple unique indexes exist, the selection is arbitrary with one exception:

- For partitioned tables, matching unique partitioned indexes are favored over matching unique nonpartitioned indexes or matching non-unique indexes (partitioned or nonpartitioned).

If no matching index is found, a unique bidirectional index is automatically created for the columns, as described in CREATE TABLE. See Notes for details on index names that are associated with unique constraints.

If the primary key is being added to a shadow table, the columns of the primary key must match the columns of an enforced primary key constraint or an enforced unique constraint of the base table that is referenced in the *fullselect* of *materizalized-query-definition*.

A primary key cannot be created on a materialized query table that is not defined with MAINTAINED BY REPLICATION.

Only one primary key can be defined on a table.

BUSINESS_TIME WITHOUT OVERLAPS

For a constraint, BUSINESS_TIME indicates the period name in this table. The period must exist in the table (SQLSTATE 42727).

BUSINESS_TIME WITHOUT OVERLAPS specifies that overlapping periods for BUSINESS_TIME are not allowed, and that values for the rest of the keys must be unique regarding any period of BUSINESS_TIME. When BUSINESS_TIME WITHOUT OVERLAPS is specified, the end column and begin column of the period BUSINESS_TIME (in this order of the columns) is automatically added to the index key in ascending order and enforce that no overlaps in time exist. The columns that are used to defined BUSINESS_TIME must not be specified as part of the constraint (SQLSTATE 428HW).

When a partition is attached to a range partitioned application-period temporal table that has a partitioned BUSINESS_TIME WITHOUT OVERLAPS index, the source table must have an index that matches the partitioned BUSINESS_TIME WITHOUT OVERLAPS index. Additionally, the PERIODNAME and PERIODPOLICY attributes on the indexes must also match.

constraint-attributes

See *constraint-attributes* in "CREATE TABLE".

ADD *referential-constraint*

Defines a referential constraint. See *referential-constraint* in "CREATE TABLE".

ADD *check-constraint*

Defines a check constraint or functional dependency. See *check-constraint* in "CREATE TABLE".

constraint-attributes

See *constraint-attributes* in "CREATE TABLE".

ADD *distribution-clause*

Defines a distribution key. The table must be defined in a table space on a single-partition database partition group (SQLSTATE 55037) and must not already have a distribution key (SQLSTATE 42889). If a distribution key exists for the table, the existing key must be dropped before you add the new distribution key. A distribution key cannot be added to a table that is a subtable (SQLSTATE 428DH) .

DISTRIBUTE BY HASH (*column-name*...)

Defines a distribution key by using the specified columns. Each *column-name* must identify a column of the table, and the same column must not be identified more than once. The name cannot be qualified. A column cannot be used as part of a distribution key if the data type of the column is a BLOB, CLOB, DBCLOB, XML, distinct type on any of these types, or structured type.

ADD RESTRICT ON DROP

Specifies that the table cannot be dropped, and that the table space that contains the table cannot be dropped.

ADD MATERIALIZED QUERY

materialized-query-definition

Changes a regular table to a materialized query table for use during query optimization. The table that is specified by *table-name* must not:

- Be previously defined as a materialized query table
- Be a typed table

- Have any constraints, unique indexes, or triggers defined
- Reference a nickname that is marked with caching disabled
- Be referenced in the definition of another materialized query table
- Be referenced in the definition of a view that is enabled for query optimization

If *table-name* does not meet these criteria, an error is returned (SQLSTATE 428EW).

If row level or column level access control is activated for any table that is directly or indirectly referenced in the *fullselect* of *materizalized-query-definition*, and row level access control is not activated for the altered table, row level access control is implicitly activated for the altered table. This restricts direct access to the contents of the materialized query table. A query that explicitly references the table before such a row permission is defined returns a warning that no data in the table exists (SQLSTATE 02000). To provide access to the materialized query table, an appropriate row permission can be created, or an **ALTER TABLE DEACTIVATE ROW ACCESS CONTROL** on the materialized query table can be entered to remove the row level protection if that is appropriate.

If the materialized query table references any table that has row level or column level access control that is activated, the functions that are referenced in the *fullselect* of *materizalized-query-definition* must be defined with the SECURED attribute (SQLSTATE 428EC).

If the table that is altered to a materialized query table has any permissions (excluding the system generated default permission) or masks defined on it, ALTER fails (SQLSTATE 428EW).

fullselect
Defines the query in which the table is based. The columns of the existing table must:

- have the same number of columns
- have the same data types
- have the same column names in the same ordinal positions

As the result columns of *fullselect* (SQLSTATE 428EW). For information about specifying the *fullselect* for a materialized query table, see "CREATE TABLE". One extra restriction is that *table-name* cannot be directly or indirectly referenced in the fullselect.

refreshable-table-options
Specifies the refreshable options for altering a materialized query table.

DATA INITIALLY DEFERRED
The data in the table must be validated by using the REFRESH TABLE or SET INTEGRITY statement.

REFRESH
Indicates how the data in the table is maintained.

DEFERRED
The data in the table can be refreshed at any time by using the REFRESH TABLE statement. The data in the table reflects the result of the query as a snapshot at the time the REFRESH TABLE statement is processed only. Materialized query tables that are defined with this attribute do not allow INSERT, UPDATE, or DELETE statements (SQLSTATE 42807).

IMMEDIATE
The changes that are made to the underlying tables as part of a DELETE, INSERT, or UPDATE are cascaded to the materialized query table. In this case, the content of the table, at any point-in-time, is the same as if the specified subselect is processed. Materialized query tables (MQTs) defined with this attribute do not allow INSERT, UPDATE, or DELETE statements (SQLSTATE 42807). Column-organized MQTs that use the REFRESH IMMEDIATE option are not supported when the MAINTAINED BY SYSTEM clause is specified (SQL20058N).

ENABLE QUERY OPTIMIZATION
The materialized query table can be used for query optimization.

DISABLE QUERY OPTIMIZATION
The materialized query table is not used for query optimization. The table can still be queried directly.

MAINTAINED BY
Specifies whether the data in the materialized query table is maintained by the system, user, or replication tool.

SYSTEM
Specifies that the data in the materialized query table is maintained by the system.

USER
Specifies that the data in the materialized query table is maintained by the user. The user is allowed to run update, delete, or insert operations against user-maintained materialized query tables. The REFRESH TABLE statement, which is used for system-maintained materialized query tables, cannot be started against user-maintained materialized query tables. Only a REFRESH DEFERRED materialized query table can be defined as MAINTAINED BY USER.

REPLICATION
Specifies that the data in the materialized query table is maintained by an external replication technology. The REFRESH TABLE statement, which is used for system-maintained materialized query tables, cannot be issued against replication-maintained materialized query tables, which are referred to as *shadow tables*. Only a REFRESH DEFERRED materialized query table can be defined as MAINTAINED BY REPLICATION, and the altered table must be a column-organized table.

FEDERATED_TOOL
Specifies that the data in the materialized query table is maintained by a federated replication tool. The REFRESH TABLE statement, which is used for system-maintained materialized query tables, cannot be started against federated_tool-maintained materialized query tables. Only a REFRESH DEFERRED materialized query table can be defined as MAINTAINED BY FEDERATED_TOOL.

ALTER FOREIGN KEY *constraint-name*
Alters the constraint attributes of the referential constraint *constraint-name*. The *constraint-name* must identify an existing referential constraint (SQLSTATE 42704).

ALTER CHECK *constraint-name*
Alters the constraint attributes of the check constraint or functional dependency *constraint-name*. The *constraint-name* must identify an existing check constraint or functional dependency (SQLSTATE 42704).

constraint-alteration
Options for changing attributes that are associated with referential or check constraints.

ENABLE QUERY OPTIMIZATION or DISABLE QUERY OPTIMIZATION
Specifies whether the constraint or functional dependency can be used for query optimization under appropriate circumstances.

ENABLE QUERY OPTIMIZATION
The constraint is assumed to be true and can be used for query optimization.

DISABLE QUERY OPTIMIZATION
The constraint cannot be used for query optimization.

ENFORCED or NOT ENFORCED
Specifies whether the constraint is enforced by the database manager during normal operations such as insert, update, or delete. A foreign key constraint cannot be altered from NOT ENFORCED to ENFORCED if the parent key is not enforced (SQLSTATE 42888).

ENFORCED
Change the constraint to ENFORCED. ENFORCED cannot be specified for a functional dependency (SQLSTATE 42621).

NOT ENFORCED
Change the constraint to NOT ENFORCED.

TRUSTED
The data can be trusted to conform to the constraint. TRUSTED must be used only if the data in the table is independently known to conform to the constraint. Query results might be unpredictable if the data does not conform to the constraint. This is the default option.

Informational constraints must not be violated at any time. Informational constraints are used in query optimization, and the incremental processing of REFRESH IMMEDIATE MQT and staging tables. These processes might produce unpredictable results or incorrect MQT and staging table content if the constraints are violated. For example, the order in which parent-child tables are maintained is important. When you want to add rows to a parent-child table, you must insert rows into the parent table first. To remove rows from a parent-child table, you must delete rows from the child table first. This ensures that no orphan rows in the child table exist at any time. If informational constraints are violated, the incremental maintenance of dependent MQT data and staging table data might be optimized based on the violated informational constraints, producing incorrect data.

NOT TRUSTED
The data cannot be trusted to conform to the constraint. NOT TRUSTED is intended for cases where the data conforms to the constraint for most rows, but it is not independently known that all the rows or future additions conform to the constraint. If a constraint is NOT TRUSTED and enabled for query optimization, then it is not used to run optimizations that depend on the data that conforms completely to the constraint. NOT TRUSTED can be specified only for foreign keys (SQLSTATE 42601).

ALTER *column-alteration*
Alters the definition of a column. Only the specified attributes are altered; others remain unchanged. Columns of a typed table cannot be altered (SQLSTATE 428DH). The table must not be defined as a history table (SQLSTATE 428FR). Columns that are used in expression-based index keys cannot be altered (SQLSTATE 42893), unless the operation involves the following column attributes:

- Identity (by using clauses under *identity-alteration*)
- Default compression (by using COMPRESS clause)
- Security (by using SECURED WITH or DROP COLUMN SECURITY clauses)

column-name
Specifies the name of the column that is to be altered. The *column-name* must identify an existing column of the table (SQLSTATE 42703). The name must not be qualified. The name must not identify a column that is otherwise being added, altered, or dropped in the same ALTER TABLE statement (SQLSTATE 42711).

SET DATA TYPE *altered-data-type*
Specifies the new data type of the column. The new data type must be castable from the existing data type of the column (SQLSTATE 42837) except when one of the data types is a distinct type, in which case the source data type of the distinct type is used in determining whether the data types are castable. A LOB column cannot be altered to a different data type (SQLSTATE 42837). A non-LOB column cannot be altered to a LOB data type (SQLSTATE 42837).

Altering a column to a character or graphic string data type that results in the truncation of non-blank characters from existing data is not allowed (SQLSTATE 42837). Similarly, altering a column to a binary string data type that results in truncation of bytes other than hexadecimal zeros is not allowed.

Data type alterations require a classic table reorganization before the table can be fully accessed (SQLSTATE 57007), except in the following situations:

- Increasing the length of a VARCHAR, VARGRAPHIC, or VARBINARY column
- Decreasing the length of a VARCHAR, VARGRAPHIC, or VARBINARY column without truncating trailing blanks from existing data, when no indexes exist on the column

The administrative routine SYSPROC.ADMIN_REVALIDATE_DB_OBJECTS can be called to do table reorganization. A data type alteration that requires a table reorganization cannot be specified if the table is in SET INTEGRITY PENDING state (SQLSTATE 57007).

A string data type cannot be altered if the column is a column of a table-partitioning key.

If the column is a column of a distribution key, then the new data type must meet the following requirements (SQLSTATE 42997):

- Be the same data type as the current column type
- Have the same length of the current column type, except when increasing column length of VARCHAR, VARGRAPHIC, and VARBINARY data type columns
- Cannot be modified to FOR BIT DATA or vice versa in the cases of CHAR and VARCHAR data types

If the data type is LOB, the specified length attribute cannot allow for the possibility of any truncated data (SQLSTATE 42837).

The data type of an identity column cannot be altered (SQLSTATE 42997).

The data type of a column that is defined as ROW BEGIN, ROW END, or TRANSACTION START ID cannot be altered (SQLSTATE 428FR).

The data type and nullability of BUSINESS_TIME period columns cannot be altered (SQLSTATE 428FR).

The table cannot have data capture enabled (SQLSTATE 42997).

The data type of a column cannot be altered if any of the following conditions are true (SQLSTATE 42893):

- The column is a generated expression column and the data of the generated expression column changes if the column is altered
- The column is referenced in an expression of a generated expression column and the data of the generated expression column changes if the column is altered
- The column is referenced in a check constraint and the check constraint is not satisfied if the column is altered
- The column is used in a referential integrity constraint and the referential integrity constraint is not satisfied if the column is altered

Altering a column must not cause the total byte count of all columns to exceed the maximum record size (SQLSTATE 54010). If the column is used in a unique constraint or an index, the new length must not cause the sum of the stored lengths for the unique constraint or index to exceed the index key length limit for the page size (SQLSTATE 54008). For column stored lengths, see "Byte Counts" in "CREATE TABLE". For key length limits, see "SQL and XML limits".

If **auto_reval** is set to DISABLED, the cascaded effects of altering a column is shown in Table 127 on page 846.

If either a row permission or a column mask depends on the column that is altered (as recorded in the SYSCAT.CONTROLDEP catalog view), an error is returned (SQLSTATE 42917).

Table 127. Cascaded effects of altering a column

Operation	Effect
Altering a column that is referenced by a view or check constraint	The object is regenerated during alter processing. If a view, function or method resolution for the object is different after the alter operation, changes the semantics of the object. In the case of a check constraint, if the semantics of the object change as a result of the alter operation, the operation fails.

Table 127. Cascaded effects of altering a column (continued)

Operation	Effect
Altering a column in a table that has a dependent package, trigger, or SQL routine	The object is marked invalid, and is re-validated on next use.
Altering the type of a column in a table that is referenced by an XSROBJECT enabled for decomposition	The XSROBJECT is marked inoperative for decomposition. Reenabling the XSROBJECT might require readjustment of its mappings; afterward, issue an ALTER XSROBJECT ENABLE DECOMPOSITION statement against the XSROBJECT.
Altering a column that is referenced in the default expression of a global variable	The default expression of the global variable is validated during alter processing. If a user-defined function that is used in the default expression cannot be resolved, the operation fails.

If the table is a system-period temporal table, the column is also changed in any associated history table. If the table is a system-period temporal table, string data type columns cannot be altered to a length that requires data truncation, and numeric data type columns cannot be altered to lower precision data types (SQLSTATE 42837).

built-in-type
See "CREATE TABLE" for the description of built-in data types.

SET NOT NULL
Specifies that the column cannot contain null values. No value for this column in existing rows of the table can be the null value (SQLSTATE 23502). This clause is not allowed if the column is specified in the foreign key of a referential constraint with a DELETE rule of SET NULL, and no other nullable columns exist in the foreign key (SQLSTATE 42831).

Altering this attribute for a column requires a classic table reorganization before full table access is allowed (SQLSTATE 57007).

If a row permission or column mask exists, which depends on the column to be altered, an error is issued (SQLSTATE 42917).

If the table is a system-period temporal table, the column is also changed in any associated history table.

SET INLINE LENGTH *integer*
Changes the inline length of an existing structured type, XML, or LOB data type column. The inline length indicates the maximum size in bytes of an instance of a structured type, XML, or LOB data type to store in the base table row. Instances of a structured type or XML data type that cannot be stored inline in the base table row are stored separately, similar to the way that LOB values are stored.

The data type of *column-name* must be a structured type, XML, or LOB data type (SQLSTATE 42842).

The default inline length for a structured type column is the inline length of its data type (specified explicitly or by default in the CREATE TYPE statement). If the inline length of a structured type is less than 292, the value 292 is used for the inline length of the column.

The explicit inline length value can be increased only (SQLSTATE 429B2); it cannot exceed 32673 (SQLSTATE 54010). For a structured type or XML data type column, it must be at least 292. For a LOB data type column, the INLINE LENGTH must not be less than the maximum LOB descriptor size.

Altering the column must not cause the total byte count of all columns to exceed the row size limit (SQLSTATE 54010).

Data that is already stored separately from the rest of the row is not moved inline into the base table row by this statement.

To take advantage of the altered inline length of a structured type column, start the REORG command against the specified table after altering the inline length of its column.

To take advantage of the altered inline length of an XML data type column in an existing table, update all rows with an UPDATE statement.

The REORG command has no effect on the row storage of XML documents.

To take advantage of the altered inline length of a LOB data type column, use the REORG command with the LONGLOBDATA option or UPDATE the corresponding LOB column.

For example:

```
UPDATE table-name SET lob-column = lob-column
   WHERE LENGTH(lob-column) <= chosen-inline-length - 4
```

where *table-name* is the table that had the inline length of the LOB data type column that is altered, *lob-column* is the LOB data type column that was altered, and *chosen-inline-length* is the new value that was chosen for the INLINE LENGTH.

If a row permission or column mask exists, which depends on the column to be altered, an error is returned (SQLSTATE 42917).

If the table is a system-period temporal table, inline length changes are propagated to the history table.

SET *default-clause*
Specifies a new default value for the column that is to be altered. The column must not already be defined as a generated column (SQLSTATE 42623). The specified default value must represent a value that could be assigned to the column in accordance with the rules for assignment as described in "Assignments and comparisons". Altering the default value does not change the value that is associated with this column for existing rows.

SET EXPRESSION AS (*generation-expression*)
Changes the expression for the column to the specified *generation-expression*. SET EXPRESSION requires the table to be put in set integrity pending state by using the SET INTEGRITY statement with the OFF option. After the ALTER TABLE statement, the SET INTEGRITY statement with the IMMEDIATE CHECKED and FORCE GENERATED options must be used to update and check all the values in that column against the new expression. The column must already be defined as a generated column based on an expression (SQLSTATE 42837), and must not have appeared in the PARTITIONING KEY, DIMENSIONS, or KEY SEQUENCE clauses of the table (SQLSTATE 42997). The generation-expression must conform to the same rules that apply when defining a generated column. The result data type of the generation-expression must be assignable to the data type of the column (SQLSTATE 42821).

The *generation-expression* must not reference a column for which a column mask is defined (SQLSTATE 42621).

SET NOT HIDDEN or SET IMPLICITLY HIDDEN
Specifies the hidden attribute for the column.

If the table is a system-period temporal table, the column is also changed in any associated history table.

NOT HIDDEN
Specifies that the column is included in implicit references to the table, and that the column can be explicitly referenced.

IMPLICITLY HIDDEN
Specifies that the column is not visible in SQL statements unless the column is explicitly referenced by name. For example, assuming that a table includes a column that is defined with the IMPLICITLY HIDDEN clause, the result of a SELECT * does not include the implicitly

hidden column. However, the result of a SELECT that explicitly refers to the name of an implicitly hidden column includes that column in the result table.

IMPLICITLY HIDDEN must not be specified for the last column of the table that is not hidden (SQLSTATE 428GU).

SET *generation-alteration*
Specifies that the generation attribute for the column is to be changed. GENERATED might be specified whether the column is an identity column or a row change timestamp column (SQLSTATE 42837). If the table is a system-period temporal table, the column in the associated history table is not affected by the change. If a default for the column exists, that default must be dropped, which can be done in the same *column-alteration* by using one of the DROP DEFAULT clauses. SET GENERATED must not be specified for a column of a temporal history table (SQLSTATE 428FR).

GENERATED ALWAYS
Specifies that the database manager always generates a value for the column when a row is inserted or updated and a value must be generated. GENERATED ALWAYS is the recommended option unless data propagation or unload and reload operations are being performed. ALWAYS is the default for generated columns.

GENERATED BY DEFAULT
Specifies that the database manager generates a value for the column when a row is inserted into the table, or updated, specifying DEFAULT for the column, unless an explicit value is specified. GENERATED BY DEFAULT can be specified with *as-row-change-timestamp-clause* only. GENERATED BY DEFAULT is the recommended option when using data propagation or performing unload and reload operations.

identity-alteration
Alters the identity attributes of the column. The column must be an identity column.

SET INCREMENT BY *numeric-constant*
Specifies the interval between consecutive values of the identity column. The next value to be generated for the identity column will be determined from the last assigned value with the increment applied. The column must already be defined with the IDENTITY attribute (SQLSTATE 42837).

This value can be any positive or negative value that might be assigned to this column (SQLSTATE 42815), and does not exceed the value of a large integer constant (SQLSTATE 42820), without nonzero digits that exist to the right of the decimal point (SQLSTATE 428FA).

If this value is negative, this is a descending sequence after the ALTER statement. If this value is 0 or positive, this is an ascending sequence after the ALTER statement.

SET NO MINVALUE or MINVALUE *numeric-constant*
Specifies the minimum value at which a descending identity column either cycles or stops generating values, or the value to which an ascending identity column cycle after it reaches the maximum value. The column must exist in the specified table (SQLSTATE 42703), and must already be defined with the IDENTITY attribute (SQLSTATE 42837).

NO MINVALUE
For an ascending sequence, the value is the original starting value. For a descending sequence, the value is the minimum value of the data type of the column.

MINVALUE *numeric-constant*
Specifies the numeric constant that is the minimum value. This value can be any positive or negative value that might be assigned to this column (SQLSTATE 42815), without nonzero digits that exist to the right of the decimal point (SQLSTATE 428FA), but the value must be less than or equal to the maximum value (SQLSTATE 42815).

SET NO MAXVALUE or MAXVALUE *numeric-constant*
Specifies the maximum value at which an ascending identity column either cycles or stops generating values, or the value to which a descending identity column cycle after it reaches the minimum value. The column must exist in the specified table (SQLSTATE 42703), and must already be defined with the IDENTITY attribute (SQLSTATE 42837).

NO MAXVALUE
For an ascending sequence, the value is the maximum value of the data type of the column. For a descending sequence, the value is the original starting value.

MAXVALUE *numeric-constant*
Specifies the numeric constant that is the maximum value. This value can be any positive or negative value that might be assigned to this column (SQLSTATE 42815), without nonzero digits that exist to the right of the decimal point (SQLSTATE 428FA), but the value must be greater than or equal to the minimum value (SQLSTATE 42815).

SET NO CYCLE or CYCLE
Specifies whether this identity column continues to generate values after generating either its maximum or minimum value. The column must exist in the specified table (SQLSTATE 42703), and must already be defined with the IDENTITY attribute (SQLSTATE 42837).

NO CYCLE
Specifies that values are not generated for the identity column after the maximum or minimum value is reached.

CYCLE
Specifies that values continue to be generated for this column after the maximum or minimum value is reached. If this option is used, then after an ascending identity column reaches the maximum value, it generates its minimum value; or after a descending sequence reaches the minimum value, it generates its maximum value. The maximum and minimum values for the identity column determine the range that is used for cycling.

When CYCLE is in effect, duplicate values can be generated for an identity column. Although not required, if unique values are wanted, a single-column unique index that is defined by using the identity column ensures uniqueness. If a unique index exists on such an identity column and a non-unique value is generated, an error occurs (SQLSTATE 23505).

SET NO CACHE or CACHE *integer-constant*
Specifies whether to keep some pre-allocated values in memory for faster access. This is a performance and tuning option. The column must already be defined with the IDENTITY attribute (SQLSTATE 42837).

NO CACHE
Specifies that values for the identity column are not to be pre-allocated.

In a Db2 pureScale environment, if the identity values *must* be generated in order of request, the NO CACHE option must be used.

When this option is specified, the values of the identity column are not stored in the cache. In this case, every request for a new identity value results in synchronous I/O to the log.

CACHE *integer-constant*
Specifies how many values of the identity sequence are pre-allocated and kept in memory. When values are generated for the identity column, pre-allocating and storing values in the cache reduces synchronous I/O to the log.

If a new value is needed for the identity column and no unused values available in the cache exist, the allocation of the value requires waiting for I/O to the log. However, when a new value is needed for the identity column and an unused value in the cache exist, the allocation of that identity value can happen more quickly by avoiding the I/O to the log.

If a database deactivates, either normally or due to a system failure, all cached sequence values that are not used in committed statements are lost (that is, they are never used). The maximum number of identity column values that can be lost is calculated as follows:

- If ORDER is specified, the maximum is the value that is specified for the CACHE option.
- In a multi-partition or Db2 pureScale, the maximum is the value that is specified for the CACHE option times the number of members that generate new identity values.

The minimum value is 2 (SQLSTATE 42815).

In a Db2 pureScale environment, if both CACHE and ORDER are specified, the specification of ORDER overrides the specification of CACHE and instead NO CACHE is in effect.

SET NO ORDER or ORDER
Specifies whether the identity column values must be generated in order of request. The column must exist in the specified table (SQLSTATE 42703), and must already be defined with the IDENTITY attribute (SQLSTATE 42837).

NO ORDER
Specifies that the identity column values do not need to be generated in order of request.

ORDER
Specifies that the identity column values must be generated in order of request.

RESTART or RESTART WITH *numeric-constant*
Resets the state of the sequence that is associated with the identity column. If WITH *numeric-constant* is not specified, the sequence for the identity column is restarted at the value that was specified, either implicitly or explicitly, as the starting value when the identity column was originally created.

The column must exist in the specified table (SQLSTATE 42703), and must already be defined with the IDENTITY attribute (SQLSTATE 42837). RESTART does *not* change the original START WITH value.

The *numeric-constant* is an exact numeric constant that can be any positive or negative value that might be assigned to this column (SQLSTATE 42815), without nonzero digits that exist to the right of the decimal point (SQLSTATE 428FA). The *numeric-constant* will be used as the next value for the column.

SET *generation-attribute as-identity-clause*
Changes the column to an identity column. This column alteration must not be specified whether the column has a default or is already a generated column (SQLSTATE 42837). If the table is a system-period temporal table, the column in the associated history table is not affected by the change.

GENERATED ALWAYS
Specifies that the database manager always generates a value for the column when a row is inserted or updated and a value must be generated. ALWAYS is the default for generated columns.

GENERATED BY DEFAULT
Specifies that the database manager generates a value for the column when a row is inserted or updated and a default value must be generated, unless an explicit value is specified.

as-identity-clause
Specifies that the column is the identity column for the table. A table can have a single identity column (SQLSTATE 428C1). The column must be specified as not nullable (SQLSTATE 42997) only, and the data type associated with the column must be an exact numeric data type with a scale of zero (SQLSTATE 42815). An exact numeric data type is one of: SMALLINT, INTEGER, BIGINT, DECIMAL, or NUMERIC with a scale of zero, or a distinct type based on one of these types. For details on identity options, see "CREATE TABLE".

SET GENERATED ALWAYS
Changes the column to a generated expression column, a row-begin column, a row-end column, or a transaction-start-ID column. GENERATED ALWAYS specifies that the database manager always generates a value for the column when a row is inserted or updated and a value must be generated.

AS (*generation-expression*)
Specifies that the definition of the column is based on an expression. The column must not already be defined with a generation expression, cannot be the identity column, or cannot have an explicit default (SQLSTATE 42837). The *generation-expression* must conform to the same rules that apply when defining a generated column. The result data type of the *generation-expression* must be assignable to the data type of the column (SQLSTATE 42821).

The column must not be referenced in the distribution key column or in the multidimensional clustering (MDC) key (SQLSTATE 42997).

The *generation-expression* must not reference a column for which a column mask is defined (SQLSTATE 42621).

AS ROW BEGIN

Specifies that the value is assigned by the database manager whenever a row is inserted into the table or any column in the row is updated. The value is generated by using a reading of the time-of-day clock during execution of the first of the following events in the transaction:

- A data change statement that requires a value to be assigned to the row-begin or transaction start-ID column in a table
- A deletion of a row in a system-period temporal table

For a system-period temporal table, the database manager ensures uniqueness of the generated values for a row-begin column across transactions. The timestamp value might be adjusted to ensure that rows that are inserted into an associated history table have the end timestamp value greater than the begin timestamp value (SQLSTATE 01695). This can happen when a conflicting transaction is updating the same row in the system-period temporal table. The database configuration parameter **systime_period_adj** must be set to Yes for this adjustment to the timestamp value to occur otherwise an error is returned (SQLSTATE 57062). If multiple rows are inserted or updated within a single SQL transaction and an adjustment is not needed, the values for the row-begin column are the same for all the rows and are unique from the values that are generated for the column for another transaction. A row-begin column is required as the begin column of a SYSTEM_TIME period, which is the intended use for this type of generated column.

A table can have only one row-begin column (SQLSTATE 428C1). If *data-type* is not specified, the column is defined as a TIMESTAMP(12). If *data-type* is specified, it must be TIMESTAMP(12) (SQLSTATE 42842). The column must be defined as NOT NULL (SQLSTATE 42831). A row-begin column cannot be updated.

AS ROW END

Specifies that the maximum value for the data type of the column is assigned by the database manager whenever a row is inserted or any column in the row is updated.

A row-end column is required as the second column of a SYSTEM_TIME period, which is the intended use for this type of generated column.

A table can have only one row-end column (SQLSTATE 428C1). If *data-type* is not specified, the column is defined as TIMESTAMP(12). If *data-type* is specified, it must be TIMESTAMP(12) (SQLSTATE 42842). The column must be defined as NOT NULL (SQLSTATE 42831). A row-end column cannot be updated.

AS TRANSACTION START ID

Specifies that the value is assigned by the database manager whenever a row is inserted into the table or any column in the row is updated. The database manager assigns a unique timestamp value per transaction or the null value. The null value is assigned to the transaction start-ID column if the column is nullable and if a row-begin column exists in the table for which the value did not need to be adjusted. Otherwise, the value is generated by using a reading of the time-of-day clock during execution of the first of the following events in the transaction:

- A data change statement that requires a value to be assigned to the row-begin or transaction start-ID column in a table
- A deletion of a row in a system-period temporal table

If multiple rows are inserted or updated within a single SQL transaction, the values for the transaction start-ID column are the same for all the rows and are unique from the values that are generated for the column for another transaction.

A transaction start-ID column is required for a system-period temporal table, which is the intended use for this type of generated column.

A table can have only one transaction start-ID column (SQLSTATE 428C1). If *data-type* is not specified, the column is defined as TIMESTAMP(12). If *data-type* is specified, it must be TIMESTAMP(12). A transaction start-ID column cannot be updated.

DROP DEFAULT
Drops the current default for the column. The specified column must have a default value (SQLSTATE 42837). This action is propagated to the history table for a system-period temporal table.

DROP GENERATED
Drops the generated attributes of the column. The column must be defined as a generated column (SQLSTATE 42837). The column must not be defined as a row-begin column, row-end column, or a transaction-start-ID column in a system-period temporal table (SQLSTATE 428FR).

DROP NOT NULL
Drops the NOT NULL attribute of the column, allowing the column to have the null value. This clause is not allowed if the column is specified in the primary key, in a unique constraint of the table (SQLSTATE 42831), a row-begin column, or a row-end column (SQLSTATE 42837).

Altering this attribute for a column requires a classic table reorganization before full table access is allowed (SQLSTATE 57007).

The table cannot have data capture enabled (SQLSTATE 42997). DROP NOT NULL is blocked for columns that belong to the BUSINESS_TIME period (SQLSTATE 428FR).

If the table is a system-period temporal table, the NOT NULL attribute is also dropped from the corresponding column in any associated history table.

If either a row permission or column mask exists, which depends on the column to be altered, an error is issued (SQLSTATE 42917).

ADD SCOPE
Add a scope to an existing reference type column that does not already define a scope that is defined (SQLSTATE 428DK). If the altered table is a typed table, the column must not be inherited from a supertable (SQLSTATE 428DJ).

typed-table-name
The name of a typed table. The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-table-name* (SQLSTATE 428DM). No checking is done of any existing values in *column-name* to ensure that the values reference existing rows in *typed-table-name*.

typed-view-name
The name of a typed view. The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-view-name* (SQLSTATE 428DM). No checking is done of any existing values in *column-name* to ensure that the values reference existing rows in *typed-view-name*.

COMPRESS
Specifies whether default values for this column are to be stored more efficiently.

SYSTEM DEFAULT
Specifies that system default values (that is, the default values used for the data types when no specific values are specified) are to be stored by using minimal space. If the table is not already set with the VALUE COMPRESSION attribute activated, a warning is returned (SQLSTATE 01648), and system default values are not stored by using minimal space.

Allowing system default values to be stored in this manner causes a slight performance penalty during insert and update operations on the column because of the extra checking that is done.

Existing data in the column is not changed. Consider offline table reorganization to enable existing data to take advantage of storing system default values by using minimal space.

OFF
Specifies that system default values are to be stored in the column as regular values. Existing data in the column is not changed. Offline reorganization is recommended to change existing data.

The base data type must not be DATE, TIME, or TIMESTAMP (SQLSTATE 42842). If the base data type is a varying-length string, this clause is ignored. String values of length 0 are automatically compressed if a table was set with VALUE COMPRESSION.

If the altered table is a typed table, the column must not be inherited from a supertable (SQLSTATE 428DJ).

SECURED WITH *security-label-name*
Identifies a security label that exists for the security policy that is associated with the table. The name must not be qualified (SQLSTATE 42601). The table must be associated with a security (SQLSTATE 55064). The table must not be a system-period temporal table.

DROP COLUMN SECURITY
Alters a column to make it a non-protected column.

ACTIVATE ROW ACCESS CONTROL

Activates row level access control on the table. The table must not be a typed table, a catalog table (SQLSTATE 55019), a created temporary table, a declared temporary table (SQLSTATE 42995), a nickname (SQLSTATE 42809), a view (SQLSTATE 42809), or an external table (SQLSTATE 42858). The table must not identify a shadow table or a base table of a shadow table (SQLSTATE 428HZ).

A default row permission is implicitly created and allows no access to any rows of the table, unless permitted by a row permission that is explicitly created by a user with SECADM authority.

When the table is referenced in a data manipulation statement, all enabled row permissions that were created for the table, including the default row permission, are applied implicitly by the database manager to control the set of rows in the table that are accessible.

If a trigger exists for the table, the trigger must be defined with the SECURED attribute (SQLSTATE 55019).

The table must not be referenced in the definition of a view if an INSTEAD OF trigger that is defined with the NOT SECURED attribute exists for the view (SQLSTATE 55019).

If a materialized query table references the table, the functions that are referenced in the *fullselect* of *materizalized-query-definition* must be defined with the SECURED attribute (SQLSTATE 55019).

If a materialized query table (or a staging table) that depends on the table (directly or indirectly through a view) for which row level access control is being activated and that materialized query table (or a staging table) did not already activate row level access control, row level access control is implicitly activated for the materialized query table (or a staging table). This restricts direct access to the contents of the materialized query table (or a staging table). A query that explicitly references the table before such a row permission is defined returns a warning that no data in the table exists (SQLSTATE 02000). To provide access to the materialized query table (or a staging table), an appropriate row permission can be created, or an ALTER TABLE DEACTIVATE ROW ACCESS CONTROL statement on the materialized query table (or a staging table) can be issued to remove the row level protection if that is appropriate.

ACTIVATE ROW ACCESS CONTROL is ignored if row access control is already defined as activated for the table.

If the table is a system-period temporal table, the database manager automatically activates row access control on the history table and creates a default row permission for the history table.

If the table is a column-organized table, the database manager automatically activates row access control on the synopsis table and creates a default row permission for the synopsis table.

ACTIVATE COLUMN ACCESS CONTROL

Activates column level access control on the table. The table must not be a typed table, a catalog table (SQLSTATE 55019), a created temporary table, a declared temporary table (SQLSTATE 42995), a nickname (SQLSTATE 42809), a view (SQLSTATE 42809), or an external table (SQLSTATE 42858). The table must not identify a shadow table or a base table of a shadow table (SQLSTATE 428HZ).

The access to the table is not restricted but when the table is referenced in a data manipulation statement, all enabled column masks that were created for the table are applied implicitly by the database manager to mask the values that are returned for the referenced columns in the final result table of the queries.

If a trigger exists for the table, the trigger must be defined with the SECURED attribute (SQLSTATE 55019).

If a materialized query table references the table, the functions that are referenced in the *fullselect* of *materizalized-query-definition* must be defined with the SECURED attribute (SQLSTATE 55019).

The table must not be referenced in the definition of a view if an INSTEAD OF trigger that is defined with the NOT SECURED attribute exists for the view (SQLSTATE 55019). If a materialized query table that depends on the table (directly or indirectly through a view) for which column level access control is being activated and that materialized query table did not already activate row level access control, row level access control is implicitly activated for the materialized query table. This restricts direct access to the contents of the materialized query table. A query that explicitly references the table before such a row permission is defined returns a warning that no data in the table exists (SQLSTATE 02000). To provide access to the materialized query table, an appropriate row permission can be created, or an ALTER TABLE DEACTIVATE ROW ACCESS CONTROL statement on the materialized query table can be issued to remove the row level protection if that is appropriate.

ACTIVATE COLUMN ACCESS CONTROL is ignored if column level access control is already defined as activated for the table.

If the table is a system-period temporal table, the database manager automatically activates row access control on the history table and creates a default row permission for the history table.

If the table is a column-organized table, the database manager automatically activates row access control on the synopsis table and creates a default row permission for the synopsis table.

DEACTIVATE ROW ACCESS CONTROL

Deactivates row level access control on the table. When the table is referenced in a data manipulation statement, any existing enabled row permissions that are defined on the table are not applied by the database manager to control the set of rows in the table that are accessible. The table must not identify a shadow table or a base table of a shadow table (SQLSTATE 428HZ).

DEACTIVATE ROW ACCESS CONTROL is ignored if row access control is not activated for the table.

DEACTIVATE COLUMN ACCESS CONTROL

Deactivates column level access control on the table. When the table is referenced in a data manipulation statement, any existing enabled column masks defined on the table are not applied by the database manager to control the values that are returned for the columns that are referenced in the final result table of the queries. The table must not identify a shadow table or a base table of a shadow table (SQLSTATE 428HZ).

DEACTIVATE COLUMN ACCESS CONTROL is ignored if column access control is not activated for the table.

RENAME COLUMN *source-column-name* TO *target-column-name*

Renames the column that is specified in *source-column-name* to the name that is specified in *target-column-name*. If the **auto_reval** database configuration parameter is set to DISABLED, the RENAME COLUMN option of the ALTER TABLE statement behaves like it is under the control of revalidation immediate semantics.

The table must not be defined as a history table (SQLSTATE 42986). If the table is a system-period temporal table, the column is also renamed in any associated history table.

Columns that are used in expression-based index keys cannot be renamed (SQLSTATE 42893).

RENAME COLUMN must not rename a column that is referenced in the definition of a row permission or a column mask. Also, It must not rename a column for which a column mask is defined (SQLSTATE 42917). If you rename a column that belongs to a table on which a mask or a permission is defined, or to a table that is referenced in the definition of a mask or a permission, that mask or permission is invalidated. Access to a table that activated column access control and a defined invalid mask on it is blocked until the invalid mask is either disabled, dropped, or re-created (SQLSTATE 560D0). Access to a table that activated row access control and defined an invalid row permission on it is blocked until the invalid permission is either disabled, dropped, or re-created (SQLSTATE 560D0).

source-column-name
Specifies the name of the column that is to be renamed. The *source-column-name* must identify an existing column of the table (SQLSTATE 42703). The name must not be qualified. The name must not identify a column that is otherwise being added, altered, or dropped in the same ALTER TABLE statement (SQLSTATE 42711).

target-column-name
The new name for the column. The name must not be qualified. Existing column names or period names in the table must not be used (SQLSTATE 42711).

DROP PRIMARY KEY
Drops the definition of the primary key and all referential constraints dependent on this primary key. The table must have a primary key (SQLSTATE 42888).

DROP FOREIGN KEY *constraint-name*
Drops the referential constraint *constraint-name*. The *constraint-name* must identify a referential constraint (SQLSTATE 42704). For information about implications of dropping a referential constraint, see Notes.

DROP UNIQUE *constraint-name*
Drops the definition of the unique constraint *constraint-name* and all referential constraints dependent on this unique constraint. The *constraint-name* must identify an existing UNIQUE constraint (SQLSTATE 42704). For information on implications of dropping a unique constraint, see Notes.

DROP CHECK *constraint-name*
Drops the check constraint *constraint-name*. The *constraint-name* must identify an existing check constraint that is defined on the table (SQLSTATE 42704).

DROP CONSTRAINT *constraint-name*
Drops the constraint *constraint-name*. The *constraint-name* must identify an existing check constraint, referential constraint, primary key, or unique constraint defined on the table (SQLSTATE 42704). For information about implications of dropping a constraint, see Notes.

DROP COLUMN
Drops the identified column from the table. The table must not be a typed table (SQLSTATE 428DH). The table cannot have data capture enabled (SQLSTATE 42997).

Dropping a column requires a classic table reorganization before full table access is allowed (SQLSTATE 57007).

An XML column can be dropped only if all of the other XML columns in the table are dropped at the same time only.

DROP COLUMN must not drop a column that is referenced in the definition of a row permission or a column mask (SQLSTATE 42917). However, a column for which a column mask is defined can be dropped. When the column is dropped, any column mask that is defined on that column is also dropped.

column-name
Identifies the column that is to be dropped. The column name must not be qualified. The name must identify a column of the specified table (SQLSTATE 42703). The name must not identify the only column of the table (SQLSTATE 42814), or a column referenced in the definition of a period (SQLSTATE 42817). The name must not identify the last column of the table that is not

hidden (SQLSTATE 428GU). The name must not identify a column in a table that is defined as a system-period temporal table or history table (SQLSTATE 428FR). The name must not identify a column that is part of the distribution key, table-partitioning key, or organizing dimensions (SQLSTATE 42997).

CASCADE

Specifies the following actions, based on the object:

- Any views that depend on the column that is dropped are marked inoperative
- Any indexes, triggers, SQL functions, constraints, or global variables that depend on the column that is dropped are also dropped
- Any decomposition-enabled XSROBJECTs that depend on the table that contains the column are made inoperative for decomposition.

A trigger depends on the column if it is referenced in the UPDATE OF column list, or anywhere in the triggered action. A decomposition-enabled XSROBJECT depends on a table if it contains a mapping of an XML element or attribute to the table. If an SQL function or global variable depends on another database object, it might not be possible to drop the function or global variable by using the CASCADE option. CASCADE is the default.

RESTRICT

Specifies that the column cannot be dropped if any views, indexes, triggers, constraints, or global variables depend on the column, or if any decomposition-enabled XSROBJECT depends on the table that contains the column (SQLSTATE 42893). A trigger depends on the column if it is referenced in the UPDATE OF column list, or anywhere in the triggered action. A decomposition-enabled XSROBJECT depends on a table if it contains a mapping of an XML element or attribute to the table. The first dependent object that is detected is identified in the administration log.

Table 128. Cascaded Effects of Dropping a Column

Operation	RESTRICT Effect	CASCADE Effect
Dropping a column that is referenced by a view or a trigger	Dropping the column is not allowed.	The object and all objects that depend on that object are dropped.
Dropping a column that is referenced in the key of an index	If all columns that are referenced in the index are dropped in the same ALTER TABLE statement, dropping the index is allowed. Otherwise, dropping the column is not allowed.	The index is dropped.
Dropping a column that is referenced in a unique constraint	If all columns that are referenced in the unique constraint are dropped in the same ALTER TABLE statement, and the unique constraint is not referenced by a referential constraint, the columns and the constraint are dropped. (The index that is used to satisfy the constraint is also dropped). Otherwise, dropping the column is not allowed.	The unique constraint and any referential constraints, which reference that unique constraint, are dropped. (Any indexes that are used by those constraints are also dropped).

Table 128. Cascaded Effects of Dropping a Column (continued)

Operation	RESTRICT Effect	CASCADE Effect
Dropping a column that is referenced in a referential constraint	If all columns that are referenced in the referential constraint are dropped in the same ALTER TABLE statement, the columns and the constraint are dropped. Otherwise, dropping the column is not allowed.	The referential constraint is dropped.
Dropping a column that is referenced by a system-generated column that is not being dropped.	Dropping the column is not allowed.	Dropping the column is not allowed.
Dropping a column that is referenced in a check constraint	Dropping the column is not allowed.	The check constraint is dropped.
Dropping a column that is referenced in a decomposition-enabled XSROBJECT	Dropping the column is not allowed.	The XSROBJECT is marked inoperative for decomposition. Reenabling the XSROBJECT might require readjustment of its mappings; afterward, issue an ALTER XSROBJECT ENABLE DECOMPOSITION statement against the XSROBJECT.
Dropping a column that is referenced in the default expression of a global variable	Dropping the column is not allowed.	The global variable is dropped, unless the dropping of the global variable is disallowed because other objects exist, which do not allow the cascade, that depends on the global variable.

DROP RESTRICT ON DROP
Removes the restriction, if there is one, on dropping the table and the table space that contains the table.

DROP DISTRIBUTION
Drops the distribution definition for the table. The table must have a distribution definition (SQLSTATE 428FT). The table space for the table must be defined on a single partition database partition group.

DROP MATERIALIZED QUERY
Changes a materialized query table so that it is no longer considered to be a materialized query table. The table that is specified by *table-name* must be defined as a materialized query table that is not replicated (SQLSTATE 428EW). The definition of the columns of *table-name* is not changed, but the table can no longer be used for query optimization, and the REFRESH TABLE statement can no longer be used.

If row level access control or column level access control is in effect for the table, this control remains after the table is no longer a materialized query table.

ADD PERIOD period-definition
Adds a period definition to the table.

SYSTEM_TIME (*begin-column-name*, *end-column-name*)

Defines a system period with the name SYSTEM_TIME. There must not be a column in the table with the name SYSTEM_TIME (SQLSTATE 42711). A table can have only one SYSTEM_TIME period

(SQLSTATE 42711). *begin-column-name* must be defined as ROW BEGIN and *end-column-name* must be defined as ROW END (SQLSTATE 428HN).

BUSINESS_TIME (*begin-column-name*, *end-column-name*)

Defines an application period with the name BUSINESS_TIME. There must not be a column in the table with the name BUSINESS_TIME (SQLSTATE 42711). A table can have only one BUSINESS_TIME period (SQLSTATE 42711). *begin-column-name* and *end-column-name* must both be defined as DATE or TIMESTAMP(p) where p is 0 - 12 (SQLSTATE 42842), and the columns must be defined as NOT NULL (SQLSTATE 42831). *begin-column-name* and *end-column-name* must not identify a column that is defined with a GENERATED clause (SQLSTATE 428HZ). Business time period columns cannot be added to a table that is in set integrity pending state.

An implicit check constraint is generated to ensure that the value of *end-column-name* is greater than the value of *begin-column-name*. The name of the implicitly created check constraint is DB2_GENERATED_CHECK_CONSTRAINT_FOR_BUSINESS_TIME and must not be the name of an existing check constraint (SQLSTATE 42710).

DROP PERIOD *period-name*

Drops the identified period from the table. The name must not identify a period that was already added or altered in this ALTER TABLE statement (SQLSTATE 42711). Any implicitly generated check constraints for the period (created when the period was defined) and any indexes that reference the period are also dropped.

period-name

Identifies the period. Valid period names are BUSINESS_TIME or SYSTEM_TIME. The period must exist in the table (SQLSTATE 4274M).

When a BUSINESS_TIME period is dropped, all packages with the application-period temporal table dependency type on that table are invalidated. Other dependent objects like views and triggers that record a dependency on the table are also marked as invalid.

SYSTEM_TIME period cannot be dropped if the table is a system-period temporal table (SQLSTATE 428HZ).

DATA CAPTURE

Indicates whether extra information for data replication is to be written to the log.

If the table is a typed table, then this option is not supported (SQLSTATE 428DH for root tables or 428DR for other subtables).

NONE

Indicates that no extra information is logged.

CHANGES

Indicates that extra information with regards to SQL changes to this table is written to the log. This option is required if this table is replicated and the Capture program is used to capture changes for this table from the log.

INCLUDE LONGVAR COLUMNS

Allows data replication utilities to capture changes that are made to LONG VARCHAR or LONG VARGRAPHIC columns. The clause can be specified for tables that do not have any LONG VARCHAR or LONG VARGRAPHIC columns since it is possible to ALTER the table to include such columns.

ACTIVATE NOT LOGGED INITIALLY

Activates the NOT LOGGED INITIALLY attribute of the table for this current unit of work.

Any changes that are made to the table by an INSERT, DELETE, UPDATE, CREATE INDEX, DROP INDEX, or ALTER TABLE in the same unit of work after the table is altered by this statement are not logged. Any changes that are made to the system catalog by the ALTER statement in which the NOT LOGGED INITIALLY attribute is activated are logged. Any subsequent changes that are made in the same unit of work to the system catalog information are logged.

At the completion of the current unit of work, the NOT LOGGED INITIALLY attribute is deactivated and all operations that are done on the table in subsequent units of work are logged.

If you use this feature to avoid locks on the catalog tables while you insert data, it is important that only this clause is specified on the ALTER TABLE statement. Use of any other clause in the ALTER TABLE statement results in catalog locks. If no other clauses are specified for the ALTER TABLE statement, then only a SHARE lock is acquired on the system catalog tables. This can greatly reduce the possibility of concurrency conflicts during time between when this statement is run and when the unit of work in which it was run is ended.

If the table is a typed table, this option is only supported on the root table of the typed table hierarchy (SQLSTATE 428DR).

If the table is a system-period temporal table or a history table, this option is not supported.

For more information about the NOT LOGGED INITIALLY attribute, see the description of this attribute in "CREATE TABLE " on page 1351.

Note: If non-logged activity occurs against a table that has the NOT LOGGED INITIALLY attribute activated, and if a statement fails (causing a rollback), or a ROLLBACK TO SAVEPOINT is run, the entire unit of work is rolled back (SQL1476N). Furthermore, the table for which the NOT LOGGED INITIALLY attribute was activated is marked inaccessible after the rollback occurs and can be dropped only. Therefore, the opportunity for errors within the unit of work in which the NOT LOGGED INITIALLY attribute is activated is minimized.

WITH EMPTY TABLE
Causes all data currently in table to be removed. When the data is removed, it cannot be recovered except through use of the RESTORE facility. If the unit of work in which this alter statement was issued is rolled back, the table data is not returned to its original state.

When this action is requested, no DELETE triggers defined on the affected table are fired. The index data is also deleted for all indexes that exist on the table.

A partitioned table with attached data partitions or logically detached partitions cannot be emptied (SQLSTATE 42928).

PCTFREE *integer*
Specifies the percentage of each page that is to be left as free space during a load or a table reorganization operation. The first row on each page is added without restriction. When more rows are added to a page, at least *integer* percent of the page is left as free space. The PCTFREE value is considered only by the load and table reorg utilities. The value of *integer* can range 0 - 99. A PCTFREE value of -1 in the system catalog (SYSCAT.TABLES) is interpreted as the default value. The default PCTFREE value for a table page is 0. If the table is a typed table, this option is only supported on the root table of the typed table hierarchy (SQLSTATE 428DR).

LOCKSIZE
Indicates the size (granularity) of locks used when the table is accessed. Use of this option in the table definition does not prevent normal lock escalation from occurring.

If the table is a typed table, this option is only supported on the root table of the typed table hierarchy (SQLSTATE 428DR).

The LOCKSIZE keyword is not supported for column-organized tables (SQLSTATE 42858).

ROW
Indicates the use of row locks. This is the default lock size when a table is created.

BLOCKINSERT
Indicates the use of block locks during insert operations. This means that the appropriate exclusive lock is acquired on the block before insertion, and row locking is not done on the inserted row. This option is useful when separate transactions are inserting into separate cells in the table. Transactions inserting into the same cells can still do so concurrently, but insert into distinct blocks, and this can impact the size of the cell if more blocks are needed. This option is only valid for MDC tables (SQLSTATE 42613).

TABLE
Indicates the use of table locks. This means that the appropriate share or exclusive lock is acquired on the table, and that intent locks (except intent none) are not used. For partitioned

tables, this lock strategy is applied to both the table lock and the data partition locks for any data partitions that are accessed. Use of this value can improve the performance of queries by limiting the number of locks that need to be acquired. However, concurrency is also reduced because all locks are held over the complete table.

APPEND
Indicates whether data is appended to the end of the table data or placed where free space is available in data pages. If the table is a typed table, this option is only supported on the root table of the typed table hierarchy (SQLSTATE 428DR).

ON
Indicates that table data is appended and information about free space on pages is not kept. The table must not have a clustered index (SQLSTATE 428CA).

OFF
Indicates that table data is placed in available space. This is the default when a table is created.

The table is reorganized after you set APPEND OFF since the information about available free space is not accurate and can result in poor performance during insert.

VOLATILE CARDINALITY or NOT VOLATILE CARDINALITY
Indicates to the optimizer whether the cardinality of table *table-name* can vary significantly at run time. Volatility applies to the number of rows in the table, not to the table itself. CARDINALITY is an optional keyword. The default is NOT VOLATILE.

VOLATILE
Specifies that the cardinality of table *table-name* can vary significantly at run time, from empty to large. To access the table, the optimizer uses an index scan (rather than a table scan, regardless of the statistics) if that index is index-only (all referenced columns are in the index), or that index is able to apply a predicate in the index scan. The list prefetch access method is not used to access the table. If the table is a typed table, this option is only supported on the root table of the typed table hierarchy (SQLSTATE 428DR).

NOT VOLATILE
Specifies that the cardinality of *table-name* is not volatile. Access plans to this table continues to be based on existing statistics and on the current optimization level.

COMPRESS
Specifies whether data compression applies to the rows of the table.

YES
Specifies that row and XML compression are enabled. Insert and update operations on the table are subject to compression. Index compression is enabled for new indexes unless explicitly disabled in the CREATE INDEX statement. Existing indexes can be compressed by using the ALTER INDEX statement.

After a table is altered to enable row compression, all rows in the table can be compressed immediately by running one of the following actions:

- REORG command
- Online table move
- Data unload and reload

ADAPTIVE
Enables adaptive compression for the table. Data rows are subject to compression with both table-level and page-level compression dictionaries. XML documents in the XML storage object are subject to compression with a table-level XML compression dictionary. Page-level compression dictionaries are created automatically as rows are inserted or updated. Table-level compression dictionaries are created for both row and XML data automatically after sufficient data is added, unless they exist.

STATIC
Enables classic row compression for the table. Data rows are subject to compression with a table-level compression dictionary, and XML documents in the XML storage object are

subject to compression by using a table-level XML compression dictionary. If no table-level compression dictionaries exist for either row or XML data, they will be created automatically after sufficient data is added.

If neither of the preceding two options are specified along with the COMPRESS YES clause, ADAPTIVE is used implicitly.

NO

Specifies that data row and XML compression are disabled. Inserted and updated data rows and XML documents in the table is no longer subject to compression. Any rows and XML documents in the table that are already in compressed format remain in compressed format until they are converted to non-compressed format when they are updated.

An offline reorganization of the table decompresses any rows that are remain compressed.

If table-level or page-level compression dictionaries exist, they are discarded during table reorganization or truncation (such as a LOAD REPLACE operation). Index compression is disabled for new indexes that are created on that table unless explicitly enabled in the CREATE INDEX statement. Index compression for existing indexes can be explicitly disabled by using the ALTER INDEX statement.

VALUE COMPRESSION

This determines the row format that is to be used. Each data type has a different byte count depending on the row format that is used. For more information, see "Byte Counts" in “CREATE TABLE ” on page 1351. An update operation causes an existing row to be changed to the new row format.

Offline table reorganization is recommended to improve the performance of update operations on existing rows. This can also result in the table to take up less space.

If the row size, which is calculated by using the appropriate column in the table "Byte Counts of Columns by Data Type" (see "CREATE TABLE"), would no longer fit within the row size limit, as indicated in the table "Limits for Number of Columns and Row Size In Each Table Space Page Size", an error is returned (SQLSTATE 54010). If the table is a typed table, this option is only supported on the root table of the typed table hierarchy (SQLSTATE 428DR).

ACTIVATE

The NULL value is stored by using 3 bytes. This is the same or less space than when VALUE COMPRESSION is not active for columns of all data types, except for CHAR(1). Whether a column is defined as nullable has no effect on the row size calculation. The zero-length data values for columns whose data type is VARCHAR, VARGRAPHIC, VARBINARY, CLOB, DBCLOB, or BLOB are to be stored by using 2 bytes only, which is less than the storage required when VALUE COMPRESSION is not active. When a column is defined by using the COMPRESS SYSTEM DEFAULT option, this also allows the system default value for the column to be stored by using 3 bytes of total storage. The row format that is used to support this determines the byte counts for each data type, and tends to cause data fragmentation when updating to or from NULL, a zero-length value, or the system default value.

DEACTIVATE

The null value is stored with space set aside for possible future updates. This space is not set aside for varying-length columns. It also does not support efficient storage of system default values for a column. If columns exist with the COMPRESS SYSTEM DEFAULT attribute, a warning is returned (SQLSTATE 01648).

LOG INDEX BUILD

Specifies the level of logging that is to be performed during create, re-create, or reorganize index operations on this table.

NULL

Specifies that the value of the **logindexbuild** database configuration parameter is used to determine whether index build operations are to be logged. This is the default when the table is created.

OFF
Specifies that any index build operations on this table will be logged minimally. This value overrides the setting of the **logindexbuild** database configuration parameter.

ON
Specifies that any index build operations on this table will be logged completely. This value overrides the setting of the **logindexbuild** database configuration parameter.

ADD PARTITION *add-partition*
Adds one or more data partitions to a partitioned table. If the specified table is not a partitioned table, an error is returned (SQLSTATE 428FT). The number of data partitions must not exceed 32 767.

partition-name
Names the data partition. The name must not be the same as any other data partition for the table (SQLSTATE 42710). If this clause is not specified, the name will be 'PART' followed by the character form of an integer value to make the name unique for the table.

boundary-spec
Specifies the range of values for the new data partition. This range must not overlap that of an existing data partition (SQLSTATE 56016). For a description of the starting-clause and the ending-clause, see "CREATE TABLE".

If the starting-clause is omitted, the new data partition is assumed to be at the end of the table. If the ending-clause is omitted, the new data partition is assumed to be at the start of the table.

IN *tablespace-name*
Specifies the table space where the data partition is to be stored. The named table space must have the same page size, be in the same database partition group, and manage space in the same way as the other table spaces of the partitioned table (SQLSTATE 42838). This can be a table space that is already being used for another data partition of the same table, or a table space that is not being used by this table, but it must be a table space on which the authorization ID of the statement holds the USE privilege (SQLSTATE 42727). If this clause is not specified, the table space of the first visible or attached data partition of the table is used.

INDEX IN *tablespace-name*
Specifies the table space where partitioned indexes on the data partition are stored. If the INDEX IN clause is not specified, partitioned indexes on the data partition are stored in the same table space as the data partition.

The table space that is used by the new index partition, whether default or specified by the INDEX IN clause, must match the type (SMS or DMS), page size, and extent size of the table spaces used by all other index partitions (SQLSTATE 42838).

LONG IN *tablespace-name*
Specifies the table space where the data partition that contains long column data is to be stored. The named table space must have the same page size, be in the same database partition group, and manage space in the same way as the other table spaces and data partitions of the partitioned table (SQLSTATE 42838); it must be a table space on which the authorization ID of the statement holds the USE privilege. The page size and extent size for the named table space can be different from the page size and extent size of the other data partitions of the partitioned table.

For rules that govern the use of the LONG IN clause with partitioned tables, see "Large object behavior in partitioned tables".

ATTACH PARTITION *attach-partition*
Attaches another table as a new data partition. The data object of the table being attached becomes a new partition of the table being attached to. There is no data movement involved. The table is placed in set integrity pending state, and referential integrity checking is deferred until execution of a SET INTEGRITY statement. The ALTER TABLE ATTACH operation does not allow the use of the IN or LONG IN clause. The placement of LOBs for that data partition is determined at the time the source table is created. For rules that govern the use of the LONG IN clause with partitioned tables, see "Large object behavior in partitioned tables".

If the table being attached has either row level access control or column level access control activated, then the table to attach to must have the same controls activated. No row permissions or column masks are automatically carried over from the table being attached to the target table. The column masks and row permissions do not necessarily need to be the same on both tables, although this would be best from a security perspective. But if the table being attached has row level access control activated then the table to attach to must also have row level access control activated (SQLSTATE 428GE). Similarly, if the table being attached has column level access control activated and at least one column mask object that is enabled then the table to attach to must also have column level access control activated and a column mask object that is enabled for the corresponding columns (SQLSTATE 428GE).

partition-name
Names the data partition. The name must not be the same as any other data partition for the table (SQLSTATE 42710). If this clause is not specified, the name is 'PART' followed by the character form of an integer value to make the name unique for the table.

boundary-spec
Specifies the range of values for the new data partition. This range must not overlap that of an existing data partition (SQLSTATE 56016). For a description of the starting-clause and the ending-clause, see "CREATE TABLE".

If the starting-clause is omitted, the new data partition is assumed to be at the end of the table. If the ending-clause is omitted, the new data partition is assumed to be at the start of the table.

FROM *table-name1*
Specifies the table that is to be used as the source of data for the new partition. The table definition of *table-name1* cannot have multiple data partitions, and it must match the altered table in the following ways (SQLSTATE 428GE):

- The number of columns must be the same.
- The data types of the columns in the same ordinal position in the table must be the same.
- The nullability characteristic of the columns in the same ordinal position in the table must be the same.
- If the target table has a row change timestamp column, the corresponding column of the source table must be a row change timestamp column.
- If the data is also distributed, it must be distributed over the same database partition group by using the same distribution keys.
- If either table is a random distribution table that uses the random by generation method, the other table must be one too.
- If the data in either table is organized, the organization must match.
- For structured, XML, or LOB data type, the value for INLINE LENGTH must be the same.
- If the target table has a defined BUSINESS_TIME period, the source table must have a defined BUSINESS_TIME period on the corresponding columns.

After the data from *table-name1* is successfully attached, an operation equivalent to DROP TABLE *table-name1* is performed to remove this table, which no longer has data, from the database.

BUILD MISSING INDEXES
Specifies that if the source table does not have indexes that correspond to the partitioned indexes on the target table, a SET INTEGRITY operation builds partitioned indexes on the new data partition to correspond to the partitioned indexes on the existing data partitions. Indexes on the source table that do not match the partitioned indexes on the target table are dropped during attach processing.

REQUIRE MATCHING INDEXES
Specifies that the source table must have indexes to match the partitioned indexes on the target table; otherwise, an error is returned (SQLSTATE 428GE) and information is written to the administration log about the indexes that do not match.

If the REQUIRE MATCHING INDEXES clause is not specified and the indexes on the source table do not match all the partitioned indexes on the target table, the following behavior occurs:

1. For indexes on the target table that do not have a match on the source table and are either unique indexes or XML indexes that are defined with REJECT INVALID VALUES, the ATTACH operation fails (SQLSTATE 428GE).
2. For all other indexes on the target table that do not have a match on the source table, the index object on the source table is marked invalid during the attach operation. If the source table does not have any indexes, an empty index object is created and marked as invalid. The ATTACH operation succeeds, but the index object on the new data partition is marked as invalid. Typically, SET INTEGRITY is the next operation to run against the data partition. SET INTEGRITY forces a rebuild, if required, of the index object on data partitions that were recently attached. The index rebuild can increase the time that is required to bring the new data online.
3. Information is written to the administration log about the indexes that do not match.

DETACH PARTITION *partition-name* INTO *table-name1*

Detaches the data partition *partition-name* from the altered table, and uses the data partition to create a new table named *table-name1*. The data partition is detached from the altered table and is used to create the new table without any data movement. The specified data partition cannot be the last remaining partition of the table being altered (SQLSTATE 428G2). The table being altered to detach a partition must not be a system-period temporal table (SQLSTATE 428HZ).

When a partition is detached from a table for which either row level access control or column level access control is defined, the new table that is created for the detached data will automatically have row level access control (though not column level access control) activated to protect the detached data. Direct access to this new table returns no rows until appropriate row permissions are defined for the table or row level access control is deactivated for this table.

ADD SECURITY POLICY *policy-name*

Adds a security policy to the table. The security policy must exist at the current server (SQLSTATE 42704). The table must not already have a security policy (SQLSTATE 55065), and must not be a typed table (SQLSTATE 428DH), materialized query table (MQT), or staging table (SQLSTATE 428FG).

DROP SECURITY POLICY

Removes the security policy and all LBAC protection from the table. The table that is specified by *table-name* must be protected by a security policy (SQLSTATE 428GT). If the table has a column with data type DB2SECURITYLABEL, the data type is changed to VARCHAR (128) FOR BIT DATA. If the table has one or more protected columns, those columns become unprotected.

ADD VERSIONING USE HISTORY TABLE *history-table-name*

Specifies that the table is a system-period temporal table. The table must not already be defined as a system-period temporal table or a history table (SQLSTATE 428HM). A SYSTEM_TIME period and a transaction-start-ID column must be defined in the table (SQLSTATE 428HM).

The table must not be a materialized query table (SQLSTATE 428HM).

Historical versions of the rows in the table are retained by the database manager. The database manager records extra information that indicates when a row was inserted into the table, and when it was updated or deleted. When a row in a system-period temporal table is updated, a previous version of the row is kept. When data in a system-period temporal table is deleted, the old version of the row is inserted as a historical record. An associated history table is used to store the historical rows of the table.

References to the table can include a time period search condition to indicate which system versions of the data are to be returned.

history-table-name identifies a history table where historical rows of the system-period temporal table are kept. *history-table-name* must identify a table that exists at the current server (SQLSTATE 42704), and is not a catalog table (SQLSTATE 42832), an existing system-period temporal table, an existing history table, a declared global temporary table, a created global temporary table, a materialized query table, or a view (SQLSTATE 428HX).

The identified history table must not contain an identity column, row change timestamp column, row-begin column, row-end column, transaction start-ID column, generated expression column, or include a period (SQLSTATE 428HX).

The system-period temporal table and the identified history table must have the same number and order of columns (SQLSTATE 428HX). The following attributes for the corresponding columns of the two tables must be the same (SQLSTATE 428HX):

- Column name
- Column data type
- Column length (including inline LOB lengths), precision, and scale
- Column FOR BIT attribute for character string columns
- Column null attribute
- Column hidden attribute

If row access control or column access control is activated for the system-period temporal table and row access control is not activated on the history table, the database manager automatically activates row access control on the history table and creates a default row permission for the history table.

DROP VERSIONING

Specifies that the table is no longer a system-period temporal table. The table must be a system-period temporal table (SQLSTATE 428HZ). Historical data is no longer recorded and maintained for the table. The definition of the columns and data of the table are not changed, but the table is no longer treated as a system-period temporal table. The SYSTEM_TIME period is retained. Subsequent queries that reference the table must not specify a SYSTEM_TIME period specification for the table. The relationship between the system-period temporal table and the associated history table is removed. The history table is not dropped and the contents of the history table are not affected.

When a table is altered with DROP VERSIONING, all packages with the system-period temporal table dependency type on that table are invalidated. Other dependent objects like views and triggers that record a dependency on the table are also marked as invalid.

Rules

- Any enforced unique or primary key constraint that is defined on the table must be a superset of the distribution key, if there is one (SQLSTATE 42997).
- Primary or unique keys cannot be subsets of dimensions (SQLSTATE 429BE).
- A column can be referenced in one ADD, ALTER, or DROP COLUMN clause in a single ALTER TABLE statement only (SQLSTATE 42711).
- A column length, data type, or hidden attribute cannot be altered, nor can the column be dropped, if the table has any materialized query tables that depend on the table (SQLSTATE 42997).
- VARCHAR, VARGRAPHIC, and VARBINARY columns that was altered to be greater than 4000 and 2000, respectively, must not be used as input parameters in functions in the SYSFUN schema (SQLSTATE 22001).
- A column length cannot be altered if the table has any views enabled for query optimization that depend on the table (SQLSTATE 42997).
- The table must be put in set integrity pending state, by using the SET INTEGRITY statement with the OFF option (SQLSTATE 55019) before:
  - Adding a column with a generation expression
  - Altering the generated expression of a column
  - Changing a column to have a generated expression
- An existing column cannot be altered to become of type DB2SECURITYLABEL (SQLSTATE 42837).
- Defining a column of type DB2SECURITYLABEL fails if the table is not associated with a security policy (SQLSTATE 55064).

- A column of type DB2SECURITYLABEL cannot be altered or dropped (SQLSTATE 42817).
- An ALTER TABLE operation to mark a table as protected fails if there exists an MQT that depends on that table (SQLSTATE 55067).
- Attaching a partition to a protected partitioned table fails if the source table and the target table are not protected by using the same security policy, do not have the same row security label column, and do not have the same set of protected columns (SQLSTATE 428GE).
- If a generated column is referenced in a table-partitioning key, the generated column expression cannot be altered (SQLSTATE 42837).
- The *isolation-clause* cannot be specified in the *fullselect* of the *materialized-query-definition* (SQLSTATE 42601).
- Adding or attaching a data partition to a partitioned table fails with SQL0612N after detaching the same partition name, if asynchronous index cleanup has not finished deleting index entries for the partition (SQLSTATE 42711).

Notes

- A *REORG-recommended operation* occurs when changes that result from an ALTER TABLE statement affect the row format of existing data. When this occurs, most subsequent operations on the table are restricted until a table reorganization operation completes successfully (SQLSTATE 57007). Many REORG-recommended ALTER TABLE actions may be run against a table in a single unit of work, either in the same statement or spread over multiple statements. This is considered to be a single REORG-recommended operation. For example, dropping two columns in a single ALTER TABLE statement or in two statements in the same unit of work is considered to be a single REORG-recommended operation. There can be up to three units of work containing REORG-recommended operations before a classic offline table reorganization must be done (SQLTATE 55019). The reorg pending state and the number of REORG-recommended operations are returned from SYSIBMADM.ADMINTABINFO columns reorg_pending and num_reorg_rec_alters respectively.
- The following is the full list of REORG-recommended ALTER statements that cause a version change and place the table into a REORG-pending state:
  - DROP COLUMN
  - ALTER COLUMN SET NOT NULL
  - ALTER COLUMN DROP NOT NULL
  - ALTER COLUMN SET DATA TYPE, except in the following situations:
    - Increasing the length of a VARCHAR or VARGRAPHIC column
    - Decreasing the length of a VARCHAR or VARGRAPHIC column without truncating trailing blanks from existing data, when no indexes exist on the column
- The following table operations are allowed after a successful REORG-recommended operation occurs:
  - ALTER TABLE. However, the following operations are not allowed (SQLSTATE 57007):
    - ADD CHECK CONSTRAINT
    - ADD REFERENTIAL CONSTRAINT
    - ADD UNIQUE CONSTRAINT
  - DROP TABLE
  - RENAME TABLE
  - REORG TABLE
  - TRUNCATE TABLE
  - Table scan access of table data
- Altering a table to make it a materialized query table puts the table in set integrity pending state. If the table is defined as REFRESH IMMEDIATE, the table must be taken out of set integrity pending state before INSERT, DELETE, or UPDATE commands can be started on the table that is referenced by the

fullselect. The table can be taken out of set integrity pending state by using REFRESH TABLE or SET INTEGRITY, with the IMMEDIATE CHECKED option, to completely refresh the data in the table based on the fullselect. If the data in the table accurately reflects the result of the fullselect, the IMMEDIATE UNCHECKED option of SET INTEGRITY can be used to take the table out of set integrity pending state.

- Altering a table to change it to a REFRESH IMMEDIATE materialized query table causes any packages with INSERT, DELETE, or UPDATE usage on the table that is referenced by the fullselect to be invalidated.
- Altering a table to change from a materialized query table to a regular table causes any packages dependent on the table to be invalidated.
- Altering a table to change from a MAINTAINED BY FEDERATED_TOOL materialized query table to a regular table does not cause any change in the subscription setup of the replication tool. Because a subsequent change to a MAINTAINED BY SYSTEM materialized query table causes the replication tool to fail, you must change the subscription setting when you change a MAINTAINED BY FEDERATED_TOOL materialized query table.
- If a deferred materialized query table is associated with a staging table, the staging table is dropped if the materialized query table is altered to a regular table.
- ADD column clauses are processed before all other clauses. Other clauses are processed in the order that they are specified.
- Any columns added through an alter table operation is not automatically added to any existing view of the table.
- Adding or attaching a data partition to a partitioned table, or detaching a data partition from a partitioned table, causes any packages that depend on that table to be invalidated.
- After you detach a data partition from a data partitioned table, the STATUS of the detached partition in the SYSCAT.DATAPARTITIONS catalog can be 'L' when the partition is logically detached and the detach operation is not completed. If the STATUS of the detached partition is 'L', the following operations cannot be performed on the source table (SQLSTATE 55057):
  - Adding a unique or primary key constraint that attempts to create a nonpartitioned index
  - Adding, dropping, or renaming a column
  - Activating value compression or compression
  - Deactivating value compression or compression
- To drop the partitioning for a table, the table must be dropped and then re-created.
- To drop the organization for a table, the table must be dropped and then re-created.
- When an index is automatically created for a unique or primary key constraint, the database manager tries to use the specified constraint name as the index name with a schema name that matches the schema name of the table. If this matches an existing index name or no name for the constraint was specified, the index is created in the SYSIBM schema with a system-generated name that is formed of "SQL" followed by a sequence of 15 numeric characters that are generated by a timestamp-based function.
- When a nonpartitioned index is created on a partitioned table with attached data partitions, the index does not include the data in the attached data partitions. Use the SET INTEGRITY statement to maintain all indexes for all attached data partitions.
- When you create a partitioned index in the presence of attached data partitions (STATUS of 'A' in SYSCAT.DATAPARTITIONS), an index partition for each attached data partition is also created. If the partitioned index is being created as unique, or is an XML index that is created with REJECT INVALID VALUES, then the index creation can fail if an attached data partition contains any violations (duplicates for a unique index, or invalid values for the XML index).
- If a table has a nonpartitioned index, you cannot access a new data partition in that table within the same transaction as the add or attach operation that created the partition, if the transaction does not have the table locked in exclusive mode (SQLSTATE 57007).

- Any table that might be involved in a DELETE operation on table T is said to be *delete-connected* to T. Thus, a table is delete-connected to T if it is a dependent of T or it is a dependent of a table in which deletes from T cascade.
- A package has an insert (update/delete) usage on table T if records are inserted into (updated in/deleted from) T either directly by a statement in the package, or indirectly through constraints or triggers run by the package on behalf of one of its statements. Similarly, a package has an update usage on a column if the column is modified directly by a statement in the package, or indirectly through constraints or triggers run by the package on behalf of one of its statements.
- In a federated system, a remote base table that was created by using transparent DDL can be altered. However, transparent DDL does impose some limitations on the modifications that can be made:
  - A remote base table can be altered by adding new columns or specifying a primary key only.
  - Specific clauses that are supported by transparent DDL include:
    - ADD COLUMN *column-definition*
    - NOT NULL and PRIMARY KEY in the *column-options* clause
    - ADD *unique-constraint* (PRIMARY KEY only)
  - You cannot specify a comment on an existing column in a remote base table.
  - An existing primary key in a remote base table cannot be altered or dropped.
  - Altering a remote base table invalidates any packages that depend on the nickname that is associated with that remote base table.
  - The remote data source must support the changes being requested through the ALTER TABLE statement. Depending on how the data source responds to requests it does not support, an error might be returned or the request might be ignored.
  - An attempt to alter a remote base table that was not created by using transparent DDL returns an error.
- Any changes, whether implicit or explicit, to primary key, unique keys, or foreign keys might have the following effects on packages, indexes, and other foreign keys:

Table 129. Changes to keys, and their effects on packages, indexes, and other foreign keys

Action	**Effect on packages, indexes, and other foreign keys**
Primary key or unique key is added	There is no effect on packages, foreign keys, or existing unique keys. (If the primary or unique key uses an existing unique index that was created in a previous version and was not converted to support deferred uniqueness, the index is converted, and packages with update usage on the associated table are invalidated).
Primary key or unique key is dropped	– The index is dropped if it was automatically created for the constraint. Any packages dependent on the index are invalidated. – The index is set back to non-unique if it was converted to unique for the constraint and it is no longer system-required. Any packages dependent on the index are invalidated. – The index is set to no longer system required if it was an existing unique index that is used for the constraint. There is no effect on packages. – A primary key or unique constraint of the table cannot be dropped if it is the last enforced primary key or unique constraint whose set of columns is in the select list of an associated shadow table. – A primary key or unique constraint cannot be dropped if the table has an associated shadow table, and the primary key of the associated shadow table depends on the constraint being dropped. – All dependent foreign keys are dropped. Further action is taken for each dependent foreign key, as specified in the next row.

Table 129. Changes to keys, and their effects on packages, indexes, and other foreign keys (continued)	
Action	**Effect on packages, indexes, and other foreign keys**
Foreign key is added, dropped, altered from NOT ENFORCED to ENFORCED, or altered from ENFORCED to NOT ENFORCED	– All packages with an insert usage on the object table are invalidated. – All packages with an update usage on at least one column in the foreign key are invalidated. – All packages with a delete usage on the parent table are invalidated. – All packages with an update usage on at least one column in the parent key are invalidated.
Foreign key or a functional dependency is altered from ENABLE QUERY OPTIMIZATION to DISABLE QUERY OPTIMIZATION	All packages with dependencies on the constraint for optimization purposes are invalidated.

- Adding a column to a table results in invalidation of all packages with insert usage on the altered table. If the added column is the first user-defined structured type column in the table, packages with DELETE usage on the altered table is also invalidated.
- Adding a check or referential constraint to a table that exists and that is not in set integrity pending state, or altering the existing check or referential constraint from NOT ENFORCED to ENFORCED on an existing table that is not in set integrity pending state causes the existing rows in the table to be immediately evaluated against the constraint. If the verification fails, an error is returned (SQLSTATE 23512). If a table is in set integrity pending state, adding a check or referential constraint, or altering a constraint from NOT ENFORCED to ENFORCED does not immediately lead to the enforcement of the constraint. Issue the SET INTEGRITY statement with the IMMEDIATE CHECKED option to begin enforcing the constraint.
- Adding, altering, or dropping a check constraint results in invalidation of all packages with either an insert usage on the object table, an update usage on at least one of the columns that are involved in the constraint, or a select usage using the constraint to improve performance.
- Adding a distribution key invalidates all packages with an update usage on at least one of the columns of the distribution key.
- A distribution key that was defined by default is not affected by dropping the primary key and adding a different primary key.
- Dropping a column or changing its data type removes all runstats information from the table being altered. Runstats must be run on the table after it is again accessible. The statistical profile of the table is preserved if the table does not contain a column that was explicitly dropped.
- Altering a column (to change its length, data type, nullability, or hidden attribute) or dropping a column invalidates all packages that reference (directly or indirectly through a referential constraint or trigger) its table.
- Altering a column (to change its length, data type, nullability, or hidden attribute) regenerates views (except typed views) that depend on its table. If a problem occurs while regenerating such a view, an error is returned (SQLSTATE 56098). Any typed views that depend on the table are marked inoperative.
- Altering a column (to change its length, data type, or hidden attribute) marks all dependent triggers and SQL functions as invalid; they are implicitly recompiled on next use. If a problem occurs while regenerating such an object, an error is returned (SQLSTATE 56098).
- Altering a column (to change its length, data type, or nullability attribute) might cause errors (SQLSTATE 54010) while processing a trigger or an SQL function when a statement that involves the trigger or SQL function is prepared or bound. This can occur if the row size based on the sum of the lengths of the transition variables and transition table columns is too long. If such a trigger or SQL function is dropped, a subsequent attempt to re-create it returns an error (SQLSTATE 54040).

- A WLM activity event monitor created in an earlier version must be dropped and re-created to add new table columns that are introduced by this fix pack and any subsequent fix packs or releases.
- Altering a structured or XML type column to increase the inline length invalidates all packages that reference the table, either directly or indirectly through a referential constraint or trigger.
- Altering a structured or XML type column to increase the inline length regenerates views that depend on the table.
- A compression dictionary can be created for the XML storage object of a table only if the XML columns are added to the table, or if the table is migrated the using an online table move.
- Changing the LOCKSIZE for a table results in invalidation of all packages that have a dependency on the altered table.
- Changing VOLATILE or NOT VOLATILE CARDINALITY results in invalidation of all dynamic SQL statements that have a dependency on the altered table.
- ***Replication:*** Exercise caution when you increase the length or changing the data type of a column. The change data table that is associated with an application table might already be at or near the row size limit. The change data table must be altered before the application table, or the two tables must be altered within the same unit of work to ensure that the alteration can be completed for both tables. Consideration should be given to copies, which might also be at or near the row size limit, or reside on platforms that lack the ability to increase the length of an existing column.

  If the change data table is not altered before the Capture program processes log records with the altered attributes, the Capture program will likely fail. If a copy that contains the altered column is not altered before the subscription maintaining the copy runs, the subscription will likely fail.
- When detaching a partition from a protected table, the target table that is automatically created by the database server is protected in the same way the source table is protected.
- When a table is altered such that it becomes protected with row level granularity, any cached dynamic SQL sections that depend on such a table are invalidated. Similarly, any packages that depend on such a table are also invalidated.
- When a column of a table, T, is altered such that it becomes a protected column, any cached dynamic SQL sections that depend on table T are invalidated. Similarly, any packages that depend on table T are also invalidated.
- When a column of a table, T, is altered such that it becomes a non protected column, any cached dynamic SQL sections that depend on table T are invalidated. Similarly, any packages that depend on table T are also invalidated.
- For existing rows in the table, the value of the security label column defaults to the security label for write access of the session authorization ID at the time the ALTER statement that adds a row security label column is executed.
- ***Add materialized query:*** When a base table is altered to become a materialized query table, the label-based access control security attributes (security policy, column security labels, row security label column) are derived in the same way when creating a new materialized query table. If the base table that is altered already has label-based access control security attributes, these attributes are factored in the derivation process as follows:
  - Column access control: The existing security label for a column is aggregated with the corresponding security label that is derived from the query that defines the materialized query table.
  - Row access control: The row access control attributes are set up exactly in the same way as for a new materialized query table.
- In Db2 Version 9.7 Fix Pack 1 or later releases, new multidimensional clustering (MDC) table block indexes are partitioned. Adding a data partition to a data partitioned multidimensional clustering (MDC) table creates the corresponding empty index partitions for the new partition, including the MDC block indexes. Also, a new index partition entry is added to SYSCAT.SYSINDEXPARTITIONS for each MDC block index and for each partitioned index.
- When you attach a data partition to a partitioned MDC table created with Db2 V9.7 Fix Pack 1 or later releases, the source table that is specified by *attach-partition* can be a nonpartitioned MDC table or a single-partition partitioned MDC table.

- **If the source table is nonpartitioned:** MDC block indexes on the source table will be inherited and become the partitioned MDC indexes for the new partition after the ATTACH operation completes.
- **If the source table is partitioned:** If the source table is a partitioned MDC table that is created with Db2 V9.7 Fix Pack 1 or later releases, the block indexes are partitioned. The block indexes become the new block indexes on the partition.
- If the source partitioned MDC table is created at a level lower than Db2 V9.7 Fix Pack 1, the block indexes on the table are nonpartitioned. During the ATTACH operation, the block indexes are dropped and created as partitioned indexes similar to the other partitioned indexes on the source table.

  Issuing the SET INTEGRITY statement on the target table is required to bring the attached partition online.

  If the REQUIRE MATCHING INDEXES clause is specified, and the target table is a partitioned MDC table that is created in Db2 V9.7 Fix Pack 1 or later releases, the ALTER TABLE ... ATTACH PARTITION statement fails and returns SQL20307N (SQLSTATE 428GE). Removing the REQUIRE MATCHING INDEXES clause allows the attach process to proceed.

If the target partitioned MDC table was created at a level lower than Db2 V9.7 Fix Pack 1, the block indexes are nonpartitioned. The block indexes on the source MDC table are dropped during the ATTACH operation. Issuing a SET INTEGRITY statement on the target table is required to bring the attached partition online. New rows from the attached partition are added to existing nonpartitioned block indexes.

- When you detach a data partition from a data partitioned MDC table that is created at a level lower than Db2 V9.7 Fix Pack 1, the block indexes are nonpartitioned. The following restrictions apply:
  - Access to the newly detached table is not allowed in the same unit of work as the detach operation.
  - Block indexes on the target table, created as part of the detach operation, are rebuilt upon the first access to the table after the detach operation is committed. If the source table had any partitioned indexes before the detach operation, then the index object for the target table is marked invalid to allow for recreation of the block indexes. As a result, access time is increased while the block indexes and all other partitioned indexes are re-created.

  When you detach a partition from a partitioned MDC table created by using Db2 V9.7 Fix Pack 1 or later releases, the block indexes are partitioned, and the previous restrictions do not apply. Assuming that no other dependent objects such as dependent MQTs exist, access to the newly detached table is allowed in the same unit of work. All the partitioned indexes, including block indexes, become indexes on the target table without the need to be re-created.
- ***Considerations for implicitly hidden columns:*** A column that is defined as implicitly hidden can be explicitly referenced in an ALTER TABLE statement. For example, an implicitly hidden column can be altered or specified as part of a referential constraint, check constraint, or materialized query table definition.

  Altering a table to make some of its columns implicitly hidden can impact the behavior of data movement utilities that are working with the table. When a table contains implicitly hidden columns, utilities like IMPORT, INGEST, and LOAD require that you specify whether data for the hidden columns is included in the operation. For example, this might mean that a load operation that ran successfully before the table was altered, now fails (SQLCODE SQL2437N). Similarly, EXPORT requires that you specify whether data for the hidden columns is included in the operation.

  Data movement utilities must use the DB2_DMU_DEFAULT registry variable, or the **implicitlyhiddeninclude** or **implicitlyhiddenmissing** file type modifiers when working with tables that contain implicitly hidden columns.
- ***Row access control that is activated explicitly***: The ACTIVATE ROW ACCESS CONTROL clause is used to activate row access control for a table. When this happens, a default row permission is implicitly created and allows no access to any rows of the table, unless permitted by a row permission that is explicitly created by the security administrator. The default row permission is always enabled.

  When the table is referenced in a data manipulation statement, all enabled row permissions that have been created for the table, including the default row permission, are implicitly applied by the database manager to control which rows in the table are accessible. A row access control search

condition is derived by application of the logical 0R operator to the search condition in each enabled row permission. This derived search condition acts as a filter to the table before any user specified operations, such as predicates, grouping, ordering, are processed. This derived search condition permits the authorization IDs that are specified in the permission definitions to access certain rows in the table.

When the ACTIVATE ROW ACCESS CONTROL clause is used, all the packages and dynamically cached statements that reference the table are invalidated.

Row access control remains enforced until the DEACTIVATE ROW ACCESS CONTROL clause is used to stop enforcing it.

- ***Implicit object that is created when row access control is activated for a table***: When the ACTIVATE ROW ACCESS CONTROL clause is used to activate row access control for a table, the database manager implicitly creates a default row permission for the table. The default row permission prevents all access to the table. The implicitly created row permission resides in the same schema of the base table and has a name in the form of SYS_DEFAULT_ROW_PERMISSION__table-name ... up to 128 characters. Notice two underscores after "PERMISSION". If this name is not unique, the last 4 characters are reserved for a unique number *'nnnn'*, where *'nnnn'* is a four-alphanumeric-character string that start at '0000' and is incremented by one value each time until a unique name is found.

  The owner of the default row permission is SYSIBM. The default row permission is always enabled. The default row permission is dropped when row access control is deactivated or when the table is dropped.
- ***Activating column access control:*** The ACTIVATE COLUMN ACCESS CONTROL clause is used to activate column level access control for a table. The access to the table is not restricted but when the table is referenced in a data manipulation statement, all enabled column masks that were created for the table are applied to mask the column values referenced in the final result table.

  When column masks are used to mask the column values, they determine the values in the final result table. If a column has a column mask and the column (specifically a simple reference to a column name or a column that is embedded in an expression) appears in the outermost select list, the column mask is applied to the column to produce the values for the final result table. If the column does not appear in the outermost select list but it participates in the final result table, for example, it appears in a materialized table expression or view, the column mask is applied to the column in such a way that the masked value is included in the result table of the materialized table expression or view so that it can be used in the final result table.

  The application of column masks does not interfere with the operations of other clauses within the statement such as the WHERE, GROUP BY, HAVING, SELECT DISTINCT, and ORDER BY. The rows that are returned in the final result table remain the same, except that the values in the resulting rows might have been masked by the column masks. As such, if the masked column also appears in an ORDER BY *sort-key*, the order is based on the original column values and the masked values in the final result table might not reflect that order. Similarly, the masked values might not reflect the uniqueness that is enforced by SELECT DISTINCT.

  A column mask is applied in the following contexts:

  – The outermost SELECT clause or clauses of a SELECT or SELECT INTO statement, or if the column does not appear in the outermost select list but it participates in the final result table, one or more outermost SELECT clauses of the corresponding materialized table expression or view where the column appears.
  – The outermost SELECT clause or clauses of a SELECT FROM INSERT, SELECT FROM UPDATE, or SELECT FROM DELETE operation.
  – The outermost SELECT clause or clauses that are used to derive the new values for an INSERT, UPDATE, or MERGE statement, or a SET *transition-variable-name* assignment statement. The same masking applies to a scalar fullselect expression that appears in the outermost SELECT clause or clauses of the previously mentioned statements, the right side of a SET *host-variable* assignment statement, the VALUES INTO statement, or the VALUES statement.

  Column masks are not applied when the masked column appears in the following contexts:

  – WHERE clauses.

  - GROUP BY clauses.
  - HAVING clauses.
  - SELECT DISTINCT.
  - ORDER BY clauses.
- ***Row and column access control are not enforced when EXPLAIN tables are populated:*** Row and column access control can be enforced for EXPLAIN tables. However, the enabled row permissions and column masks are not applied when the database manager inserts rows into those tables.
- ***Row and column access control are not enforced when event monitor tables are populated:*** Row and column access control can be enforced for event monitor tables. However, the enabled row permissions and column masks are not applied when the database manager inserts rows into those tables.
- ***Row and column access control are not enforced when temporal history tables are populated:*** Row and column access control can be enforced for temporal history tables. However, the enabled row permissions and column masks are not applied when the database manager accesses those tables for operations on the system-period temporal tables.
- ***Stop enforcing row or column access control:*** The DEACTIVATE ROW ACCESS CONTROL clause is used to stop enforcing row access control for a table. The default row permission is dropped. Thereafter, when the table is referenced in a data manipulation statement, explicitly created row permissions are not applied.

  The DEACTIVATE COLUMN ACCESS CONTROL clause is used to stop enforcing column access control for a table. Thereafter, when the table is referenced in a data manipulation statement, the column masks are not applied.

  The explicitly created row permissions or column masks, if any, remain but have no effect.

  All the packages and dynamically cached statements that reference the table are invalidated when row or column access control is deactivated.
- ***Secure triggers for row and column access control:*** Triggers are used for database integrity, and as such, a balance between row and column access control (security) and database integrity is needed. Enabled row permissions and column masks are not applied to the initial values of transition variables and transition tables. Row and column access control enforced for the triggering table is also ignored for any transition variables or transition tables that are referenced in the trigger body. To ensure that no security concern exist for SQL statements in the trigger action to access sensitive data in transition variables and transition tables, the trigger must be created or altered with the SECURED option. If a trigger is not secure, row and column access control cannot be enforced for the triggering table (SQLSTATE 55019).
- ***Secure user-defined functions for row and column access control:*** If a row permission or column mask definition references a user-defined function, the function must be altered with the SECURED option because the sensitive data might be passed as arguments to the function. When a user-defined function is referenced in a data manipulation statement where a table that enforces row or column access control is referenced, and the function arguments reference the columns from such a table, if the function is not secure, this impacts the access plan selection and might yield poor performance. The database manager considers the SECURED option an assertion that declares the user has established a change control audit procedure for all changes to the user-defined function. It is assumed that such a control audit procedure is in place and that all subsequent ALTER FUNCTION statements or changes to external packages are being reviewed by this audit process.
- ***Database operations where row and column access control is not applicable:*** Row and column access control must not compromise database integrity. Columns that are involved in primary keys, unique keys, indexes, check constraints, and referential integrity must not be subject to row and column access control. Column masks can be defined for those columns but they are not applied during the process of key building or constraint or RI enforcement.
- ***Defining a system-period temporal table:*** A system-period temporal table definition includes the following aspects:
  - A system period named SYSTEM_TIME, which is defined by using a row-begin column and a row-end column. See the descriptions of AS ROW BEGIN, AS ROW END, and period-definition.

  - A transaction-start-ID column. See the description of AS TRANSACTION START ID.
  - A system-period data versioning definition that is specified on a subsequent ALTER TABLE statement by using the ADD VERSIONING action, which includes the name of the associated history table. See the description of the ADD VERSIONING clause under ALTER TABLE.

  To ensure that the history table cannot be implicitly dropped when a system-period temporal table is dropped, use the WITH RESTRICT ON DROP clause in the definition of the history table.
- ***Defining an application-period temporal table:*** An application-period temporal table definition includes an application period with the name BUSINESS_TIME. The application period is defined by using a begin column and an end column with both columns having the same data type that is either DATE or TIMESTAMP(**p**). See the description of period-definition.

  Data-change operations on an application-period temporal table can result in an automatic insert of one or two extra rows when a row is updated or deleted. When an update or delete of a row in an application-period temporal table is specified for a portion of the period represented by that row, the row is updated or deleted and one or two rows are automatically inserted to represent the portion of the row that is not changed. New values are generated for each generated column in an application-period temporal table for each row that is automatically inserted as a result of an update or delete operation on the table. If a generated column is defined as part of a unique or primary key, parent key in a referential constraint, or unique index, it is possible that an automatic insert violates a constraint or index in which case an error is returned.
- ***Considerations for transaction-start-ID columns:*** A transaction-start-ID column contains a null value if the column allows null values, and there is a row-begin column, and the value of the row-begin column is unique from values of row-begin columns that are generated for other transactions. Given that the column might contain null values, it is recommended that one of the following methods be used when you retrieve a value from the column:
  - COALESCE (transaction_start_id_col, row_begin_col)
  - CASE WHEN transaction_start_id_col IS NOT NULL THEN transaction_start_id_col ELSE row_begin_col END
- ***Considerations for system-period temporal tables and row and column access control:*** Row and column access control can be defined on both the system-period temporal table and the associated history table.
  - When a system-period temporal table is accessed, any row and column access rules that are defined on the system-period temporal table are applied to all of the rows that are returned from the system-period temporal table, regardless of whether the rows are stored in the system-period temporal table or the history table. The row and column access rules that are defined on the history table are not applied.
  - When the history table is accessed directly, the row and column access rules that are defined on the history table are applied.

  When a system-period temporal table is defined and row access control or column access control is activated for the system-period temporal table, the database manager automatically activates row access control on the history table and creates a default row permission for the history table.
- ***Considerations for column-organized tables:*** The following options can be specified to alter a column-organized table definition (underlined options are defaults):
  - ACTIVATE NOT LOGGED INITIALLY
  - ACTIVATE/DEACTIVATE COLUMN ACCESS CONTROL
  - ACTIVATE/DEACTIVATE ROW ACCESS CONTROL
  - ADD *{PRIMARY KEY | UNIQUE}* [ENFORCED | NOT ENFORCED]
  - ADD COLUMN, unless there is an outstanding asynchronous background process that is creating the table's column compression dictionary (SQL20054N)
  - ADD CONSTRAINT *<constraint-name>* CHECK NOT ENFORCED (when used with *{ENABLE | DISABLE}* QUERY OPTIMIZATION)

  - ADD CONSTRAINT *<constraint-name> {PRIMARY KEY | UNIQUE | FOREIGN KEY NOT ENFORCED}*
  - ADD MATERIALIZED QUERY
  - ADD PERIOD {SYSTEM_TIME | BUSINESS_TIME}
  - ADD RESTRICT ON DROP
  - ADD VERSIONING USE HISTORY TABLE
  - ALTER COLUMN *<column name>* SET DATA TYPE (increase VARCHAR/VARGRAPHIC column length only)
  - ALTER COLUMN <column-name> SET *{NOT HIDDEN | IMPLICITLY HIDDEN}*
  - DATA CAPTURE NONE
  - DROP *{PRIMARY KEY | UNIQUE | CONSTRAINT} <constraint-name>*
  - DROP DEFAULT
  - DROP GENERATED
  - DROP MATERIALIZED QUERY
  - DROP PERIOD {SYSTEM_TIME | BUSINESS_TIME}
  - DROP RESTRICT ON DROP
  - DROP VERSIONING
  - LOG INDEX BUILD *{NULL | OFF | ON}*
  - SET GENERATED [ALWAYS | BY DEFAULT]
  - SET DEFAULT
  - SET GENERATED AS ROW {BEGIN | END}
  - SET GENERATED AS TRANSACTION START ID

  Other options are not supported for column-organized tables.
- ***Considerations for random distribution tables***
  - Altering the table to make it a materialized query table is not supported
- ***Considerations for random distribution tables that use generate by random method***
  - The following options are not supported to alter a random distribution key column definition (in addition to those options, which are already not supported for a distribution key column in a non-random distribution table):
    - ALTER COLUMN *<column-name>* SET NOT HIDDEN
    - ALTER COLUMN *<column-name>* DROP NOT NULL
    - ALTER COLUMN *<column-name>* DROP GENERATED
    - ALTER COLUMN *<column-name>* COMPRESS *{OFF | SYSTEM DEFAULT}*
    - RENAME COLUMN *<column-name>*
  - Random distribution cannot be dropped with ALTER TABLE DROP DISTRIBUTION statement.
- ***Syntax alternatives***: The following alternatives are non-standard. They are supported for compatibility with earlier product versions or with other database products.
  - The ADD keyword is optional for:
    - Unnamed PRIMARY KEY constraints
    - Unnamed referential constraints
    - Referential constraints whose name follows the phrase FOREIGN KEY
  - The CONSTRAINT keyword can be omitted from a *column-definition* defining a references-clause
  - *constraint-name* can be specified following FOREIGN KEY (without the CONSTRAINT keyword)
  - SET SUMMARY AS can be specified in place of SET MATERIALIZED QUERY AS

- SET MATERIALIZED QUERY AS DEFINITION ONLY can be specified in place of DROP MATERIALIZED QUERY
- SET MATERIALIZED QUERY AS (fullselect) can be specified in place of ADD MATERIALIZED QUERY (fullselect)
- ADD PARTITIONING KEY can be specified in place of ADD DISTRIBUTE BY HASH; the optional USING HASHING clause can also still be specified in this case
- DROP PARTITIONING KEY can be specified in place of DROP DISTRIBUTION
- The LONG VARCHAR and LONG VARGRAPHIC data types continue to be supported but are deprecated and not recommended, especially for portable applications
- A comma can be used to separate multiple options in the *identity-alteration* clause
- PART can be specified in place of PARTITION
- VALUES can be specified in place of ENDING AT
- NOMINVALUE, NOMAXVALUE, NOCYCLE, NOCACHE, and NOORDER can be specified in place of NO MINVALUE, NO MAXVALUE, NO CYCLE, NO CACHE, and NO ORDER, respectively
- DROP EXPRESSION can be specified in place of DROP GENERATED to drop the generated expression attribute for a column.
- DROP IDENTITY can be specified in place of DROP GENERATED to drop the identity attribute for a column.
- When you specify the value of the datetime special register, NOW() can be specified in place of CURRENT_TIMESTAMP.

- When an ALTER TABLE statement invalidates an existing VIEW, the statistics profile for the invalidated VIEW is blanked.

Examples

1. Add a column named RATING, which is one character long, to the DEPARTMENT table.

```
ALTER TABLE DEPARTMENT
   ADD RATING CHAR(1)
```

2. Add a column named SITE_NOTES to the PROJECT table. Create SITE_NOTES as a varying-length column with a maximum length of 1000 bytes. The values of the column do not have an associated character set and therefore must not be converted.

```
ALTER TABLE PROJECT
   ADD SITE_NOTES  VARCHAR(1000) FOR BIT DATA
```

3. Assume a table that is called EQUIPMENT exists defined with the following columns:

```
Column Name        Data Type
EQUIP_NO           INT
EQUIP_DESC         VARCHAR(50)
LOCATION           VARCHAR(50)
EQUIP_OWNER        CHAR(3)
```

   Add a referential constraint to the EQUIPMENT table so that the owner (EQUIP_OWNER) must be a department number (DEPTNO) that is present in the DEPARTMENT table. DEPTNO is the primary key of the DEPARTMENT table. If a department is removed from the DEPARTMENT table, the owner (EQUIP_OWNER) values for all equipment that is owned by that department should become unassigned (or set to null). Give the constraint the name DEPTQUIP.

```
ALTER TABLE EQUIPMENT
     ADD CONSTRAINT DEPTQUIP
      FOREIGN KEY (EQUIP_OWNER)
        REFERENCES DEPARTMENT
           ON DELETE SET NULL
```

Also, an extra column is needed to allow the recording of the quantity that is associated with this equipment record. Unless otherwise specified, the EQUIP_QTY column should have a value of 1 and must never be null.

```
ALTER TABLE EQUIPMENT
  ADD COLUMN EQUIP_QTY
  SMALLINT NOT NULL DEFAULT 1
```

4. Alter table EMPLOYEE. Add the check constraint that is named REVENUE defined so that each employee must make a total of salary and commission greater than $30,000.

```
ALTER TABLE EMPLOYEE
  ADD CONSTRAINT REVENUE
  CHECK (SALARY + COMM > 30000)
```

5. Alter table EMPLOYEE. Drop the constraint REVENUE, which was previously defined.

```
ALTER TABLE EMPLOYEE
  DROP CONSTRAINT REVENUE
```

6. Alter a table to log SQL changes in the default format.

```
ALTER TABLE SALARY1
  DATA CAPTURE NONE
```

7. Alter a table to log SQL changes in an expanded format.

```
ALTER TABLE SALARY2
  DATA CAPTURE CHANGES
```

8. Alter the EMPLOYEE table to add four new columns with default values.

```
ALTER TABLE EMPLOYEE
   ADD COLUMN HEIGHT MEASURE   DEFAULT MEASURE(1)
   ADD COLUMN BIRTHDAY BIRTHDATE DEFAULT DATE('01-01-1850')
   ADD COLUMN FLAGS BLOB(1M)  DEFAULT BLOB(X'01')
   ADD COLUMN PHOTO PICTURE   DEFAULT BLOB(X'00')
```

The default values use various function names when specifying the default. Since MEASURE is a distinct type based on INTEGER, the MEASURE function is used. The HEIGHT column default could have been specified without the function since the source type of MEASURE is not BLOB or a datetime data type. Since BIRTHDATE is a distinct type based on DATE, the DATE function is used (BIRTHDATE cannot be used here). For the FLAGS and PHOTO columns the default is specified by using the BLOB function even though PHOTO is a distinct type. To specify a default for BIRTHDAY, FLAGS and PHOTO columns, a function must be used because the type is a BLOB or a distinct type that is sourced on a BLOB or datetime data type.

9. A table called CUSTOMERS is defined with the following columns:

```
Column Name         Data Type
BRANCH_NO           SMALLINT
CUSTOMER_NO         DECIMAL(7)
CUSTOMER_NAME       VARCHAR(50)
```

In this table, the primary key is made up of the BRANCH_NO and CUSTOMER_NO columns. To distribute the table, you need to create a distribution key for the table. The table must be defined in a table space on a single-node database partition group. The primary key must be a superset of the distribution key columns, and at least one of the columns of the primary key must be used as the distribution key. Make BRANCH_NO the distribution key as follows:

```
ALTER TABLE CUSTOMERS
  ADD DISTRIBUTE BY HASH (BRANCH_NO)
```

10. A remote table EMPLOYEE was created in a federated system by using transparent DDL. Alter the remote table EMPLOYEE to add the columns PHONE_NO and WORK_DEPT; also, add a primary key on the existing column EMP_NO and the new column WORK_DEPT.

```
ALTER TABLE EMPLOYEE
  ADD COLUMN PHONE_NO CHAR(4) NOT NULL
  ADD COLUMN WORK_DEPT CHAR(3)
  ADD PRIMARY KEY (EMP_NO, WORK_DEPT)
```

11. Alter the DEPARTMENT table to add a functional dependency FD1, then drop the functional dependency FD1 from the DEPARTMENT table.

```
ALTER TABLE DEPARTMENT
  ADD CONSTRAINT FD1
    CHECK ( DEPTNAME DETERMINED BY DEPTNO) NOT ENFORCED

ALTER TABLE DEPARTMENT
  DROP CHECK FD1
```

12. Change the default value for the WORKDEPT column in the EMPLOYEE table to 123.

```
ALTER TABLE EMPLOYEE
  ALTER COLUMN WORKDEPT
    SET DEFAULT '123'
```

13. Associate the security policy DATA_ACCESS with the table EMPLOYEE.

```
ALTER TABLE EMPLOYEE
  ADD SECURITY POLICY DATA_ACCESS
```

14. Alter the table EMPLOYEE to protect the SALARY column.

```
ALTER TABLE EMPLOYEE
  ALTER COLUMN SALARY
  SECURED WITH EMPLOYEESECLABEL
```

15. Assume that you have a table that is named SALARY_DATA that is defined with the following columns:

```
Column Name            Data Type
-----------            ---------
EMP_NAME               VARCHAR(50) NOT NULL
EMP_ID                 SMALLINT NOT NULL
EMP_POSITION           VARCHAR(100) NOT NULL
SALARY                 DECIMAL(5,2)
PROMOTION_DATE         DATE NOT NULL
```

Change this table to allow salaries to be stored in a DECIMAL(6,2) column, make PROMOTION_DATE an optional field that can be set to the null value, and remove the EMP_POSITION column.

```
ALTER TABLE SALARY_DATA
  ALTER COLUMN SALARY SET DATA TYPE DECIMAL(6,2)
  ALTER COLUMN PROMOTION_DATE DROP NOT NULL
  DROP COLUMN EMP_POSITION
```

16. Add a column named DATE_ADDED to the table BOOKS. The default value for this column is the current time stamp.

```
ALTER TABLE BOOKS
  ADD COLUMN DATE_ADDED TIMESTAMP
  WITH DEFAULT CURRENT TIMESTAMP
```

17. Alter table with label-based access control security attributes into a materialized query table. Base tables tt1 and tt2 exist and were created with the following SQL:

```
CREATE TABLE tt1
   (c1 INT SECURED WITH C, c2 DB2SECURITYLABEL) SECURITY POLICY P;
CREATE TABLE tt2
   (c3 INT SECURED WITH B, c4 DB2SECURITYLABEL) SECURITY POLICY P;
```

Table tt2 can be altered to be a materialized query table with the following SQL:

```
ALTER TABLE tt2 ADD (SELECT * FROM tt1 WHERE c1 > 10)
   DATA INITIALLY DEFERRED REFRESH DEFERRED;
```

Table tt2 becomes a materialized query table with the secure policy P. tt2.c3 has security label P.B. tt2.c4 has security label P.C and it is also DB2SECURITYLABEL.

ALTER TABLESPACE

The ALTER TABLESPACE statement is used to modify an existing table space.

You can modify a table space in the following ways:

- Add a container to, or drop a container from a DMS table space; that is, a table space created with the MANAGED BY DATABASE option.
- Modify the size of a container in a DMS table space.
- Lower the high water mark for a DMS table space through extent movement.
- Add a container to an SMS table space on a database partition that currently has no containers.
- Modify the PREFETCHSIZE setting for a table space.
- Modify the BUFFERPOOL used for tables in the table space.
- Modify the OVERHEAD setting for a table space.
- Modify the TRANSFERRATE setting for a table space.
- Modify the file system caching policy for a table space.
- Enable or disable auto-resize for a DMS or automatic storage table space.
- Rebalance a regular or large automatic storage table space.
- Modify the DATA TAG setting for a table space.
- Alter a DMS table space to an automatic storage table space.
- Modify the STOGROUP setting that is associated with a table space.

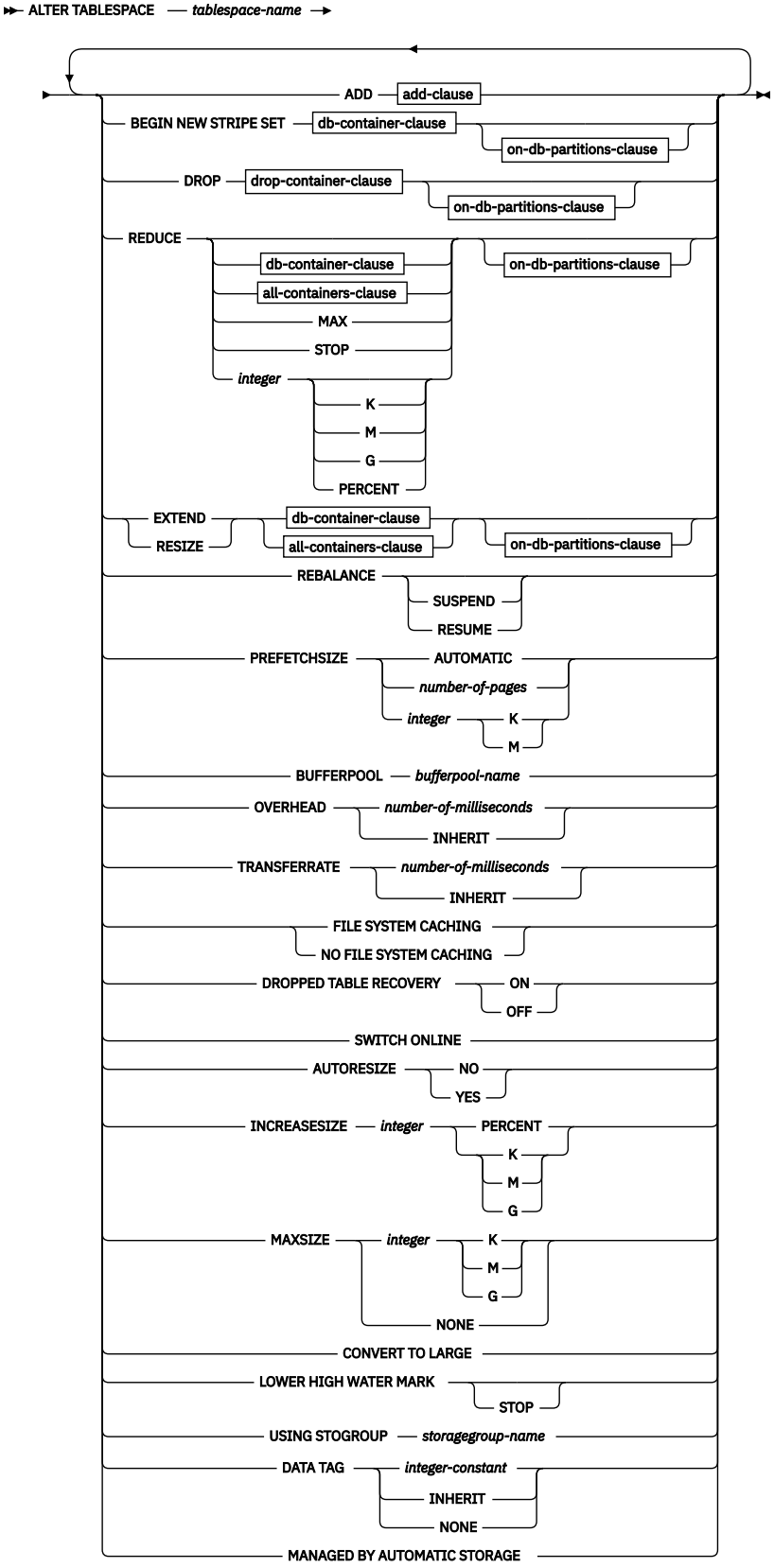
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

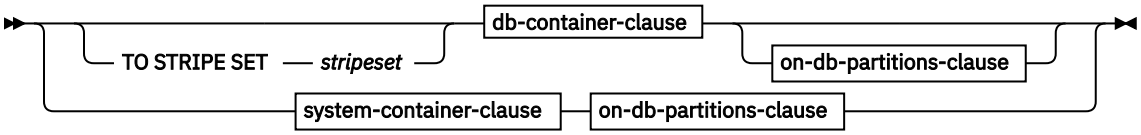
Authorization

The privileges that are held by the authorization ID of the statement must include SYSCTRL or SYSADM authority.

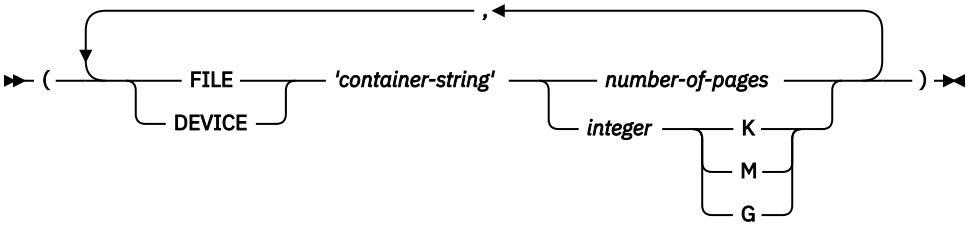
Syntax



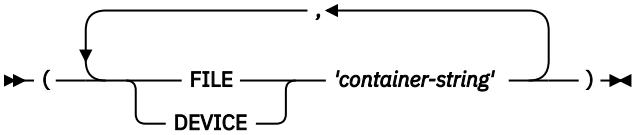
add-clause



db-container-clause

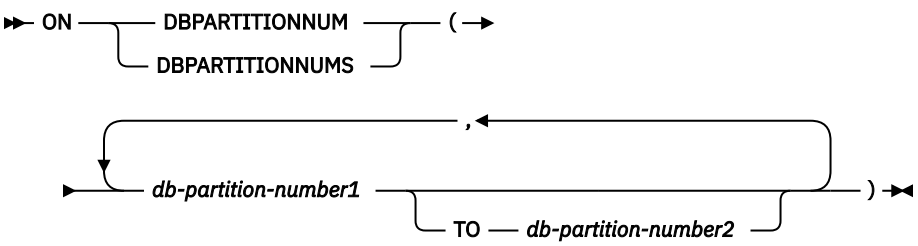


drop-container-clause

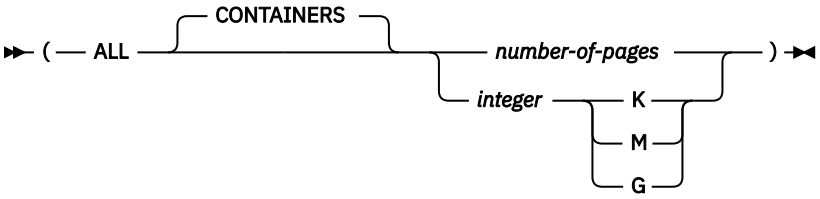


system-container-clause

on-db-partitions-clause



all-containers-clause



Description

tablespace-name
Names the table space. This is a one-part name. It is a long SQL identifier (either ordinary or delimited).

ADD
Specifies that one or more new containers are to be added to the table space.

TO STRIPE SET *stripeset*
Specifies that one or more new containers are added to the table space, and that they are placed into the provided stripe set.

BEGIN NEW STRIPE SET
Specifies that a new stripe set is to be created in the table space, and that one or more containers are to be added to this new stripe set. Subsequently, containers that are added by using the ADD option are added to this new stripe set unless TO STRIPE SET is specified.

DROP
Specifies that one or more containers are to be dropped from the table space.

REDUCE
For non-automatic storage table spaces, specifies that existing containers are to be reduced in size. The size that is specified is the size by which the existing container is decreased. If the *all-containers-clause* is specified, all containers in the table space decrease by this size. If the reduction in size results in a table space size that is smaller than the current high water mark, an attempt is made to reduce the high water mark before attempting to reduce the containers. For non-automatic storage table spaces, the REDUCE clause must be followed by a *db-container-clause* or an *all-containers-clause*.

For automatic storage table spaces, specifies that the current high water mark is to be reduced, if possible, and that the size of the table space is to be reduced to the new high water mark. For automatic storage table spaces, the REDUCE clause must not be followed by a *db-container-clause*, an *all-containers-clause*, or an *on-db-partitions-clause*.

Note: The REDUCE option with the MAX, numeric value, PERCENT, or STOP clauses, and the LOWER HIGH WATER MARK option with the STOP clause, are only available for database and automatic storage-managed table spaces with the reclaimable storage attribute. Moreover, these options must be specified and run without any other options, including each other.

The **MAX**, **STOP**, *integer* **[K | M | G]**, or *integer* **PERCENT** clause takes effect when the statement is processed and is not rolled back if the unit of work, in which the statement is executed, is rolled back.

db-container-clause
Adds one or more containers to a DMS table space. The table space must identify a DMS table space that already exists at the application server.

all-containers-clause
Extends, reduces, or resizes all of the containers in a DMS table space. The table space must identify a DMS table space that already exists at the application server.

MAX
For automatic storage table spaces with reclaimable storage, specifies that the maximum number of extents should be moved to the beginning of the table space to lower the high water mark. Additionally, the size of the table space is reduced to the new high water mark. This reduction does not apply to non-automatic storage table spaces.

STOP
For automatic storage table spaces with reclaimable storage, interrupts the extent movement operation if in progress. This option is not available for non-automatic storage table spaces.

***integer* [K | M | G] or *integer* PERCENT**
For automatic storage table spaces with reclaimable storage, specifies the numeric value by which the table space is to be reduced through extent movement. The value can be expressed in several ways:

- An integer that is specified without K, M, G, or PERCENT indicates that the numeric value is the number of pages by which the table space is to be reduced.
- An integer that is specified with K, M, or G indicates the reduction size in kilobytes, megabytes, or gigabytes. The value is first converted from bytes to number of pages based on the page size of the table space.
- An integer that is specified with PERCENT indicates the number of extents to move, as a percentage of the current size of the table space.

After extent movement is complete, the table space size is reduced to the new high water mark. This option is not available for non-automatic storage table spaces.

on-db-partitions-clause
Specifies one or more database partitions for the corresponding container operations.

EXTEND
Specifies that existing containers are to be increased in size. The size specified is the size by which the existing container is increased. If the *all-containers-clause* is specified, all containers in the table space increase by this size.

RESIZE
Specifies that the size of existing containers is to be changed. The size specified is the new size for the container. If the *all-containers-clause* is specified, all containers in the table space are changed to this size. If the operation affects more than one container, these containers must all either increase in size, or decrease in size. It is not possible to increase some while decreasing others (SQLSTATE 429BC).

db-container-clause
Adds one or more containers to a DMS table space. The table space must identify a DMS table space that already exists at the application server.

drop-container-clause
Drops one or more containers from a DMS table space. The table space must identify a DMS table space that already exists at the application server.

system-container-clause
Adds one or more containers to an SMS table space on the specified database partitions. The table space must identify an SMS table space that already exists at the application server. There must not be any containers on the specified database partitions for the table space (SQLSTATE 42921).

on-db-partitions-clause
Specifies one or more database partitions for the corresponding container operations.

all-containers-clause
Extends, reduces, or resizes all of the containers in a DMS table space. The table space must identify a DMS table space that already exists at the application server.

REBALANCE
For regular and large automatic storage table spaces, initiates the creation of containers on recently added storage paths, the drop of containers from storage paths that are in the "Drop Pending" state, or both. During the rebalance, data is moved into containers on new paths, and moved out of containers on dropped paths. The rebalance runs asynchronously in the background and does not affect the availability of data.

Note: The **SUSPEND** or **RESUME** clause takes effect when the statement is processed and is not rolled back if the unit of work, in which the statement is executed, is rolled back.

SUSPEND
Suspends the active rebalance operation on the specified table space. If no active rebalance operation exists, no action is taken and success is returned. The suspend state is persistent and if the database is deactivated while the rebalance is suspended, then upon database activation the rebalance operation is restarted from the suspended state. Suspending a rebalance operation when it is already suspended has no effect and success is returned.

RESUME
Resumes a previously suspended rebalance operation. If no active rebalance operation exists, no action is taken and success is returned. If the rebalance is PAUSED because of an online backup operation, then the table space rebalance is taken out of the suspended state but remains paused until the online backup is completed.

PREFETCHSIZE
Specifies to read in the data needed by a query before it is referenced by the query so that the query does not need to wait for I/O to be performed.

AUTOMATIC
Specifies that the prefetch size of a table space is to be updated automatically; that is, the prefetch size is managed by the database manager.

The database updates the prefetch size automatically whenever the number of containers in a table space changes (following successful execution of an ALTER TABLESPACE statement that adds or drops one or more containers). The prefetch size is also automatically updated at database startup.

Automatic updating of the prefetch size can be turned off by specifying a numeric value in the PREFETCHSIZE clause.

number-of-pages
Specifies the number of PAGESIZE pages that are read from the table space when data prefetching is being performed. The maximum value is 32767.

***integer* K | M**
Specifies the prefetch size value as an integer value followed by K (for kilobytes) or M (for megabytes). If specified in this way, the floor of the number of bytes divided by the page size is used to determine the number of pages value for prefetch size.

BUFFERPOOL *bufferpool-name*
The name of the buffer pool that is used for tables in this table space. The buffer pool must currently exist in the database (SQLSTATE 42704). The database partition group of the table space must be defined for the buffer pool (SQLSTATE 42735).

OVERHEAD *number-of-milliseconds* or OVERHEAD INHERIT
Specifies the I/O controller overhead and disk seek and latency time. This value is used to determine the cost of I/O during query optimization. For more information on tuning, refer to Table space impact on query optimization.

number-of-milliseconds
Any numeric literal (integer, decimal, or floating point) that specifies the I/O controller overhead and disk seek and latency time, in milliseconds. The number should be an average for all containers that belong to the table space, if not the same for all containers.

INHERIT
If INHERIT is specified, the table space must be defined using automatic storage and the OVERHEAD is dynamically inherited from the storage group. INHERIT cannot be specified if the table space is not defined using automatic storage (SQLSTATE 42858). If the OVERHEAD is set to undefined for the storage group and you set OVERHEAD to INHERIT, the database creation default is used.

version 10.1 For a database that was created in Db2 or later, the default I/O controller overhead and disk seek and latency time is 6.725 milliseconds.

For a database that was upgraded from a previous version of Db2 to Db2 version 10.1 or later, the default I/O controller overhead and disk seek and latency time is as follows:

- 7.5 milliseconds for a database that is created in Db2 version 9.7 or higher.

TRANSFERRATE *number-of-milliseconds* or TRANSFERRATE INHERIT
Specifies the time to read one page into memory. This value is used to determine the cost of I/O during query optimization. For more information on tuning, refer to Table space impact on query optimization.

number-of-milliseconds
Any numeric literal (integer, decimal, or floating point) that specifies the time to read one page (4 K or 8 K) into memory, in milliseconds. The number should be an average for all containers that belong to the table space, if not the same for all containers.

INHERIT
If INHERIT is specified, the table space must be defined using automatic storage and the TRANSFERRATE is dynamically inherited from the storage group. INHERIT cannot be specified if the table space is not defined using automatic storage (SQLSTATE 42858). If the DEVICE READ RATE of the storage group is set to undefined and the user sets TRANSFERRATE to INHERIT, the database creation default is used.

When an automatic storage table space inherits the TRANSFERRATE setting from the storage group it is using, the DEVICE READ RATE of the storage group, which is in megabytes per second, is converted into milliseconds per page read accounting for the table space's PAGESIZE setting of the table space. The conversion formula follows:

TRANSFERRATE = (1 / DEVICE READ RATE) * 1000 / 1024000 * PAGESIZE

For a database that was created in Db2 version 10.1 or later, the default time to read one page into memory for 4 KB PAGESIZE table space is 0.04 milliseconds.

For a database that was upgraded from a previous version of Db2 to Db2 version 10.1 or later, the default time to read one page into memory is as follows:

- 0.06 milliseconds for a database that is created in Db2 version 9.7 or higher

FILE SYSTEM CACHING or NO FILE SYSTEM CACHING
Specifies whether I/O operations are buffered or non-cached at the file system level. Changes to the I/O mode are not dynamic and will take effect on the next database activation. The default I/O mode is determined based on operating system, file system type, and in the case of SMS table spaces, data object type. For more information, see "file system caching configurations". After a non-default file system caching option is chosen, it is not possible to return to the default (unspecified) behavior. Instead, the file system caching mode must be selected explicitly.

FILE SYSTEM CACHING
All I/O operations in the target table space are cached at the file system level.

NO FILE SYSTEM CACHING
Specifies that all I/O operations are to bypass the file system-level cache. LOB and Long field data in SMS table spaces are excepted.

Note:

Db2 supports disk devices with physical sector sizes of 512 bytes or 4096 bytes.

Support for 4096 byte sector sizes is not enabled by default, and can be enabled using the **DB2_4K_DEVICE_SUPPORT** registry variable.

DROPPED TABLE RECOVERY
Specifies whether tables that have been dropped from *tablespace-name* can be recovered by using the **RECOVER DROPPED TABLE ON** option of the **ROLLFORWARD DATABASE** command. For partitioned tables, dropped table recovery is always on, even if dropped table recovery is turned off for non-partitioned tables in one or more table spaces.

ON
Specifies that dropped tables can be recovered.

OFF
Specifies that dropped tables cannot be recovered.

SWITCH ONLINE
Specifies that table spaces in OFFLINE state are to be brought online if their containers become accessible. If the containers are not accessible, an error is returned (SQLSTATE 57048).

AUTORESIZE
Specifies whether the auto-resize capability of a database-managed space (DMS) table space or an automatic storage table space is to be enabled. Auto-resizable table spaces automatically increase in size when they become full.

NO
Specifies that the auto-resize capability of a DMS table space or an automatic storage table space is to be disabled. If the auto-resize capability is disabled, any values that were previously specified for INCREASESIZE or MAXSIZE are not kept.

YES
Specifies that the auto-resize capability of a DMS table space or an automatic storage table space is to be enabled.

INCREASESIZE *integer* PERCENT or INCREASESIZE *integer* K | M | G
Specifies the amount, per database partition, by which a table space that is enabled for auto-resize is automatically increased, in the case that the table space is full and a request for space was made. The integer value must be followed by:

- PERCENT to specify the amount as a percentage of the table space size at the time that a request for space is made. When PERCENT is specified, the integer value must be in the range 0 - 100 (SQLSTATE 42615).
- K (for kilobytes), M (for megabytes), or G (for gigabytes) to specify the amount in bytes.

The actual value used might be slightly smaller or larger than what was specified, because the database manager strives to maintain consistent growth across containers in the table space.

MAXSIZE *integer* K | M | G or MAXSIZE NONE
Specifies the maximum size to which a table space that is enabled for auto-resize can automatically be increased.

integer
Specifies a hard limit on the size, per database partition, to which a DMS table space or an automatic storage table space can automatically be increased. The integer value must be followed by K (for kilobytes), M (for megabytes), or G (for gigabytes). The actual value used might be slightly smaller than what was specified, because the database manager strives to maintain consistent growth across containers in the table space.

NONE
Specifies that the table space is to be allowed to grow to file system capacity, or to the maximum table space size (described in "SQL and XML limits").

CONVERT TO LARGE
Modifies an existing regular DMS table space to be a large DMS table space. The table space and its contents are locked during conversion. This option can only be used on regular DMS table spaces. If an SMS table space, a temporary table space, or the system catalog table space is specified, an error is returned (SQLSTATE 560CF). You cannot convert a table space that contains a data partition of a partitioned table that has data partitions in another table space (SQLSTATE 560CF). Conversion cannot be reversed after being committed. If tables in the table space are defined with DATA CAPTURE CHANGES, consider the storage and capacity limits of the target table and table space.

LOWER HIGH WATER MARK
For both automatic storage and non-automatic storage table spaces with reclaimable storage, triggers the extent movement operation to move the maximum number of extents lower in the table space. Although the high water mark is lowered, the size of the table space is not reduced. This must be followed by an ALTER TABLESPACE REDUCE for automatic storage table spaces or ALTER TABLESPACE REDUCE with the *db-container-clause* or *all-containers-clause* for non-automatic storage table spaces.

Note: The LOWER HIGH WATER MARK option with the STOP clause, and the REDUCE option with the MAX, numeric value, PERCENT, or STOP clauses, are only available for database-managed and automatic storage-managed table spaces with the reclaimable storage attribute. Moreover, these options must be specified and run without any other options, including each other.

Note: This clause takes effect when the statement is processed and is not rolled back if the unit of work, in which the statement is executed, is rolled back.

STOP
For both automatic storage and non-automatic storage table spaces with reclaimable storage, interrupts the extent movement operation if in progress.

USING STOGROUP
Associates a table space with a different storage group. The data that is associated with the table space is moved from its current storage group to the specified storage group. This clause only applies to automatic storage table spaces unless specified with the MANAGED BY AUTOMATIC STORAGE clause (SQLSTATE 42858).

For automatic storage table spaces, an implicit REBALANCE is started at commit time. For a database-managed table space being converted to automatic storage-managed, an explicit REBALANCE statement is required.

In a partitioned database environment, to alter the storage group association of a table space, the table space must be defined by using automatic storage on all database partitions. If the table space on any database partition is not defined by using automatic storage, this command will fail unless specified with the MANAGED BY AUTOMATIC STORAGE clause (SQLSTATE 42858). However, it is not required that a table space has the same storage group association on all database partitions for this command to succeed in moving the table space on all database partitions.

storagegroup-name
Identifies the storage group in which table space data is stored. *storagegroup-name* must identify a storage group that exists at the current server (SQLSTATE 42704). This is a one-part name.

DATA TAG *integer-constant*, DATA TAG INHERIT or DATA TAG NONE
Specifies a tag for the data in the table space. This value can be used as part of a WLM configuration in a work class definition or referenced within a threshold definition. For more information, refer to the CREATE WORK CLASS SET, ALTER WORK CLASS SET, CREATE THRESHOLD, and ALTER THRESHOLD statements. This clause cannot be specified for USER or SYSTEM TEMPORARY table spaces or for the catalog table space (SQLSTATE 42858).

integer-constant
Valid values for *integer-constant* are integers in the range 1 - 9. If an *integer-constant* is specified and an associated storage group exists, the data tag that is specified for the table space overrides any data tag value that is specified for the associated storage group.

INHERIT
If INHERIT is specified, the table space must be defined using automatic storage and the DATA TAG is dynamically inherited from the storage group. INHERIT cannot be specified if the table space is not defined using automatic storage (SQLSTATE 42858).

NONE
If NONE is specified, there is no data tag.

MANAGED BY AUTOMATIC STORAGE
Enables automatic storage for a database-managed (DMS) table space. After automatic storage is enabled, no further container operations can be executed on the table space. The table space being converted cannot be using RAW (DEVICE) containers.

If the USING STOGROUP clause is not included when converting from a DMS table space to an automatic storage table space, then the default storage group is specified.

Rules

- The BEGIN NEW STRIPE SET clause cannot be specified in the same statement as ADD, DROP, EXTEND, REDUCE, and RESIZE, unless those clauses are being directed to different database partitions (SQLSTATE 429BC).
- The stripe set value that is specified with the TO STRIPE SET clause must be within the valid range for the table space being altered (SQLSTATE 42615).
- When adding or removing space from the table space, the following rules must be followed:
  - EXTEND and RESIZE can be used in the same statement if the size of each container is increasing (SQLSTATE 429BC).
  - REDUCE and RESIZE can be used in the same statement if the size of each container is decreasing (SQLSTATE 429BC).
  - EXTEND and REDUCE cannot be used in the same statement, unless they are being directed to different database partitions (SQLSTATE 429BC).
  - ADD cannot be used with REDUCE or DROP in the same statement, unless they are being directed to different database partitions (SQLSTATE 429BC).

  - DROP cannot be used with EXTEND or ADD in the same statement, unless they are being directed to different database partitions (SQLSTATE 429BC).
- The AUTORESIZE, INCREASESIZE, or MAXSIZE clause cannot be specified for system-managed space (SMS) table spaces, temporary table spaces that were created using automatic storage, or DMS table spaces that are defined to use raw device containers (SQLSTATE 42601).
- The INCREASESIZE or MAXSIZE clause cannot be specified if the table space is not auto-resizable (SQLSTATE 42601).
- When specifying a new maximum size for a table space, the value must be larger than the current size on each database partition (SQLSTATE 560B0).
- Container operations (ADD, EXTEND, RESIZE, DROP, or BEGIN NEW STRIPE SET) cannot be performed on automatic storage table spaces, because the database manager is controlling the space management of such table spaces (SQLSTATE 42858).
- Raw device containers cannot be added to an auto-resizable DMS table space (SQLSTATE 42601).
- The CONVERT TO LARGE clause cannot be specified in the same statement as any other clause (SQLSTATE 429BC).
- The REBALANCE clause cannot be specified with any other clause (SQLSTATE 429BC).
- The REBALANCE clause is only valid for regular and large automatic storage table spaces (SQLSTATE 42601). Temporary automatic storage table spaces should be dropped and re-created to take advantage of recently added storage paths or to have their containers removed from storage paths being dropped.
- Container operations and the REBALANCE clause cannot be specified if the table space is in the "DMS rebalancer is active" state (SQLSTATE 55041).
- The USING STOGROUP clause cannot be specified for temporary table spaces (SQLSTATE 42858).
- The following clauses are not supported in Db2 pureScale environments:
  - ADD *db-container-clause*
  - BEGIN NEW STRIPE SET *db-container-clause*
  - DROP *db-container-clause*
  - REBALANCE
  - RESIZE *db-container-clause*
  - USING STOGROUP
- The ADD, DROP, RESIZE, EXTEND, REDUCE, LOWER HIGH WATER MARK, and BEGIN_STRIPE_SET clauses cannot be used with the MANAGED BY AUTOMATIC STORAGE clause or the USING STOGROUP clause (SQLSTATE 429BC).
- The USING STOGROUP clause cannot be specified if the table space is in the "rebalancer is active "state (SQLSTATE 55041).
- ***Container size limit***: In DMS table spaces, a container must be at least two times the extent size pages in length (SQLSTATE 54039). The maximum size of a container is operating system dependent.
- ***Container definition length limit***: Each container definition requires 53 bytes plus the number of bytes necessary to store the container name. The combined length of all container definitions for the table space cannot exceed 208 kilobytes (SQLSTATE 54034).

Notes

- Default container operations are container operations that are specified in the ALTER TABLESPACE statement, but that are not explicitly directed to a specific database partition. These container operations are sent to any database partition that is not listed in the statement. If these default container operations are not sent to any database partition, because all database partitions are explicitly mentioned for a container operation, a warning is returned (SQLSTATE 01589).

- After space has been added or removed from a table space, and the transaction is committed, the contents of the table space can be rebalanced across the containers. Access to the table space is not restricted during rebalancing.
- If the table space is in OFFLINE state and the containers are accessible, the user can disconnect all applications and connect to the database again to bring the table space out of OFFLINE state. Alternatively, the SWITCH ONLINE option can bring the table space up (out of OFFLINE) while the rest of the database is still up and being used.
- If adding more than one container to a table space, it is recommended that they are added in the same statement so that the cost of rebalancing is incurred only once. An attempt to add containers to the same table space in separate ALTER TABLESPACE statements within a single transaction result in an error (SQLSTATE 55041).
- Any attempts to extend, reduce, resize, or drop containers that do not exist will raise an error (SQLSTATE 428B2).
- When extending, reducing, or resizing a container, the container type must match the type that was used when the container was created (SQLSTATE 428B2).
- An attempt to change container sizes in the same table space, using separate ALTER TABLESPACE statements but within a single transaction, will raise an error (SQLSTATE 55041).
- In a partitioned database, if more than one database partition exists on the same physical node, the same device or specific path cannot be specified for such database partitions (SQLSTATE 42730). For this environment, either specify a unique *container-string* for each database partition or use a relative path name.
- Although the table space definition is transactional and the changes to the table space definition are reflected in the catalog tables on commit, the buffer pool with the new definition cannot be used until the next time the database is started. The buffer pool that was in use when the ALTER TABLESPACE statement was issued will continue to be used in the interim.
- The REDUCE, RESIZE, or DROP option attempts to free unused extents, if necessary, for DMS table spaces, and the REDUCE option attempts to free unused extents for automatic storage table spaces. The removal of unused extents allows the table space high water mark to be reduced to a value that accurately represents the amount of space used, which, in turn, enables larger reductions in table space size.
- ***Conversion to large DMS table spaces***: After conversion, it is recommended that you issue the COMMIT statement and then increase the storage capacity of the table space.
  - If the table space is enabled for auto-resize, the MAXSIZE table space attribute should be increased, unless it is already set to NONE.
  - If the table space is not enabled for auto-resize, you have two choices:
    - Enable auto-resize by issuing the ALTER TABLESPACE statement with the AUTORESIZE YES option.
    - Add more storage by adding stripe sets, extending the size of existing containers, or both.

  Indexes for tables in a converted table space must be reorganized or rebuilt before they can support large record identifiers (RIDs).
  - The indexes can be rebuilt by using the **REORG INDEXES ALL** command with the REBUILD option. Specify the **ALLOW NO ACCESS** option for partitioned tables.
  - Alternatively, the tables can be reorganized (not INPLACE), which will rebuild all indexes and enable the tables to support more than 255 rows per page.

  To determine which tables do not yet support large RIDs, use the ADMIN_GET_TAB_INFO table function.
- The rebalance of an automatic storage table space that has containers on a storage path in the "Drop Pending" state will drop those containers. New containers might need to be created to hold the data that is being moved off the dropped containers. There must be sufficient free space on the other storage paths in the database to allow those containers to be created, otherwise an error is returned SQLSTATE 57011. The actual amount of free space that is required depends on many factors, including the location of the high-water mark extent and the stripe sets being altered. However, to ensure that the

operation is successful, there should be at least enough free space on the remaining storage paths as space is being consumed by the containers being dropped.

- If the REBALANCE clause is specified but the data server determines that there is no need to create new containers or drop existing ones, a rebalance does not occur and the statement succeeds with a warning (SQLSTATE 01690).
- In addition to adding containers on recently added paths, the REBALANCE operation can also be used to add containers on existing storage paths. Each stripe set in the table space is examined and storage paths that are not in use by a particular stripe set are identified. For each storage path identified, if sufficient free space on it exists, a new container is created. The container will have the same size as the other containers in the stripe set. This would be beneficial if a given storage path ran out of space, table spaces stopped using it (by creating stripe sets on the other paths), and more storage was given to the path. In this case, no new paths have been added, but the rebalance will attempt to include that storage path in stripe sets where it wasn't included before.
- Auto-resize can still occur while a rebalance of an automatic storage table space is in progress.
- When a DMS table space is enabled for automatic storage by the MANAGED BY AUTOMATIC STORAGE clause, that table space has one or more stripe sets of user-defined (non-automatic storage) containers and one or more stripe sets of automatic storage containers. Rebalancing the table space (by using the REBALANCE clause) removes all of the user-defined containers. The database manager might extend existing automatic storage containers or create new automatic storage containers to hold the data being moved from the user-defined containers.
- ***Syntax alternatives***: The following are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - NODE can be specified in place of DBPARTITIONNUM.
  - NODES can be specified in place of DBPARTITIONNUMS.
- For the Db2 Developer-C Edition:
  - Altering an auto-resize table space without specifying **MAXSIZE** will implicitly set **MAXSIZE** to the remaining capacity, up to the defined storage size.
  - An attempt to resize, add, or extend the container size of all table spaces larger than the defined storage size will result in a fail.
  - Altering a table space fails if there exists a subsequent ALTER TABLESPACE that hasn’t been committed.

Examples

- *Example 1:* Add a device to the PAYROLL table space.

```
ALTER TABLESPACE PAYROLL
  ADD (DEVICE '/dev/rhdisk9' 10000)
```

- *Example 2:* Change the prefetch size and I/O overhead for the ACCOUNTING table space.

```
ALTER TABLESPACE ACCOUNTING
  PREFETCHSIZE 64
  OVERHEAD 19.3
```

- *Example 3:* Create a table space TS1, then resize the containers so that all of the containers have 2000 pages. (Three different ALTER TABLESPACE statements that accomplish this resizing are shown.)

```
CREATE TABLESPACE TS1
  MANAGED BY DATABASE
  USING (FILE '/conts/cont0' 1000,
          DEVICE '/dev/rcont1' 500,
          FILE 'cont2' 700)
ALTER TABLESPACE TS1
  RESIZE (FILE '/conts/cont0' 2000,
           DEVICE '/dev/rcont1' 2000,
           FILE 'cont2' 2000)
```

OR

```
ALTER TABLESPACE TS1
  RESIZE (ALL 2000)
```

OR

```
ALTER TABLESPACE TS1
  EXTEND (FILE '/conts/cont0' 1000,
           DEVICE '/dev/rcont1' 1500,
           FILE 'cont2' 1300)
```

- *Example 4:* Extend all of the containers in the DATA_TS table space by 1000 pages.

```
ALTER TABLESPACE DATA_TS
  EXTEND (ALL 1000)
```

- *Example 5:* Resize all of the containers in the INDEX_TS table space to 100 megabytes (MB).

```
ALTER TABLESPACE INDEX_TS
  RESIZE (ALL 100 M)
```

- *Example 6:* Add three new containers. Extend the first container, and resize the second.

```
ALTER TABLESPACE TS0
  ADD (FILE 'cont2' 2000, FILE 'cont3' 2000)
  ADD (FILE 'cont4' 2000)
  EXTEND (FILE 'cont0' 100)
  RESIZE (FILE 'cont1' 3000)
```

- *Example 7:* Table space TSO exists on database partitions 0, 1 and 2. Add a new container to database partition 0. Extend all of the containers on database partition 1. Resize a container on all database partitions other than the ones that were explicitly specified (that is, database partitions 0 and 1).

```
ALTER TABLESPACE TS0
  ADD (FILE 'A' 200) ON DBPARTITIONNUM (0)
  EXTEND (ALL 200) ON DBPARTITIONNUM (1)
  RESIZE (FILE 'B' 500)
```

  The RESIZE clause is the default container clause in this example, and will be executed on database partition 2, because other operations are being explicitly sent to database partitions 0 and 1. However, if there had only been these two database partitions, the statement would have succeeded, but returned a warning (SQL1758W) that default containers had been specified but not used.

- *Example 8:* Enable the auto-resize option for table space DMS_TS1, and set its maximum size to 256 megabytes.

```
ALTER TABLESPACE DMS_TS1
  AUTORESIZE YES MAXSIZE 256 M
```

- *Example 9:* Enable the auto-resize option for table space AUTOSTORE1, and change its growth rate to 5%.

```
ALTER TABLESPACE AUTOSTORE1
  AUTORESIZE YES INCREASESIZE 5 PERCENT
```

- *Example 10:* Change the growth rate for an auto-resizable table space named MY_TS to 512 kilobytes, and set its maximum size to be as large as possible.

```
ALTER TABLESPACE MY_TS
  INCREASESIZE 512 K MAXSIZE NONE
```

- *Example 11:* Enable automatic storage for database-managed table space DMS_TS10 and have it use storage group sg_3.

```
ALTER TABLESPACE DMS_TS10
  MANAGED BY AUTOMATIC STORAGE
  USING STOGROUP sg_3
```

- *Example 12*: An ALTER DATABASE statement removed the paths `/db/filesystem1` and `/db/filesystem2` from the currently connected database. The table spaces named PRODTS1, PRODTS2, and PRODTS3 were the only table spaces using the removed paths. Rebalance these table spaces. Three ALTER TABLESPACE statements must be used.

```
ALTER TABLESPACE PRODTS1 REBALANCE
ALTER TABLESPACE PRODTS2 REBALANCE
ALTER TABLESPACE PRODTS3 REBALANCE
```

- *Example 13:* Enable automatic storage for database-managed table space DATA1 and remove all of the existing non-automatic storage containers from the table space. The first statement must be committed before the second statement can be run.

```
ALTER TABLESPACE DATA1 MANAGED BY AUTOMATIC STORAGE
ALTER TABLESPACE DATA1 REBALANCE
```

- *Example 14:* Trigger extent movement for an automatic storage table space with reclaimable storage attribute, in order to reduce the size of the containers by 10 MB.

```
ALTER TABLESPACE DMS_TS1 REDUCE 10 M
```

- *Example 15:* Trigger extent movement for a non-automatic storage table space with reclaimable storage attribute and then reduce the size of each container by 10 MB.

```
ALTER TABLESPACE TBSP1 LOWER HIGH WATER MARK
ALTER TABLESPACE TBSP1 REDUCE (ALL CONTAINERS 10 M)
```

ALTER THRESHOLD

The ALTER THRESHOLD statement alters the definition of a threshold.

Invocation

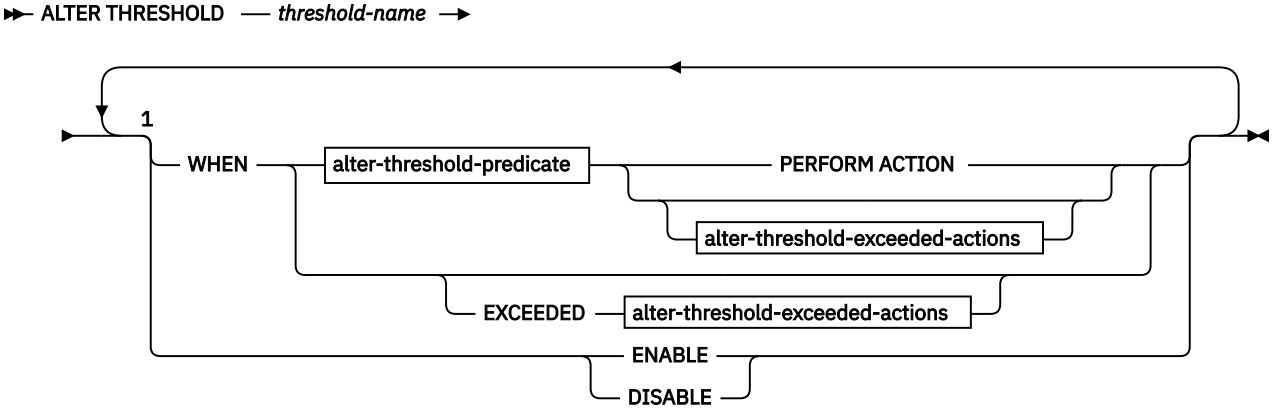
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

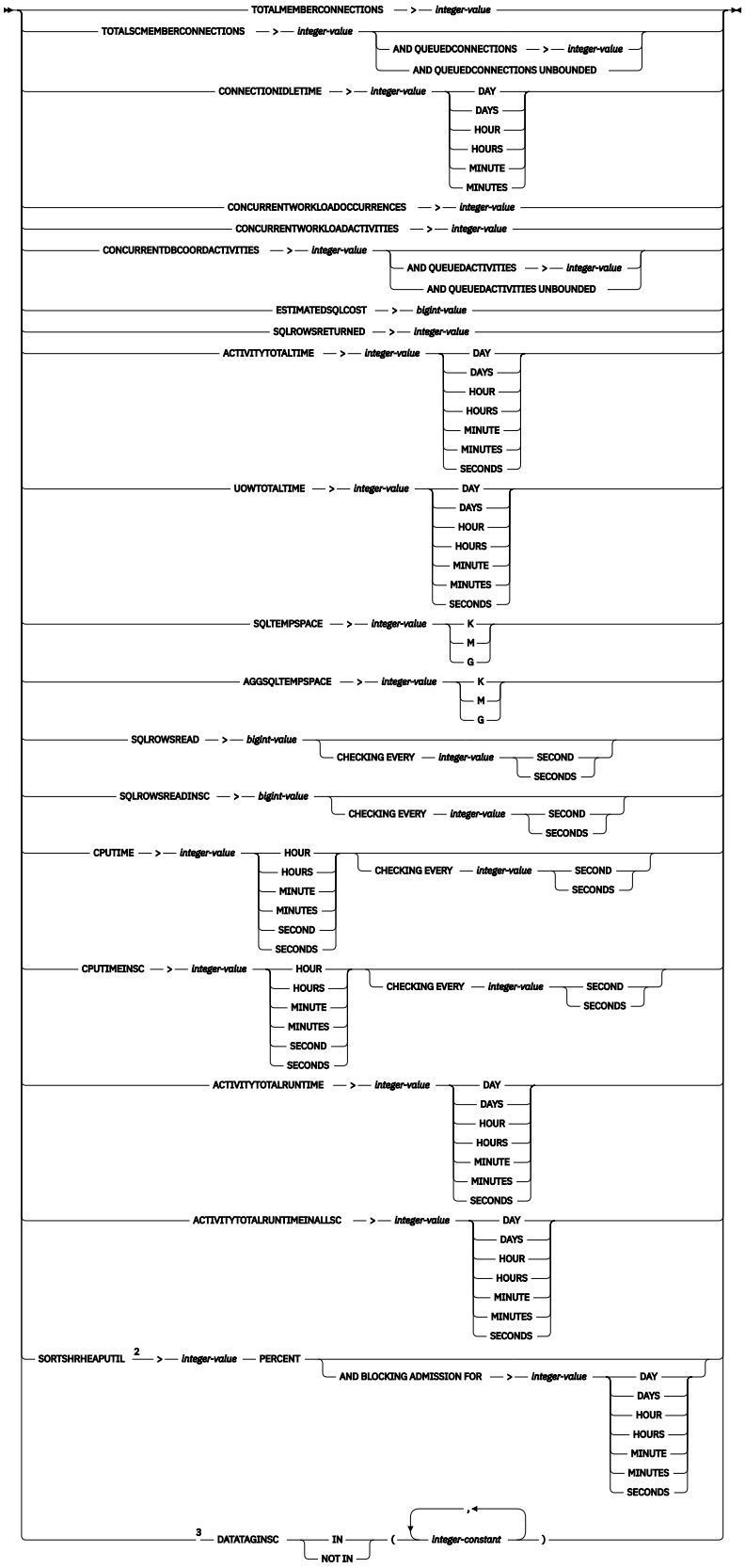
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- SQLADM authority, only if every alteration clause is a COLLECT clause
- WLMADM authority
- DBADM authority

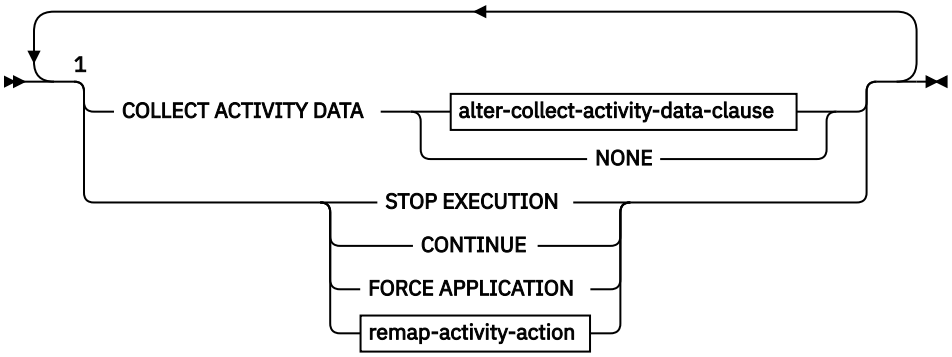
Syntax



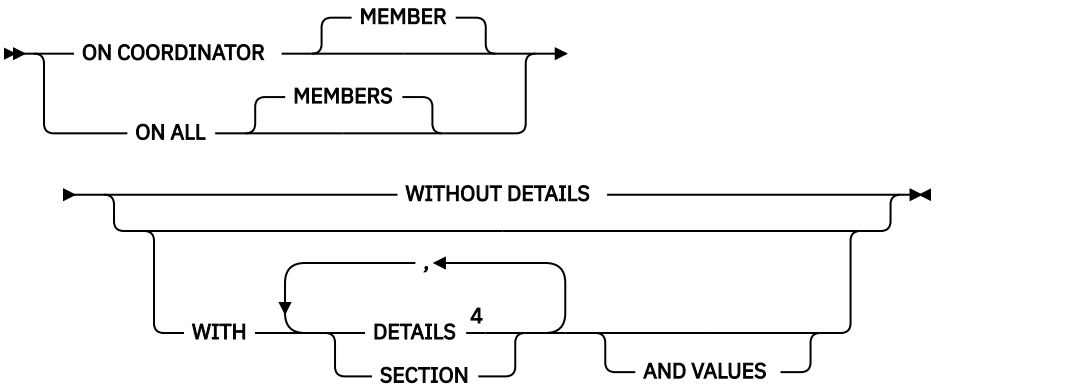
alter-threshold-predicate



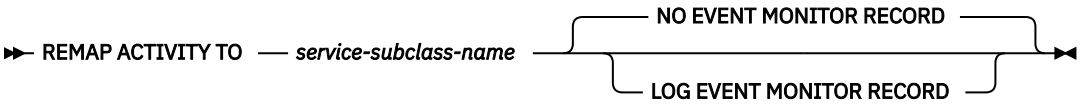
alter-threshold-exceeded-actions



alter-collect-activity-data-clause



remap-activity-action



Notes:

[1] The same clause must not be specified more than once.
[2] This feature is available in Db2 Version 11.5 Mod Pack 2 and later versions.
[3] Each data tag value can be specified only once.
[4] The DETAILS keyword is the minimum to be specified, followed by the option separated by a comma.

Description

threshold-name
Identifies the threshold to be altered. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The name must uniquely identify an existing threshold at the current server (SQLSTATE 42704).

WHEN *alter-threshold-predicate* or WHEN EXCEEDED
Replaces the existing upper bound value in the threshold predicate condition with a new upper bound value. The condition of the threshold cannot be changed to a different one.

PERFORM ACTION
When altering the value of the threshold predicate condition, specifies that the threshold exceeded action is not changed.

EXCEEDED
Specifies to keep the same threshold predicate that was specified originally for this altered threshold.

alter-threshold-predicate

TOTALMEMBERCONNECTIONS > *integer-value*
This condition defines an upper bound on the number of coordinator connections that can run concurrently on a member. This value can be zero or any positive integer (SQLSTATE 42820). A value of zero means that any new coordinator connection will be prevented from connecting. All currently running or queued connections will continue.

TOTALSCMEMBERCONNECTIONS > *integer-value*
This condition defines an upper bound on the number of coordinator connections that can run concurrently on a member in a specific service superclass. This value can be zero or any positive integer (SQLSTATE 42820). A value of zero means that any new connection will be prevented from joining the service class. All currently running or queued connections will continue.

AND QUEUEDCONNECTIONS > *integer-value* or AND QUEUEDCONNECTIONS UNBOUNDED
Specifies a queue size for when the maximum number of coordinator connections is exceeded. This value can be zero or any positive integer (SQLSTATE 42820). A value of zero means that no coordinator connections are queued. Specifying UNBOUNDED will queue every connection that exceeds the specified maximum number of coordinator connections, and the *threshold-exceeded-actions* will never be executed.

CONNECTIONIDLETIME > *integer-value* DAY | DAYS | HOUR | HOURS | MINUTE | MINUTES
This condition defines an upper bound for the amount of time the database manager will allow a connection to remain idle. This value can be any positive integer (not zero) (SQLSTATE 42820). Use a valid duration keyword to specify an appropriate unit of time for *integer-value*. This condition is enforced at the coordinator member.

If you specify the STOP EXECUTION action with CONNECTIONIDLETIME thresholds, the connection for the application is dropped when the threshold is exceeded. Any subsequent attempt by the application to access the data server will receive SQLSTATE 5U026 since the application is no longer connected to the data server.

The maximum value for this threshold is 2 147 483 640 seconds. Any value specified that has a seconds equivalent larger than 2 147 483 640 seconds will be set to this number of seconds.

CONCURRENTWORKLOADOCCURRENCES > *integer-value*
This condition defines an upper bound on the number of concurrent occurrences for the workload on each member. This value can be any positive integer (not zero) (SQLSTATE 42820).

CONCURRENTWORKLOADACTIVITIES > *integer-value*
This condition defines an upper bound on the number of concurrent coordinator activities and nested activities for the workload on each member. This value can be any positive integer (not zero) (SQLSTATE 42820).

Each nested activity must satisfy the following conditions:

- It must be a recognized coordinator activity. Any nested coordinator activity that does not fall within the recognized types of activities will not be counted. Similarly, nested subagent activities, such as remote node requests, are not counted.
- It must be directly invoked from user logic, such as a user-written procedure issuing SQL statements.

Internal SQL activities, such as those generated by the setting of a constraint or the refreshing of a materialized query table, are also not counted by this threshold, because they are initiated by the database manager and not directly invoked by user logic.

CONCURRENTDBCOORDACTIVITIES > *integer-value*
This condition defines an upper bound on the number of recognized database coordinator activities that can run concurrently on all members in the specified domain. This value can be zero or any positive integer (SQLSTATE 42820). A value of zero means that any new database coordinator activities will be prevented from executing. All currently running or queued database coordinator activities will continue. All activities are tracked by this condition, except for the following items:

- CALL statements are not controlled by this threshold, but all nested child activities started within the called routine are under this threshold's control. Anonymous blocks and autonomous routines are classified as CALL statements.
- User-defined functions are controlled by this threshold, but child activities nested in a user-defined function are not controlled. If an autonomous routine is called from within a user defined function, neither the autonomous routine nor any child activities of the autonomous routine are under threshold control.
- Trigger actions that invoke CALL statements and the child activities of these CALL statements are not controlled by this threshold. INSERT, UPDATE, or DELETE statements that can cause a trigger to activate continue to be under threshold control.

Important: Before using CONCURRENTDBCOORDACTIVITIES thresholds, be sure to become familiar with the effects that they can have on the database system.

For more information, refer to "CONCURRENTDBCOORDACTIVITIES threshold" in *Db2 Workload Management Guide and Reference*.

AND QUEUEDACTIVITIES > *integer-value* or AND QUEUEDACTIVITIES UNBOUNDED
Specifies a queue size for when the maximum number of database coordinator activities is exceeded. This value can be zero or any positive integer (SQLSTATE 42820). A value of zero means that no database coordinator activities are queued. Specifying UNBOUNDED will queue every database coordinator activity that exceeds the specified maximum number of database coordinator activities, and the *threshold-exceeded-actions* will never be executed.

Note: If a threshold action of CONTINUE is specified for a queuing threshold, it effectively makes the size of the queue unbounded, regardless of any hard value specified for the queue size.

ESTIMATEDSQLCOST > *bigint-value*
This condition defines an upper bound for the optimizer-assigned cost (in timerons) of an activity. This value can be any positive big integer (not zero) (SQLSTATE 42820). This condition is enforced at the coordinator member. Activities tracked by this condition are:

- Coordinator activities of type data manipulation language (DML).
- Nested DML activities that are invoked from user logic. Consequently, DML activities that can be initiated by the database manager through internal SQL are not tracked by this condition (unless their cost is included in the parent's estimate, in which case they are indirectly tracked).

SQLROWSRETURNED > *integer-value*
This condition defines an upper bound for the number of rows returned to a client application from the application server. This value can be any positive integer (not zero) (SQLSTATE 42820).This condition is enforced at the coordinator member. Activities tracked by this condition are:

- Coordinator activities of type DML.
- Nested DML activities that are derived from user logic. Activities that are initiated by the database manager through internal SQL are not affected by this condition.

Result sets returned from within a procedure are treated separately as individual activities. There is no aggregation of the rows that are returned by the procedure itself.

ACTIVITYTOTALTIME > *integer-value* DAY | DAYS | HOUR | HOURS | MINUTE | MINUTES | SECONDS
This condition defines an upper bound for the amount of time the database manager will allow an activity to execute, including the time the activity was queued. The definition domain for this condition must be DATABASE, work action (a threshold for a work action definition domain is created using a CREATE WORK ACTION SET or ALTER WORK ACTION SET statement, and the work action set must be applied to a workload or a database), SERVICE SUPERCLASS, SERVICE SUBCLASS, or WORKLOAD, and the enforcement scope must be DATABASE (SQLSTATE 5U037). This condition is logically enforced at the coordinator member.

The specified *integer-value* must be an integer that is greater than zero (SQLSTATE 42820). Use a valid duration keyword to specify an appropriate unit of time for *integer-value*. If the specified

time unit is SECONDS, the value must be a multiple of 10 (SQLSTATE 42615). The maximum value that can be specified for this threshold is 2 147 483 640 seconds. If any value (using the DAY, HOUR, MINUTE, or SECONDS time unit) has a seconds equivalent larger than the maximum value, an error is returned (SQLSTATE 42615).

UOWTOTALTIME > *integer-value* DAY | DAYS | HOUR | HOURS | MINUTE | MINUTES | SECONDS
This condition defines an upper bound for the amount of time the database manager will allow a unit of work to execute. This value can be any non-zero positive integer (SQLSTATE 42820). Use a valid duration keyword to specify an appropriate unit of time for *integer-value*. If the specified time unit is SECONDS, the value must be a multiple of 10 (SQLSTATE 42615). This condition is enforced at the coordinator member.

The maximum value that can be specified for this threshold is 2 147 483 640 seconds. If any value (using the DAY, HOUR, MINUTE, or SECONDS time unit) has a seconds equivalent larger than the maximum value, an error is returned (SQLSTATE 42615).

SQLTEMPSPACE > *integer-value* K | M | G
This condition defines the maximum amount of system temporary space that can be consumed by an SQL statement on a member. This value can be any positive integer (not zero) (SQLSTATE 42820).

If *integer-value K* (in either upper- or lowercase) is specified, the maximum size is 1024 times *integer-value*. If *integer-value M* is specified, the maximum size is 1 048 576 times *integer-value*. If *integer-value G* is specified, the maximum size is 1 073 741 824 times *integer-value*.

Activities tracked by this condition are:

- Coordinator activities of type DML and corresponding subagent work (subsection execution).
- Nested DML activities that are derived from user logic and their corresponding subagent work (subsection execution). Activities that are initiated by the database manager through an internal SQL are not affected by this condition.

AGGSQLTEMPSPACE > *integer-value* K | M | G

This condition defines the maximum amount of system temporary space that can be consumed by a set of statements in a service class on a member. This value can be any positive integer (not zero) (SQLSTATE 42820).

If *integer-value* K (in either upper- or lowercase) is specified, the maximum size is 1024 times *integer-value*. If *integer-value* M is specified, the maximum size is 1 048 576 times *integer-value*. If *integer-value* G is specified, the maximum size is 1 073 741 824 times *integer-value*.

Activities contributing to the aggregate that is tracked by this condition are:

- Coordinator activities of type DML and corresponding subagent work like subsection execution.
- Nested DML activities that are derived from user logic and their corresponding subagent work like subsection execution. Activities initiated by the database manager through an internal SQL statement are not affected by this condition.

SQLROWSREAD > *bigint-value*
This condition defines an upper bound on the number of rows that may be read by an activity during its lifetime on a particular member. This value can be any positive big integer (not zero) (SQLSTATE 42820). Note that the number of rows read is different from the number of rows returned, which is controlled by the SQLROWSRETURNED condition.

Activities tracked by this condition are:

- Coordinator activities of type DML and corresponding subagent work (like subsection execution).
- Internal SQL activities like those initiated by the setting of a constraint, or the refreshing of a materialized query table, are also not tracked by this threshold, because they are initiated by the database manager and not directly invoked by user logic.

CHECKING EVERY *integer-value* SECOND | SECONDS
Specifies how frequently the threshold condition is checked for an activity. The threshold is checked at the end of each request (like a fetch operation, for example) and on the interval defined by the CHECKING clause. The CHECKING clause defines an upper bound on how long a threshold violation may go undetected. The value can be any positive integer (not zero) with a maximum value of 86400 seconds (SQLSTATE 42820). Setting a low value may impact system performance negatively.

SQLROWSREADINSC > *bigint-value*
This condition defines an upper bound on the number of rows that may be read by an activity on a particular member while it is executing in a service subclass. Rows read before executing in the service subclass specified are not counted. This value can be any positive big integer (not zero) (SQLSTATE 42820). Note that the number of rows read is different from the number of rows returned, which is controlled by the SQLROWSRETURNED condition.

Activities tracked by this condition are:

- Coordinator activities of type DML and corresponding subagent work (like subsection execution).
- Internal SQL activities like those initiated by the setting of a constraint, or the refreshing of a materialized query table, are also not tracked by this threshold, because they are initiated by the database manager and not directly invoked by user logic.

CHECKING EVERY *integer-value* SECOND | SECONDS
Specifies how frequently the threshold condition is checked for an activity. The threshold is checked at the end of each request (like a fetch operation, for example) and on the interval defined by the CHECKING clause. The CHECKING clause defines an upper bound on how long a threshold violation may go undetected. The value can be any positive integer (not zero) with a maximum value of 86400 seconds (SQLSTATE 42820). Setting a low value may impact system performance negatively.

CPUTIME > *integer-value* DAY | DAYS | HOUR | HOURS | MINUTE | MINUTES | SECOND | SECONDS
This condition defines an upper bound for the amount of processor time that an activity may consume during its lifetime on a particular member. The processor time tracked by this threshold is measured from the time that the activity starts executing. This value can be any positive integer (not zero) (SQLSTATE 42820).

Activities tracked by this condition are:

- Coordinator activities of type DML and corresponding subagent work (like subsection execution).
- Internal SQL activities, like those initiated by the setting of a constraint or the refreshing of a materialized query table, are also not tracked by this threshold, because they are initiated by the database manager and not directly invoked by user logic.
- Activities of type CALL. For CALL activities, the processor time tracked for the procedure does not include the processor time used by any child activity or by any fenced mode processes. The threshold condition will be checked only upon return from user logic to the database engine. For example: During execution of a trusted routine, the threshold condition will be checked only when the routine issues a request to the database engine.

CHECKING EVERY *integer-value* SECOND | SECONDS
Specifies how frequently the threshold condition is checked for an activity. The granularity of the CPUTIME threshold is approximately this number multiplied by the degree of parallelism for the activity. For example: If the threshold is checked every 60 seconds and the degree of parallelism is 2, the activity might use an extra 2 minutes of processor time instead of 1 minute before the threshold violation is detected. The value can be any positive integer (not zero) with a maximum value of 86400 seconds (SQLSTATE 42820). Setting a low value may impact system performance negatively.

CPUTIMEINSC > *integer-value* DAY | DAYS | HOUR | HOURS | MINUTE | MINUTES | SECOND | SECONDS
This condition defines an upper bound for the amount of processor time that an activity may consume on a particular member while it is executing in a service subclass. The processor time

tracked by this threshold is measured from the time that the activity starts executing in the service subclass identified in the threshold domain. Any processor time used before that point is not counted toward the limit imposed by this threshold. This value can be any positive integer (not zero) (SQLSTATE 42820).

Activities tracked by this condition are:

- Coordinator activities of type DML and corresponding subagent work (like subsection execution).
- Internal SQL activities, like those initiated by the setting of a constraint or the refreshing of a materialized query table, are also not tracked by this threshold, because they are initiated by the database manager and not directly invoked by user logic.
- Activities of type CALL. For CALL activities, the processor time tracked for the procedure does not include the processor time used by any child activity or by any fenced mode processes. The threshold condition will be checked only upon return from user logic to the database engine. For example: During execution of a trusted routine, the threshold condition will be checked only when the routine issues a request to the database engine.

CHECKING EVERY *integer-value* SECOND | SECONDS
Specifies how frequently the threshold condition is checked for an activity. The granularity of the CPUTIMEINSC threshold is approximately this number multiplied by the degree of parallelism for the activity. For example: If the threshold is checked every 60 seconds and the degree of parallelism is 2, the activity might use an extra 2 minutes of processor time instead of 1 minute before the threshold violation is detected. The value can be any positive integer (not zero) with a maximum value of 86400 seconds (SQLSTATE 42820). Setting a low value may impact system performance negatively.

ACTIVITYTOTALRUNTIME > *integer-value* DAY | DAYS | HOUR | HOURS | MINUTE | MINUTES | SECONDS
This condition is used to define an upper bound for the amount of time the database manager allows an activity to run. The amount of time does not include the time that the activity was queued by a WLM concurrency threshold. The definition domain for this condition must be one of the following thresholds (SQLSTATE 5U037):

- Database
- Service superclass
- Service subclass
- Statement
- Workload
- Work action [1]

1. A threshold for a work action definition domain is created by using a CREATE WORK ACTION SET or ALTER WORK ACTION SET statement. The work action set must be applied to a workload or a database.

The enforcement scope must be DATABASE (SQLSTATE 5U037).

The specified *integer-value* must be an integer that is greater than zero (SQLSTATE 42820). Use a valid duration keyword to specify an appropriate unit of time for *integer-value*. If the specified time unit is SECONDS, the value must be a multiple of 10 (SQLSTATE 42615). The maximum value that can be specified for this threshold is 2 147 483 640 seconds. If any value for the DAY, HOUR, MINUTE, or SECONDS time unit has a seconds equivalent larger than the maximum value, an error is returned (SQLSTATE 42615).

ACTIVITYTOTALRUNTIMEINALLSC > *integer-value* DAY | DAYS | HOUR | HOURS | MINUTE | MINUTES | SECONDS
This condition is used to define an upper bound for the amount of time the database manager allows an activity to run. The amount of time does not include the time that the activity was queued by a WLM concurrency threshold. The execution time that is tracked by this threshold is measured from the time that the activity starts running.

The definition domain for this condition must be service subclass (SERVICE CLASS specifying the UNDER clause), and the enforcement scope must be DATABASE (SQLSTATE 5U037).

The specified *integer-value* must be an integer that is greater than zero (SQLSTATE 42820). Use a valid duration keyword to specify an appropriate unit of time for *integer-value*. If the specified time unit is SECONDS, the value must be a multiple of 10 (SQLSTATE 42615). The maximum value that can be specified for this threshold is 2 147 483 640 seconds. If any value for the DAY, HOUR, MINUTE, or SECONDS time unit has a seconds equivalent larger than the maximum value, an error is returned (SQLSTATE 42615).

SORTSHRHEAPUTIL > *integer-value* PERCENT

Attention: This feature is available in Db2 Version 11.5 Mod Pack 2 and later versions.

This condition defines the maximum shared sort memory that may be requested by a query as a percentage of the total database shared sort memory (sheapthres_shr). When the adaptive workload manager is enabled, the threshold considers both estimated and actual memory requirements for a query. Any positive integer between 1 to 100 can be specified as a percent value.

Activities tracked by this condition are:

- Coordinator activities of type data manipulation language (DML).
- Nested DML activities that are directly invoked from user logic.

AND BLOCKING ADMISSION FOR *integer-value*

Specifies that action will only be taken if the sort memory requirements are exceeded, work is currently queued behind the violating activity and WLM admission control has not admitted any requests for the specified amount of time. For work inside the WLM admission queue, this condition will only be evaluated once a request reaches the front of the admission queue. Every time a request is allowed by admission control, the queue time will be reset. If multiple requests violate this threshold a cascading effect will be observed until something that doesn't violate this threshold is found or the last request is reached (as in, no other requests behind).

The maximum value for this threshold is 2147483640 seconds. Any value specified that has a seconds equivalent larger than 2147483640 seconds will be set to this number of seconds. The time specified has a minimum accuracy of 10 seconds, so any value specified is subject to accuracy of this amount. A value of zero is equivalent to not specifying a BLOCKING ADMISSION FOR clause.

DATATAGINSC IN *(integer-constant, ...)*

This condition defines one or more data tag values specified on a table space that the activity touches. The data tag on a table space, or its underlying storage group (where applicable), can be either not be set or set to a value from 1 to 9. If the activity touches a table space that has no data tag set (either at the table space or storage group level), this threshold will not have any affect on that activity. The definition domain for this condition must be a service subclass (SERVICE CLASS specifying the UNDER clause), and the enforcement scope must be DATABASE PARTITION (SQLSTATE 5U037). This condition is enforced independently at each database partition.

Activities tracked by this condition are:

- Coordinator activities of type data manipulation language (DML).
- Nested DML activities that are directly invoked from user logic.

DML activities that can be initiated by the database manager through internal SQL are not tracked by this condition.

This threshold is only checked when a scan is opened on a table or when an insert is performed into a table. Fetching data from a table after a scan has been opened will not violate the threshold.

DATATAGINSC NOT IN *(integer-constant, ...)*

This condition defines one or more data tag values not specified on a table space that the activity touches. The data tag on a table space, or its underlying storage group (where applicable), can be

either not be set or set to a value from 1 to 9. If the activity touches a table space that has no data tag set (either at the table space or storage group level), this threshold will not have any affect on that activity. The definition domain for this condition must be a service subclass (SERVICE CLASS specifying the UNDER clause), and the enforcement scope must be DATABASE PARTITION (SQLSTATE 5U037). This condition is enforced independently at each database partition.

Activities tracked by this condition are:

- Coordinator activities of type data manipulation language (DML).
- Nested DML activities that are directly invoked from user logic.

DML activities that can be initiated by the database manager through internal SQL are not tracked by this condition.

This threshold is only checked when a scan is opened on a table or when an insert is performed into a table. Fetching data from a table after a scan has been opened will not violate the threshold.

alter-threshold-exceeded-actions

Specifies what action is to be taken when a condition is exceeded. Each time that a condition is exceeded, an event is recorded in all active threshold violations event monitors.

COLLECT ACTIVITY DATA

Specifies that data about each activity that exceeded the threshold is to be sent to any active activities event monitor when the activity completes. The COLLECT ACTIVITY DATA setting does not apply to non-activity thresholds, such as CONNECTIONIDLETIME, TOTALDBPARTITIONCONNECTIONS, TOTALSCPARTITIONCONNECTIONS, CONCURRENTWORKLOADOCCURRENCES, or UOWTOTALTIME.

alter-collect-activity-data-clause

ON COORDINATOR MEMBER

Specifies that the activity data is to be collected only at the coordinator member of the activity.

ON ALL MEMBERS

Specifies that the activity data is to be collected at all members on which the activity is processed. On remote members, a record for the activity may be captured multiple times as the activity comes and goes on those members. For predictive thresholds, activity information is collected at all members only if you also specify the CONTINUE action for exceeded thresholds. For reactive thresholds, the ON ALL MEMBERS clause has no effect and activity information is always collected only at the coordinator member. For both predictive and reactive thresholds, any input data values, section information, or values will be collected only at the coordinator member.

WITHOUT DETAILS

Specifies that data about each activity associated with the work class for which this work action is defined should be sent to any active activities event monitor, when the activity completes execution. Details about statement, compilation environment, and section environment data are not sent.

WITH

DETAILS

Specifies that statement and compilation environment data is to be sent to any active activities event monitor, for those activities that have them. Section environment data is not sent.

SECTION

Specifies that statement, compilation environment, section environment data, and section actuals are to be sent to any active activities event monitor for those activities that have them. DETAILS must be specified if SECTION is specified. For predictive thresholds, section actuals will be collected on any member where the activity data is collected. For reactive thresholds, section actuals will be collected only on the coordinator member.

AND VALUES
Specifies that input data values are to be sent to any active activities event monitor, for those activities that have them. This data does not include SQL statements that are compiled by using the REOPT ALWAYS bind option.

NONE
Specifies that activity data should not be collected for each activity that exceeds the threshold.

STOP EXECUTION
The execution of the activity is stopped and an error is returned (SQLSTATE 5U026). In the case of the UOWTOTALTIME threshold, the unit of work is rolled back.

CONTINUE
The execution of the activity is not stopped. When the condition also has a queue, this option causes queuing to extend beyond the size of the queue.

FORCE APPLICATION
The application is forced off the system (SQLSTATE 55032). This action can only be specified for the UOWTOTALTIME threshold.

remap-activity-action

REMAP ACTIVITY TO *service-subclass-name*
The activity is mapped to *service-subclass-name*. The execution of the activity is not stopped. This action is valid only for in-service-class and in-all-service-class thresholds like CPUTIMEINSC, SQLROWSREADINSC, DATATAGINSC IN and DATATAGINSC NOT IN and ACTIVITYTOTALRUNTIMEINALLSC thresholds (SQLSTATE 5U037). The service-subclass-name must identify an existing service subclass under the same superclass associated with the threshold (SQLSTATE 5U037). The service-subclass-name cannot be the same as the associated service subclass of the threshold (SQLSTATE 5U037).

NO EVENT MONITOR RECORD
Specifies that no threshold violation record will be written.

LOG EVENT MONITOR RECORD
Specifies that if a THRESHOLD VIOLATIONS event monitor exists and is active, a threshold violation record is written to it.

ENABLE or DISABLE
Specifies whether or not the threshold is enabled for use by the database manager.

ENABLE
The threshold is used by the database manager to restrict the execution of database activities. Currently running database activities will continue to execute without the restriction of this threshold.

DISABLE
The threshold is not used by the database manager to restrict the execution of database activities. New database activities will not be restricted by this threshold. Thresholds with a queue, for example TOTALSCMEMBERCONNECTIONS or CONCURRENTDBCOORDACTIVITIES, must be disabled before they can be dropped.

Notes

- Thresholds can be defined on different aspects of database behavior to monitor and control that behavior. When a threshold is defined on activities, unless otherwise specified, it will be enforced only during the actual execution of SQL statements, not including compilation time, and the load utility.
- The CONCURRENTWORKLOADOCCURRENCES threshold and the CONCURRENTWORKLOADACTIVITIES threshold differ in scope. CONCURRENTWORKLOADOCCURRENCES controls how many connections can map to a workload definition simultaneously, and CONCURRENTWORKLOADACTIVITIES controls how many activities each connection that is mapped to the workload definition can submit concurrently.
- Changes are written to the system catalog, but do not take effect until after a COMMIT statement, even for the connection that issues the statement.

- Only one uncommitted WLM-exclusive SQL statement at a time is allowed across all partitions. If an uncommitted WLM-exclusive SQL statement is executing, subsequent WLM-exclusive SQL statements will wait until the current WLM-exclusive SQL statement commits or rolls back.
- The new value for a threshold affects only database activities that start executing after the alter operation commits.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - DATABASE PARTITION can be specified in place of MEMBER, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
  - DATABASE PARTITIONS can be specified in place of MEMBERS, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
  - TOTALDBPARTITIONCONNECTIONS can be specified in place of TOTALMEMBERCONNECTIONS, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
  - TOTALSCPARTITIONCONNECTIONS can be specified in place of TOTALSCMEMBERCONNECTIONS, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.

Example

Alter the threshold MAXBIGQUERIESCONCURRENCY to a maximum of three activities rather than two.

```
ALTER THRESHOLD MAXBIGQUERIESCONCURRENCY
  WHEN CONCURRENTDBCOORDACTIVITIES > 3
  STOP EXECUTION
```

Because this is a threshold with a queue, the threshold cannot be dropped unless it is disabled, as follows:

```
ALTER THRESHOLD MAXBIGQUERIESCONCURRENCY DISABLE
```

ALTER TRIGGER

The ALTER TRIGGER statement changes the description of a trigger at the current server.

Invocation

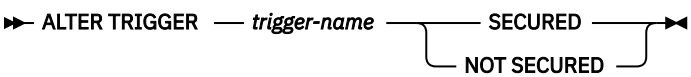
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following privileges:

- SECADM authority
- CREATE_SECURE_OBJECT authority

Syntax



Description

trigger-name
Identifies the trigger to be altered. The *trigger-name* must identify a trigger that exists at the current server (SQLSTATE 42704).

NOT SECURED or SECURED
Specifies whether the trigger is considered secure.

SECURED
Specifies the trigger is considered secure. SECURED must be specified for a trigger whose subject table is a table on which row level or column level access control has been activated (SQLSTATE 428H8). Similarly, SECURED must be specified for a trigger that is created on a view and one or more of the underlying tables in that view definition has row level or column level access control activated (SQLSTATE 428H8).

NOT SECURED
Specifies the trigger is considered not secure. Altering a trigger from secured to not secured fails if the trigger is defined on a table for which row or column level access control is activated (SQLSTATE 428H8). Similarly, altering a trigger from secured to not secured fails if the trigger is defined on a view and one or more of the underlying tables in that view definition has row or column level access control activated (SQLSTATE 428H8).

Examples

- *Example 1*: Alter trigger TRIGGER1 to SECURED.

```
ALTER TRIGGER TRIGGER1 SECURED
```

- *Example 2*: Alter trigger TRIGGER1 to NOT SECURED.

```
ALTER TRIGGER TRIGGER1 NOT SECURED
```

ALTER TRUSTED CONTEXT

The ALTER TRUSTED CONTEXT statement modifies the definition of a trusted context at the current server.

Important: The DATA_ENCRYPT authentication type is deprecated and might be removed in a future release. To encrypt data in-transit between clients and Db2 databases, we recommend that you use the Db2 database system support of Transport Layer Security (TLS). For more information, see Encryption of data in transit

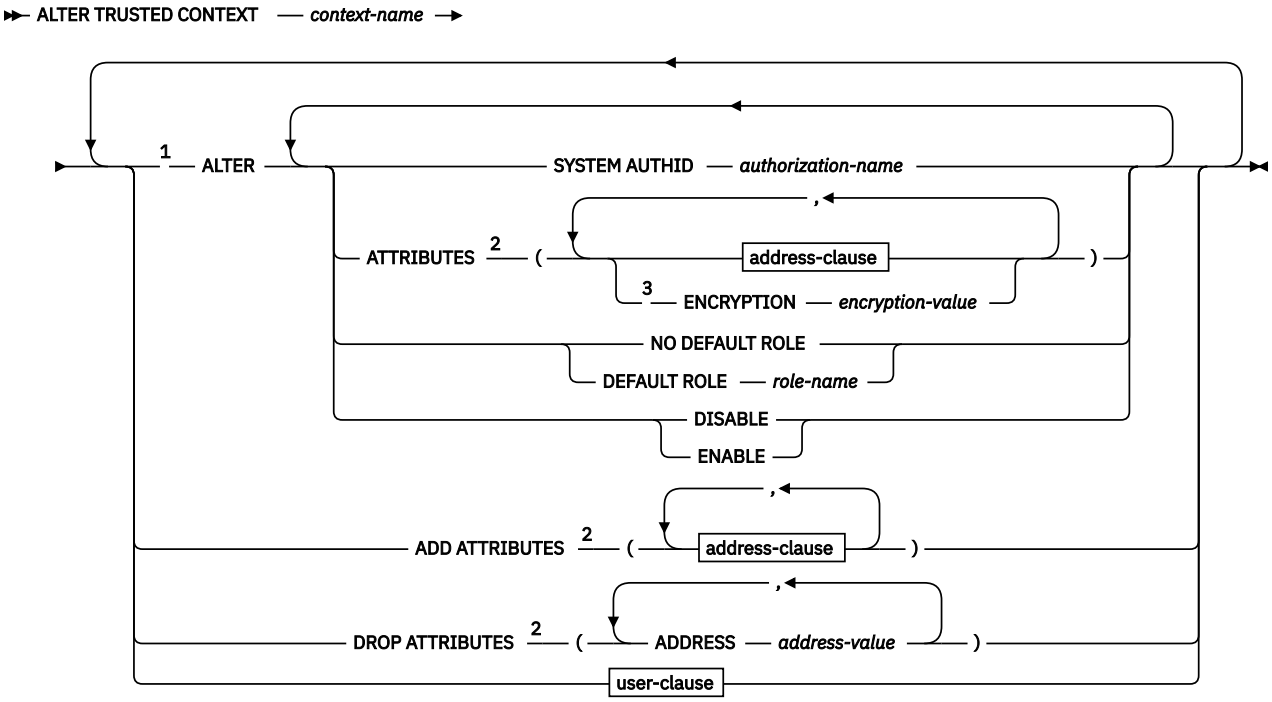
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

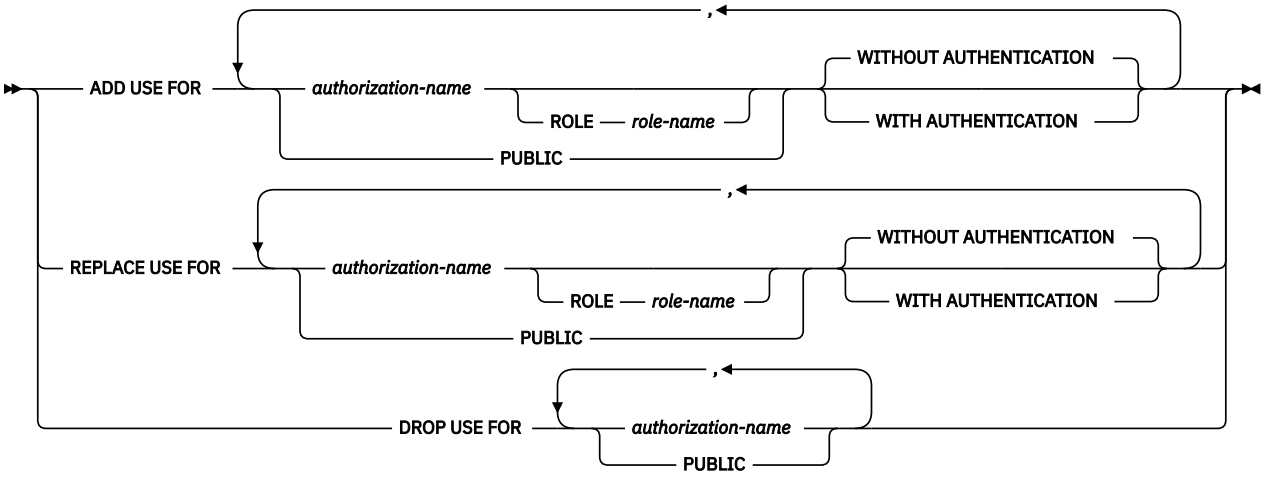
The privileges held by the authorization ID of the statement must include SECADM authority.

Syntax



address-clause

user-clause



Notes:

[1] Each of the ATTRIBUTES, DEFAULT ROLE, ENABLE, and WITH USE clauses can be specified at most once (SQLSTATE 42614).

[2] Each attribute name and corresponding value must be unique (SQLSTATE 4274D).

[3] ENCRYPTION cannot be specified more than once (SQLSTATE 42614); however, WITH ENCRYPTION can be specified for each ADDRESS that is specified.

Description

context-name
Identifies the trusted context that is to be altered. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *context-name* must identify a trusted context that exists at the current server (SQLSTATE 42704).

ALTER
Alters the options and attributes of a trusted context.

SYSTEM AUTHID *authorization-name*
Specifies that the context is a connection established by system authorization ID *authorization-name*, which must not be associated with an existing trusted context (SQLSTATE 428GL). It cannot be the authorization ID of the statement (SQLSTATE 42502).

ATTRIBUTES (...)
Specifies a list of one or more connection trust attributes, upon which the trusted context is defined, that are to be modified. Existing values for the specified attributes are replaced with the new values. If an attribute is not currently part of the trusted context definition, an error is returned (SQLSTATE 4274C). Attributes that are not specified retain their previous values.

ADDRESS *address-value*
Specifies the actual communication address used by the client to communicate with the database server. The only protocol supported is TCP/IP. Previous ADDRESS values for the specified trusted context are removed. The ADDRESS attribute can be specified multiple times, but each *address-value* pair must be unique for the set of attributes (SQLSTATE 4274D).

When establishing a trusted connection, if multiple values are defined for the ADDRESS attribute of a trusted context, a candidate connection is considered to match this attribute if the address used by the connection matches any of the defined values for the ADDRESS attribute of the trusted context.

address-value
Specifies a string constant that contains the value to be associated with the ADDRESS trust attribute. The *address-value* must be an IPv4 address, an IPv6 address, or a secure domain name.

- An IPv4 address must not contain leading spaces and is represented as a dotted decimal address. An example of an IPv4 address is 9.112.46.111. The value 'localhost' or its equivalent representation '127.0.0.1' will not result in a match; the real IPv4 address of the host must be specified instead.
- An IPv6 address must not contain leading spaces and is represented as a colon hexadecimal address. An example of an IPv6 address is 2001:0DB8:0000:0000:0008:0800:200C:417A. IPv4-mapped IPv6 addresses (for example, ::ffff:192.0.2.128) will not result in a match. Similarly, 'localhost' or its IPv6 short representation '::1' will not result in a match.
- A domain name is converted to an IP address by the domain name server where a resulting IPv4 or IPv6 address is determined. An example of a domain name is corona.torolab.ibm.com. When a domain name is converted to an IP address, the result of this conversion could be a set of one or more IP addresses. In this case, an incoming connection is said to match the ADDRESS attribute of a trusted context object if the IP address from which the connection originates matches any of the IP addresses to which the domain name was converted. When creating a trusted context object, it is advantageous to provide domain name values for the ADDRESS attribute instead of static IP addresses, particularly in Dynamic Host Configuration Protocol (DHCP) environments. With DHCP, a device can have a different IP address each time it connects to the network. So, if a static IP address is provided for the ADDRESS attribute of a trusted context object, some device might acquire a trusted connection unintentionally. Providing domain names for the ADDRESS attribute of a trusted context object avoids this problem in DHCP environments.

WITH ENCRYPTION *encryption-value*
Specifies the minimum level of encryption of the data stream or network encryption for this specific *address-value*. This *encryption-value* overrides the global ENCRYPTION attribute setting for this specific *address-value*.

encryption-value
Specifies a string constant that contains the value to be associated with the ENCRYPTION trust attribute for this specific *address-value*. The *encryption-value* must be one of the following values (SQLSTATE 42615):

- NONE, no specific level of encryption is required
- LOW, a minimum of light encryption is required; the authentication type on the database manager must be DATA_ENCRYPT if an incoming connection is to match the encryption setting for this specific address
- HIGH, Secure Sockets Layer (SSL) encryption, or equivalent, must be used for data communication between the database client and the database server if an incoming connection is to match the encryption setting for this specific address

ENCRYPTION *encryption-value*
Specifies the minimum level of encryption of the data stream or network encryption. The default is NONE.

encryption-value
Specifies a string constant that contains the value to be associated with the ENCRYPTION trust attribute for this specific *address-value*. The *encryption-value* must be one of the following values (SQLSTATE 42615):

- NONE, no specific level of encryption is required for an incoming connection to match the ENCRYPTION attribute of this trusted context object
- LOW, a minimum of light encryption is required; the authentication type on the database manager must be DATA_ENCRYPT if an incoming connection is to match the ENCRYPTION attribute of this trusted context object
- HIGH, Secure Sockets Layer (SSL) encryption, or equivalent, must be used for data communication between the database client and the database server if an incoming connection is to match the ENCRYPTION attribute of this trusted context object

For details about the ENCRYPTION trust attribute, see "CREATE TRUSTED CONTEXT".

NO DEFAULT ROLE or DEFAULT ROLE *role-name*
Specifies whether or not a default role is associated with a trusted connection that is based on this trusted context. If a trusted connection for this context is active, the change comes into effect on the next switch user request or a new connection request.

NO DEFAULT ROLE
Specifies that the trusted context does not have a default role.

DEFAULT ROLE *role-name*
Specifies that *role-name* is the default role for the trusted context. The *role-name* must identify a role that exists at the current server (SQLSTATE 42704). This role is used with the user in a trusted connection, based on this trusted context, when the user does not have a user-specific role defined as part of the definition of the trusted context.

ENABLE or DISABLE
Specifies whether the trusted context is enabled or disabled.

ENABLE
Specifies that the trusted context is enabled.

DISABLE
Specifies that the trusted context is disabled. A trusted context that is disabled is not considered when a trusted connection is established.

ADD ATTRIBUTES
Specifies a list of one or more additional trust attributes on which the trusted context is defined.

ADDRESS *address-value*

Specifies the actual communication address used by the client to communicate with the database server. The only protocol supported is TCP/IP. The ADDRESS attribute can be specified multiple times, but each *address-value* pair must be unique for the set of attributes (SQLSTATE 4274D).

When establishing a trusted connection, if multiple values are defined for the ADDRESS attribute of a trusted context, a candidate connection is considered to match this attribute if the address used by the connection matches any of the defined values for the ADDRESS attribute of the trusted context.

address-value

Specifies a string constant that contains the value to be associated with the ADDRESS trust attribute. The *address-value* must be an IPv4 address, an IPv6 address, or a secure domain name.

- An IPv4 address must not contain leading spaces and is represented as a dotted decimal address. An example of an IPv4 address is 9.112.46.111. The value 'localhost' or its equivalent representation '127.0.0.1' will not result in a match; the real IPv4 address of the host must be specified instead.
- An IPv6 address must not contain leading spaces and is represented as a colon hexadecimal address. An example of an IPv6 address is 2001:0DB8:0000:0000:0008:0800:200C:417A. IPv4-mapped IPv6 addresses (for example, ::ffff:192.0.2.128) will not result in a match. Similarly, 'localhost' or its IPv6 short representation '::1' will not result in a match.
- A domain name is converted to an IP address by the domain name server, where a resulting IPv4 or IPv6 address is determined. An example of a domain name is corona.torolab.ibm.com.

WITH ENCRYPTION *encryption-value*

Specifies the minimum level of encryption of the data stream or network encryption for this specific *address-value*. This *encryption-value* overrides the global ENCRYPTION attribute setting for this specific *address-value*.

encryption-value

Specifies a string constant that contains the value to be associated with the ENCRYPTION trust attribute for this specific *address-value*. The *encryption-value* must be one of the following values (SQLSTATE 42615):

- NONE, no specific level of encryption is required
- LOW, a minimum of light encryption is required; the authentication type on the database manager must be DATA_ENCRYPT if an incoming connection is to match the encryption setting for this specific address
- HIGH, Secure Sockets Layer (SSL) encryption, or equivalent, must be used for data communication between the database client and the database server if an incoming connection is to match the ENCRYPTION attribute of this trusted context object

DROP ATTRIBUTES

Specifies that one or more attributes are to be dropped from the definition of the trusted context. If the attribute and attribute value pair is not currently part of the trusted context definition, an error is returned (SQLSTATE 4274C).

ADDRESS *address-value*

Specifies that the identified communication address is to be removed from the definition of the trusted context. The *address-value* specifies a string constant that contains the value of an existing ADDRESS trust attribute.

ADD USE FOR

Specifies additional users who can use a trusted connection based on this trusted context. If the definition of a trusted context allows access by PUBLIC and a list of users, the specifications for a user override the specifications for PUBLIC.

authorization-name
Specifies that the trusted connection can be used by the specified *authorization-name*. The *authorization-name* must not identify an authorization ID that is already defined to use the trusted context, and must not be specified more than once in the ADD USE FOR clause (SQLSTATE 428GM). It must also not be the authorization ID of the statement (SQLSTATE 42502).

ROLE *role-name*
Specifies that *role-name* is the role to be used for the user. The *role-name* must identify a role that exists at the current server (SQLSTATE 42704). The role explicitly specified for the user overrides any default role associated with the trusted context.

PUBLIC
Specifies that a trusted connection that is based on this trusted context can be used by any user. PUBLIC must not already be defined to use the trusted context, and PUBLIC must not be specified more than once in the ADD USE FOR clause (SQLSTATE 428GM).

WITHOUT AUTHENTICATION or WITH AUTHENTICATION
Specifies whether or not switching the current user on a trusted connection based on this trusted context requires authentication.

WITHOUT AUTHENTICATION
Specifies that switching the current user on a trusted connection based on this trusted context to this user does not require authentication.

WITH AUTHENTICATION
Specifies that switching the current user on a trusted connection based on this trusted context to this user requires authentication.

REPLACE USE FOR
Specifies that the way in which a particular user or PUBLIC uses the trusted context is to change.

authorization-name
Specifies the *authorization-name* of the user whose use of the trusted connection is to change. The trusted context must already be defined to allow use by the *authorization-name* (SQLSTATE 428GN), and *authorization-name* must not be specified more than once in the REPLACE USE FOR clause (SQLSTATE 428GM). It must also not be the authorization ID of the statement (SQLSTATE 42502).

ROLE *role-name*
Specifies that *role-name* is the role for the user. The *role-name* must identify a role that exists at the current server (SQLSTATE 42704). The role explicitly specified for the user overrides any default role associated with the trusted context.

PUBLIC
Specifies that the attributes for use of the trusted connection by PUBLIC are to change. The trusted context must already be defined to allow use by PUBLIC (SQLSTATE 428GN), and PUBLIC must not be specified more than once in the REPLACE USE FOR clause (SQLSTATE 428GM).

WITHOUT AUTHENTICATION or WITH AUTHENTICATION
Specifies whether or not switching the current user on a trusted connection based on this trusted context requires authentication.

WITHOUT AUTHENTICATION
Specifies that switching the current user on a trusted connection based on this trusted context to this user does not require authentication.

WITH AUTHENTICATION
Specifies that switching the current user on a trusted connection based on this trusted context to this user requires authentication.

DROP USE FOR
Specifies who can no longer use the trusted context. The users who are removed from the definition of the trusted context are those users who are currently allowed to use the trusted context. If one or more, but not all, users can be removed from the definition of the trusted context, the specified

users are removed and a warning is returned (SQLSTATE 01682). If none of the specified users can be removed from the definition of the trusted context, an error is returned (SQLSTATE 428GN).

authorization-name
Removes the ability of the specified authorization ID to use this trusted context.

PUBLIC
Removes the ability of all users (except the system authorization ID and individual authorization IDs that have been explicitly enabled) to use this trusted context.

Rules

- A trusted context-exclusive SQL statement must be followed by a COMMIT or a ROLLBACK statement (SQLSTATE 5U021). Trusted context-exclusive SQL statements are:
  - CREATE TRUSTED CONTEXT, ALTER TRUSTED CONTEXT, or DROP (TRUSTED CONTEXT)
- A trusted context-exclusive SQL statement cannot be issued within a global transaction; for example, an XA transaction or a global transaction that is initiated as part of two-phase commit for federated transactions (SQLSTATE 51041).

Notes

- When providing an IP address as part of a trusted context definition, the address must be in the format that is in effect for the network. For example, providing an address in an IPv6 format when the network is IPv4 will not result in a match. In a mixed environment, it is advantageous to specify both the IPv4 and the IPv6 representations of the address, or better yet, to specify a secure domain name (for example, corona.torolab.ibm.com), which hides the address format details.
- Only one uncommitted trusted context-exclusive SQL statement is allowed at a time across all database partitions. If an uncommitted trusted context-exclusive SQL statement is executing, subsequent trusted context-exclusive SQL statements will wait until the current trusted context-exclusive SQL statement commits or rolls back.
- Changes are written to the system catalog but do not take effect until they are committed, even for the connection that issues the statement.
- ***Order of operations:*** The order of operations within an ALTER TRUSTED CONTEXT statement is:
  - DROP
  - ALTER
  - ADD ATTRIBUTES
  - ADD USE FOR
  - REPLACE USE FOR
- ***Effect of changes on existing trusted connections:*** If trusted connections exist for the trusted context being altered, the connections remain trusted with the definition in effect before the ALTER TRUSTED CONTEXT statement until the next switch user request or the connection terminates. If the trusted context is disabled while trusted connections for this context are active, the connections remain trusted until the next switch user request or the connection terminates. If trust attributes are changed with the ALTER TRUSTED CONTEXT statement, trusted connections that exist at the time of the ALTER TRUSTED CONTEXT statement that use the trusted context are allowed to continue.
- ***Role privileges:*** If there is no role associated with the user or the trusted context, only the privileges associated with the user are applicable. This is the same as not being in a trusted context.

Examples

- *Example 1:* Assume that trusted context APPSERVER exists and that it is enabled. Issue an ALTER TRUSTED CONTEXT statement to allow Bill to use the trusted context APPSERVER, but put the trusted context in the disabled state.

```
ALTER TRUSTED CONTEXT APPSERVER
  DISABLE
  ADD USE FOR BILL
```

- *Example 2:* Assume that trusted context SECUREROLE exists. Issue an ALTER TRUSTED CONTEXT statement to modify the existing user Joe to use the trusted context with authentication and to add everyone else to use the trusted context without authentication.

```
ALTER TRUSTED CONTEXT SECUREROLE
  REPLACE USE FOR JOE WITH AUTHENTICATION
  ADD USE FOR PUBLIC WITHOUT AUTHENTICATION
```

- *Example 3:* Assume that trusted context SECUREROLEENCRYPT exists with ADDRESS attribute values '9.13.55.100' and '9.12.30.112', and ENCRYPTION attribute value 'NONE'. Issue an ALTER statement to modify the ADDRESS attribute values and the encryption attribute to 'LOW'.

```
ALTER TRUSTED CONTEXT SECUREROLEENCRYPT
  ALTER ATTRIBUTES (ADDRESS '9.12.155.200',
    ENCRYPTION 'LOW')
```

ALTER TYPE (structured)

The ALTER TYPE statement is used to add or drop attributes or method specifications of a user-defined structured type. Properties of existing methods can also be altered.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

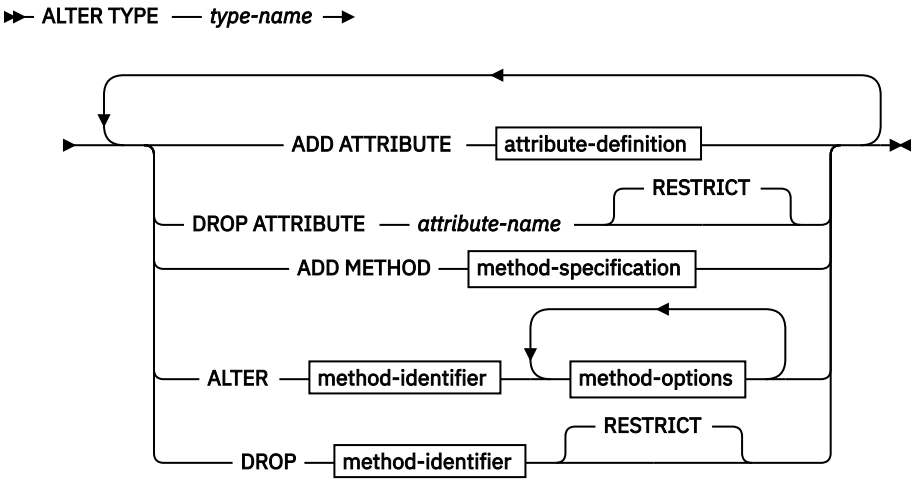
- ALTERIN privilege on the schema of the type
- Owner of the type, as recorded in the OWNER column of the SYSCAT.DATATYPES catalog view
- SCHEMAADM authority on the schema of the type
- DBADM authority

To alter a method to be not fenced, the privileges held by the authorization ID of the statement must also include at least one of the following authorities:

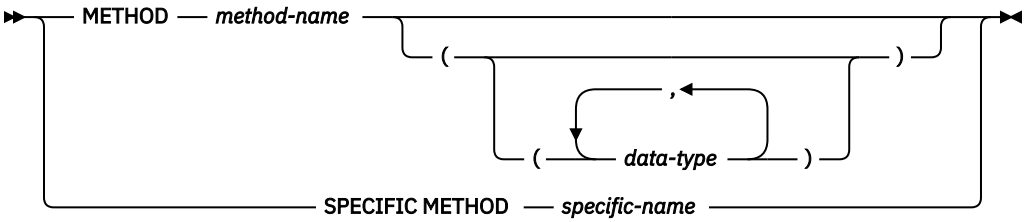
- CREATE_NOT_FENCED_ROUTINE authority on the database
- DBADM authority

To alter a method to be fenced, no additional authorities or privileges are required.

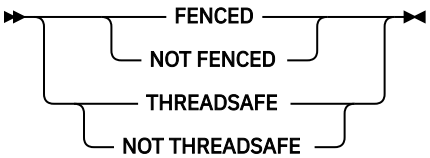
Syntax



method-identifier



method-options



Description

type-name

Identifies the structured type to be changed. It must be an existing type defined in the catalog (SQLSTATE 42704), and the type must be a structured type (SQLSTATE 428DP). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

ADD ATTRIBUTE

Adds an attribute after the last attribute of the existing structured type.

attribute-definition

Defines the attributes of the structured type.

attribute-name

Specifies a name for the attribute. The name cannot be the same as any other attribute of this structured type (including inherited attributes) or any subtype of this structured type (SQLSTATE 42711).

A number of names used as keywords in predicates are reserved for system use, and may not be used as an *attribute-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH and the comparison operators.

data-type 1
Specifies the data type of the attribute. It is one of the data types listed under CREATE TABLE, other than XML (SQLSTATE 42601). The data type must identify an existing data type (SQLSTATE 42704). If *data-type* is specified without a schema name, the type is resolved by searching the schemas on the SQL path. The description of various data types is given in "CREATE TABLE". If the attribute data type is a reference type, the target type of the reference must be a structured type that exists (SQLSTATE 42704).

To prevent type definitions that, at run time, would permit an instance of the type to directly, or indirectly, contain another instance of the same type or one of its subtypes, there is a restriction that a type may not be defined such that one of its attribute types directly or indirectly uses itself (SQLSTATE 428EP).

Character and graphic string data types cannot specify string units of CODEUNITS32.

lob-options
Specifies the options associated with LOB types (or distinct types based on LOB types). For a detailed description of lob-options, see "CREATE TABLE".

DROP ATTRIBUTE
Drops an attribute of the existing structured type.

attribute-name
The name of the attribute. The attribute must exist as an attribute of the type (SQLSTATE 42703).

RESTRICT
Enforces the rule that no attribute can be dropped if *type-name* is used as the type of an existing table, view, column, attribute nested inside the type of a column, or an index extension.

ADD METHOD *method-specification*
Adds a method specification to the type identified by *type-name*. The method cannot be used until a separate CREATE METHOD statement is used to give the method a body. For more information about *method-specification*, see "CREATE TYPE (Structured)".

ALTER *method-identifier*
Uniquely identifies an instance of a method that is to be altered. The specified method may or may not have an existing method body. Methods declared as LANGUAGE SQL cannot be altered (SQLSTATE 42917).

method-identifier

METHOD *method-name*
Identifies a particular method, and is valid only if there is exactly one method instance with the name *method-name* for the type *type-name*. The identified method can have any number of parameters defined for it. If no method by this name exists for the type, an error (SQLSTATE 42704) is raised. If there is more than one instance of the method for the type, an error (SQLSTATE 42725) is raised.

METHOD *method-name (data-type,...)*
Provides the method signature, which uniquely identifies the method. The method resolution algorithm is not used.

method-name
Specifies the name of the method for the type *type-name*.

(data-type,...)
Values must match the data types that were specified (in the corresponding position) on the CREATE TYPE statement. The number of data types, and the logical concatenation of the data types, is used to identify the specific method instance.

If a data type is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision, or scale for the parameterized data types. Instead, an empty set of parentheses can be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (REAL or DOUBLE).

If length, precision, or scale is coded, the value must exactly match that specified in the CREATE TYPE statement.

A type of FLOAT(*n*) does not need to match the defined value for *n*, because 0 < *n* < 25 means REAL, and 24 < *n* < 54 means DOUBLE. Matching occurs on the basis of whether the type is REAL or DOUBLE.

If no method with the specified signature exists for the type in the named or implied schema, an error (SQLSTATE 42883) is raised.

SPECIFIC METHOD *specific-name*
Identifies a particular method, using the name that is specified or defaulted to at method creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific method instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

method-options
Specifies the options that are to be altered for the method.

FENCED or NOT FENCED
Specifies whether the method is considered safe to run in the database manager operating environment's process or address space (NOT FENCED), or not (FENCED). Most methods have the option of running as FENCED or NOT FENCED.

If a method is altered to be FENCED, the database manager insulates its internal resources (for example, data buffers) from access by the method. In general, a method running as FENCED will not perform as well as a similar one running as NOT FENCED.

CAUTION: Use of NOT FENCED for methods that were not adequately coded, reviewed, and tested can compromise the integrity of a Db2 database. Db2 databases take some precautions against many of the common types of inadvertent failures that might occur, but cannot guarantee complete integrity when NOT FENCED methods are used.

A method declared as NOT THREADSAFE cannot be altered to be NOT FENCED (SQLSTATE 42613).

If a method has any parameters defined AS LOCATOR, and was defined with the NO SQL option, the method cannot be altered to be FENCED (SQLSTATE 42613).

This option cannot be altered for LANGUAGE OLE methods (SQLSTATE 42849).

THREADSAFE or NOT THREADSAFE
Specifies whether a method is considered safe to run in the same process as other routines (THREADSAFE), or not (NOT THREADSAFE).

If the method is defined with LANGUAGE other than OLE:

- If the method is defined as THREADSAFE, the database manager can invoke the method in the same process as other routines. In general, to be threadsafe, a method should not use any global or static data areas. Most programming references include a discussion of writing threadsafe routines. Both FENCED and NOT FENCED methods can be THREADSAFE. If the method is defined with LANGUAGE OLE, THREADSAFE may not be specified (SQLSTATE 42613).
- If the method is defined as NOT THREADSAFE, the database manager will never invoke the method in the same process as another routine. Only a fenced method can be NOT THREADSAFE (SQLSTATE 42613).

DROP *method-identifier*
Uniquely identifies an instance of a method that is to be dropped. The specified method must not have an existing method body (SQLSTATE 428ER). Use the DROP METHOD statement to drop the method

body before using ALTER TYPE DROP METHOD. Methods implicitly generated by the CREATE TYPE statement (such as mutators and observers) cannot be dropped (SQLSTATE 42917).

RESTRICT
Indicates that the specified method is restricted from having an existing method body. Use the DROP METHOD statement to drop the method body before using ALTER TYPE DROP METHOD.

Rules

- Adding or dropping an attribute is not allowed for type *type-name* (SQLSTATE 55043) if either:
  - The type or one of its subtypes is the type of an existing table or view.
  - There exists a column of a table whose type directly or indirectly uses *type-name*. The terms *directly uses* and *indirectly uses* are defined in "Structured types".
  - The type or one of its subtypes is used in an index extension.
- A type may not be altered by adding attributes so that the total number of attributes for the type, or any of its subtypes, exceeds 4082 (SQLSTATE 54050).
- ADD ATTRIBUTE option:
  - ADD ATTRIBUTE generates observer and mutator methods for the new attribute. These methods are similar to those generated when a structured type is created (see "CREATE TYPE (Structured)"). If these methods conflict with or override any existing methods or functions, the ALTER TYPE statement fails (SQLSTATE 42745).
  - If the INLINE LENGTH for the type (or any of its subtypes) was explicitly specified by the user with a value less than 292, and the attributes added cause the specified inline length to be less than the size of the result of the constructor function for the altered type (32 bytes plus 10 bytes per attribute), then an error results (SQLSTATE 42611).
- DROP ATTRIBUTE option:
  - An attribute that is inherited from an existing supertype cannot be dropped (SQLSTATE 428DJ).
  - DROP ATTRIBUTE drops the mutator and observer methods of the dropped attributes, and checks dependencies on those dropped methods.
- DROP METHOD option:
  - An original method that is overridden by other methods cannot be dropped (SQLSTATE 42893).

Notes

- It is not possible to alter a method that is in the SYSIBM, SYSFUN, or SYSPROC schema (SQLSTATE 42832).
- When a type is altered by adding or dropping an attribute, all packages are invalidated that depend on functions or methods that use this type or a subtype of this type as a parameter or a result.
- When an attribute is added to or dropped from a structured type:
  - If the INLINE LENGTH of the type was calculated by the system when the type was created, the INLINE LENGTH values are automatically modified for the altered type, and all of its subtypes to account for the change. The INLINE LENGTH values are also automatically (recursively) modified for all structured types where the INLINE LENGTH was calculated by the system and the type includes an attribute of any type with a changed INLINE LENGTH.
  - If the INLINE LENGTH of any type affected by adding or dropping attributes was explicitly specified by a user, then the INLINE LENGTH for that particular type is not changed. Special care must be taken for explicitly specified inline lengths. If it is likely that a type will have attributes added later on, then the inline length, for any uses of that type or one of its subtypes in a column definition, should be large enough to account for the possible increase in length of the instantiated object.
  - If new attributes are to be made visible to application programs, existing transform functions must be modified to match the new structure of the data type.

- In a partitioned database environment, the use of SQL in external user-defined functions or methods is not supported (SQLSTATE 42997).
- ***Privileges***: The EXECUTE privilege is not given for any methods explicitly specified in the ALTER TYPE statement until a method body is defined using the CREATE METHOD statement. The owner of the user-defined type has the ability to drop the method specification using the ALTER TYPE statement.

Examples

- *Example 1:* The ALTER TYPE statement can be used to permit a cycle of mutually referencing types and tables. Consider mutually referencing tables named EMPLOYEE and DEPARTMENT.

  The following sequence would allow the types and tables to be created.

```
CREATE TYPE DEPT ...
CREATE TYPE EMP ... (including attribute named DEPTREF of type REF(DEPT))
ALTER TYPE DEPT ADD ATTRIBUTE MANAGER REF(EMP)
CREATE TABLE DEPARTMENT OF DEPT ...
CREATE TABLE EMPLOYEE OF EMP (DEPTREF WITH OPTIONS SCOPE DEPARTMENT)
ALTER TABLE DEPARTMENT ALTER COLUMN MANAGER ADD SCOPE EMPLOYEE
```

  The following sequence would allow these tables and types to be dropped.

```
DROP TABLE EMPLOYEE   (the MANAGER column in DEPARTMENT becomes unscoped)
DROP TABLE DEPARTMENT
ALTER TYPE DEPT DROP ATTRIBUTE MANAGER
DROP TYPE EMP
DROP TYPE DEPT
```

- *Example 2:* The ALTER TYPE statement can be used to create a type with an attribute that references a subtype.

```
CREATE TYPE EMP ...
CREATE TYPE MGR UNDER EMP ...
ALTER TYPE EMP ADD ATTRIBUTE MANAGER REF(MGR)
```

- *Example 3:* The ALTER TYPE statement can be used to add an attribute. The following statement adds the SPECIAL attribute to the EMP type. Because the inline length was not specified on the original CREATE TYPE statement, the inline length is recalculated by adding 13 (10 bytes for the new attribute + attribute length + 2 bytes for a non-LOB attribute).

```
ALTER TYPE EMP ...
  ADD ATTRIBUTE SPECIAL CHAR(1)
```

- *Example 4:* The ALTER TYPE statement can be used to add a method associated with a type. The following statement adds a method called BONUS.

```
ALTER TYPE EMP ...
  ADD METHOD BONUS (RATE DOUBLE)
    RETURNS INTEGER
    LANGUAGE SQL
    CONTAINS SQL
    NO EXTERNAL ACTION
    DETERMINISTIC
```

  Note that the BONUS method cannot be used until a CREATE METHOD statement is issued to create the method body. If it is assumed that type EMP includes an attribute called SALARY, then the following example shows a method body definition.

```
CREATE METHOD BONUS(RATE DOUBLE) FOR EMP
  RETURN CAST(SELF.SALARY * RATE AS INTEGER)
```

ALTER USAGE LIST

The ALTER USAGE LIST statement alters the definition of a usage list.

Invocation

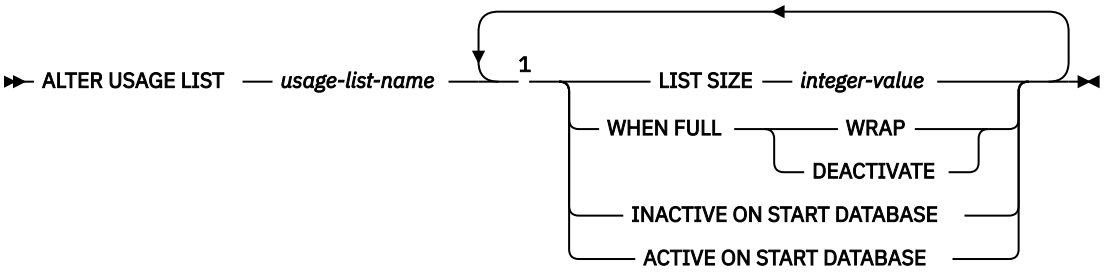
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include one of the following privileges:

- DBADM authority
- SQLADM authority

Syntax



Notes:

[1] The same clause cannot be specified more than once

Description

usage-list-name
Identifies the usage list to be altered. The *usage-list-name* must identify a usage list that exists at the current server (SQLSTATE 42704).

LIST SIZE *integer-value*
Specifies that the size of this list is *integer-value* entries. The minimum size that can be specified is 10 and the maximum is 5000 (SQLSTATE 428B7).

WHEN FULL
Specifies the action to perform when an active usage list becomes full.

WRAP
Specifies that the usage list wraps and replaces the oldest entries.

DEACTIVATE
Specifies that the usage list deactivates.

INACTIVE ON START DATABASE
Specifies that the usage list is not activated for monitoring whenever the database is activated. Collection must be explicitly started using the SET USAGE LIST statement.

ACTIVE ON START DATABASE
Specifies that the usage list is automatically activated for monitoring whenever the database is activated. In a partitioned database environment or Db2 pureScale environment, the collection is automatically started whenever the database member is activated.

Notes

- ***When changes take effect***: If the current state of a usage list is active, then the alterations do not take effect when the statement is processed or when the changes are committed. The changes to the usage list take effect the next time the state of usage list is set to active. In a partitioned database environment or Db2 pureScale environment, the alterations take effect the next time the usage list at a member is activated.

ALTER USER MAPPING

The ALTER USER MAPPING statement is used to change the authorization ID or password that is used at a data source for a specified federated server authorization ID.

Invocation

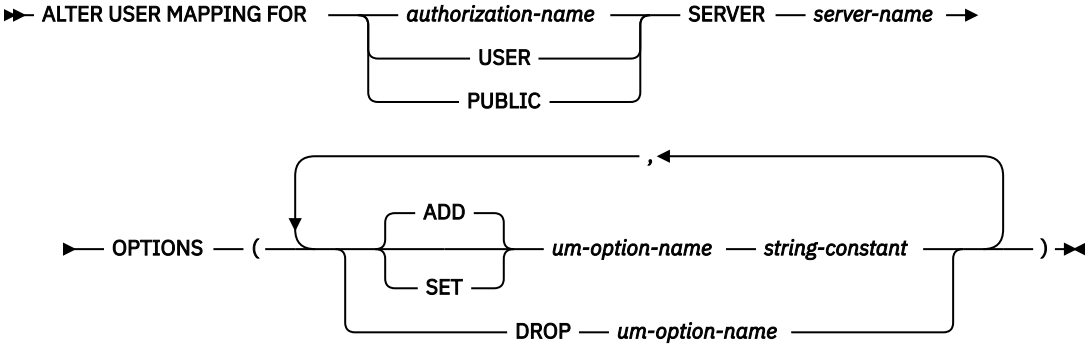
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

If the authorization ID of the statement is different from the authorization name that is mapped to the data source, the privileges held by the authorization ID of the statement must include DBADM authority. Otherwise, if the authorization ID and the authorization name match, no authorities or privileges are required.

When altering a public user mapping, the privileges held by the authorization ID of the statement must include DBADM authority.

Syntax



Description

authorization-name
Specifies the authorization name under which a user or application connects to a federated database.

USER
The value in the special register USER. When USER is specified, then the authorization ID of the ALTER USER MAPPING statement will be mapped to the data source authorization ID that is specified in the REMOTE_AUTHID user option.

PUBLIC
Specifies that any valid authorization ID for the local federated database will be mapped to the data source authorization ID that is specified in the REMOTE_AUTHID user option.

SERVER *server-name*
Identifies the data source accessible under the remote authorization ID that maps to the local authorization ID that's denoted by *authorization-name* or referenced by USER.

OPTIONS
Indicates what user options are to be enabled, reset, or dropped for the mapping that is being altered.

ADD
Enables a new user option.

SET
Changes the setting of a user option.

um-option-name
The user mapping option that is to be added or reset. Which options you can specify depends on the data source of the object for which a wrapper is being created. For a list of data sources and the user mapping options that apply to each, see Data source options.

string-constant
The user-mapping option setting as a character string constant enclosed in single quotation marks.

DROP *um-option-name*
Drops a user mapping option.

Notes

- A user option cannot be specified more than once in the same ALTER USER MAPPING statement (SQLSTATE 42853). When a user option is enabled, reset, or dropped, any other user options that are in use are not affected.
- An ALTER USER MAPPING statement within a given unit of work (UOW) cannot be processed (SQLSTATE 55007) if the UOW already includes one of the following items:
  - A SELECT statement that references a nickname for a table or view at the data source that is to be included in the mapping
  - An open cursor on a nickname for a table or view at the data source that is to be included in the mapping
  - Either an INSERT, DELETE, or UPDATE issued against a nickname for a table or view at the data source that is to be included in the mapping.
- Public user mappings and non-public user mappings cannot coexist on the same federated server. This means that if you have created public user mappings, you will not be able to create non-public user mappings on the same federated server. The reverse is also true, if you have created non-public user mappings, you will not be able to create public user mappings on the same federated server.

Examples

1. Jim uses a local database to connect to an Oracle data source called ORACLE1. He accesses the local database under the authorization ID KLEEWEIN; KLEEWEIN maps to CORONA, the authorization ID under which he accesses ORACLE1. Jim is going to start accessing ORACLE1 under a new ID, JIMK. So KLEEWEIN now needs to map to JIMK.

```
ALTER USER MAPPING FOR KLEEWEIN
  SERVER ORACLE1
  OPTIONS ( SET REMOTE_AUTHID 'JIMK'  )
```

2. Mary uses a federated database to connect to a Db2 for z/OS data source called DORADO. She uses one authorization ID to access Db2 and another to access DORADO, and she has created a mapping between these two IDs. She has been using the same password with both IDs, but now decides to use a separate password, ZNYQ, with the ID for DORADO. Accordingly, she needs to map her federated database password to ZNYQ.

```
ALTER USER MAPPING FOR MARY
  SERVER DORADO
  OPTIONS ( ADD REMOTE_PASSWORD 'ZNYQ'  )
```

ALTER VIEW

The ALTER VIEW statement modifies an existing view by altering a reference type column to add a scope. The ALTER VIEW statement also enables or disables a view for use in query optimization.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

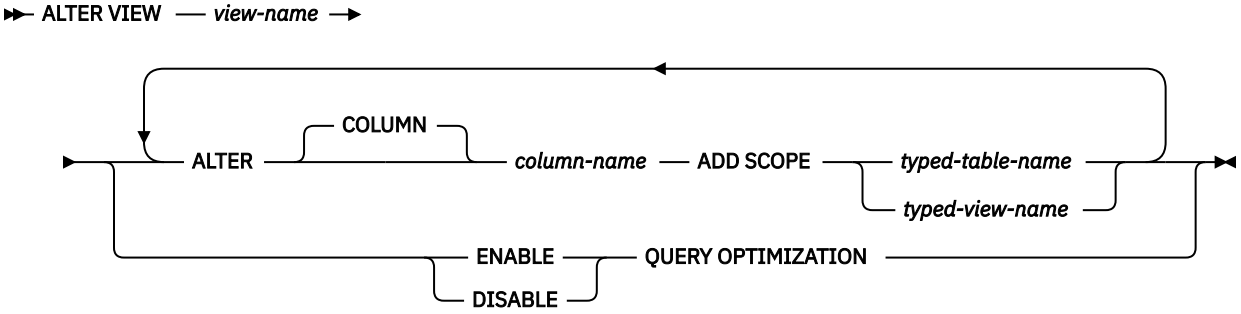
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- ALTERIN privilege on the schema of the view
- Owner of the view to be altered
- CONTROL privilege on the view to be altered
- SCHEMAADM authority on the schema of the view
- DBADM authority

To enable or disable a view for use in query optimization, the privileges held by the authorization ID of the statement must also include at least one of the following authorities for each of the tables or underlying tables of views that are referenced in the FROM clause of the view fullselect:

- ALTER privilege on the table
- ALTERIN privilege on the schema of the table
- SCHEMAADM authority on the schema of the table
- DBADM authority

Syntax



Description

view-name
Specifies the view that is to be changed. It must be a view that is described in the catalog.

ALTER COLUMN *column-name*
Specifies the name of the column that is to be altered. The *column-name* must identify an existing column of the view (SQLSTATE 42703). The name cannot be qualified.

ADD SCOPE
Adds a scope to an existing reference type column that does not already have a scope defined (SQLSTATE 428DK). The column must not be inherited from a superview (SQLSTATE 428DJ).

typed-table-name
Specifies the name of a typed table. The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-table-name* (SQLSTATE 428DM). No checking is done of any existing values in *column-name* to ensure that the values actually reference existing rows in *typed-table-name*.

typed-view-name
Specifies the name of a typed view. The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-view-name* (SQLSTATE 428DM). No checking is done of any existing values in *column-name* to ensure that the values actually reference existing rows in *typed-view-name*.

ENABLE QUERY OPTIMIZATION or DISABLE QUERY OPTIMIZATION
Specifies whether or not the view and any associated statistics are to be used to improve the optimization of queries. DISABLE QUERY OPTIMIZATION is the default when a view is created.

ENABLE QUERY OPTIMIZATION
Specifies that the view includes statistics that can be used to improve the optimization of queries that involve this view or queries that include subqueries similar to the fullselect of this view.

DISABLE QUERY OPTIMIZATION
Specifies that the view and any associated statistics are not to be used to improve the optimization of queries.

Rules

- A view cannot be enabled for query optimization if:
  - The view directly or indirectly references a materialized query table (MQT). Note that an MQT or statistical view can reference a statistical view
  - The view directly or indirectly references a catalog table.
  - It is a typed view

Notes

- To be considered for optimizing a query, a view:
  - Cannot contain an aggregation or distinct operation
  - Cannot contain a union, except, or intersect operation
  - Cannot contain an OLAP specification
- If a view is altered to disable query optimization, cached query plans that used the view for query optimization are invalidated. If a view is altered to enable query optimization, cached query plans are invalidated if they reference the same tables as the newly enabled view references, either directly or indirectly through other views. The invalidation of these cached query plans results in implicit revalidation that takes the view's changed query optimization property into account.

  The query optimization property for a view has no impact on static embedded SQL statements.

ALTER WORK ACTION SET

The ALTER WORK ACTION SET statement alters a work action set by adding, altering, or dropping work actions within the work action set.

Invocation

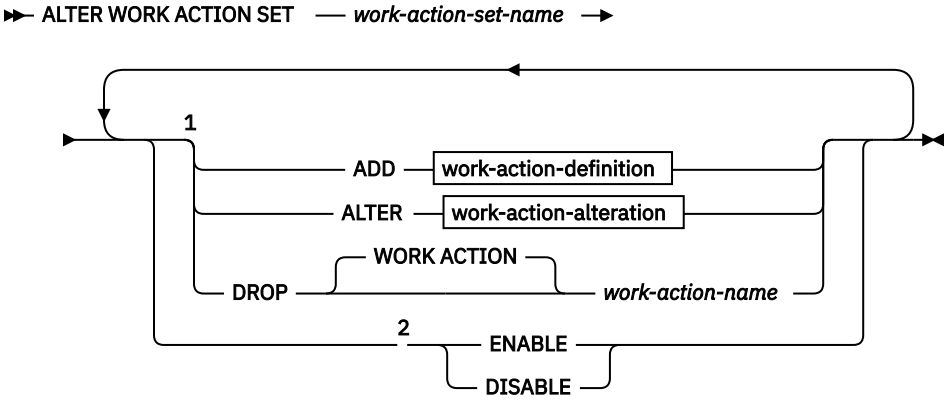
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

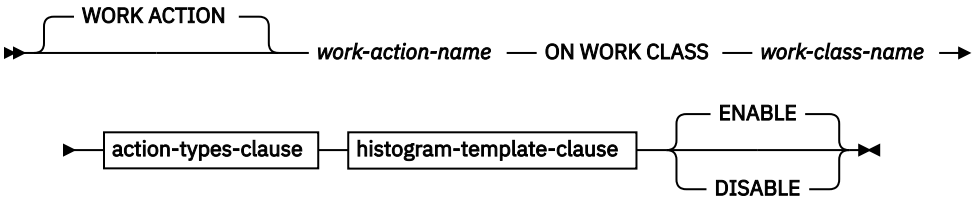
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- SQLADM authority, only if every alteration clause is a COLLECT clause
- WLMADM authority
- DBADM authority

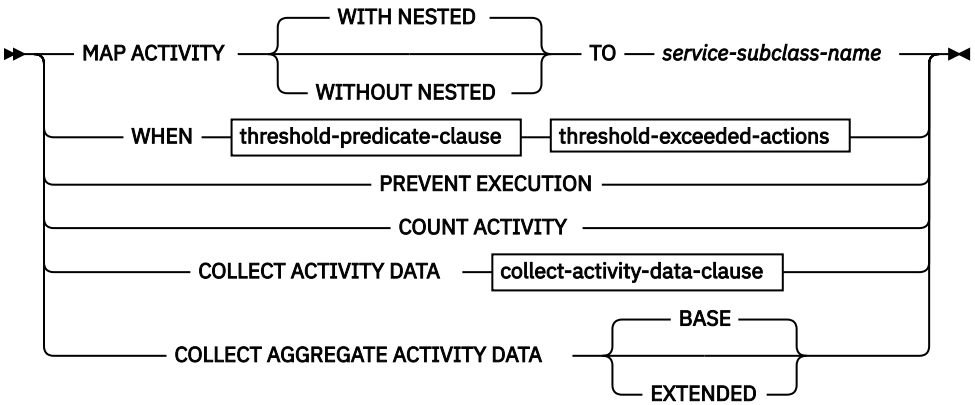
Syntax



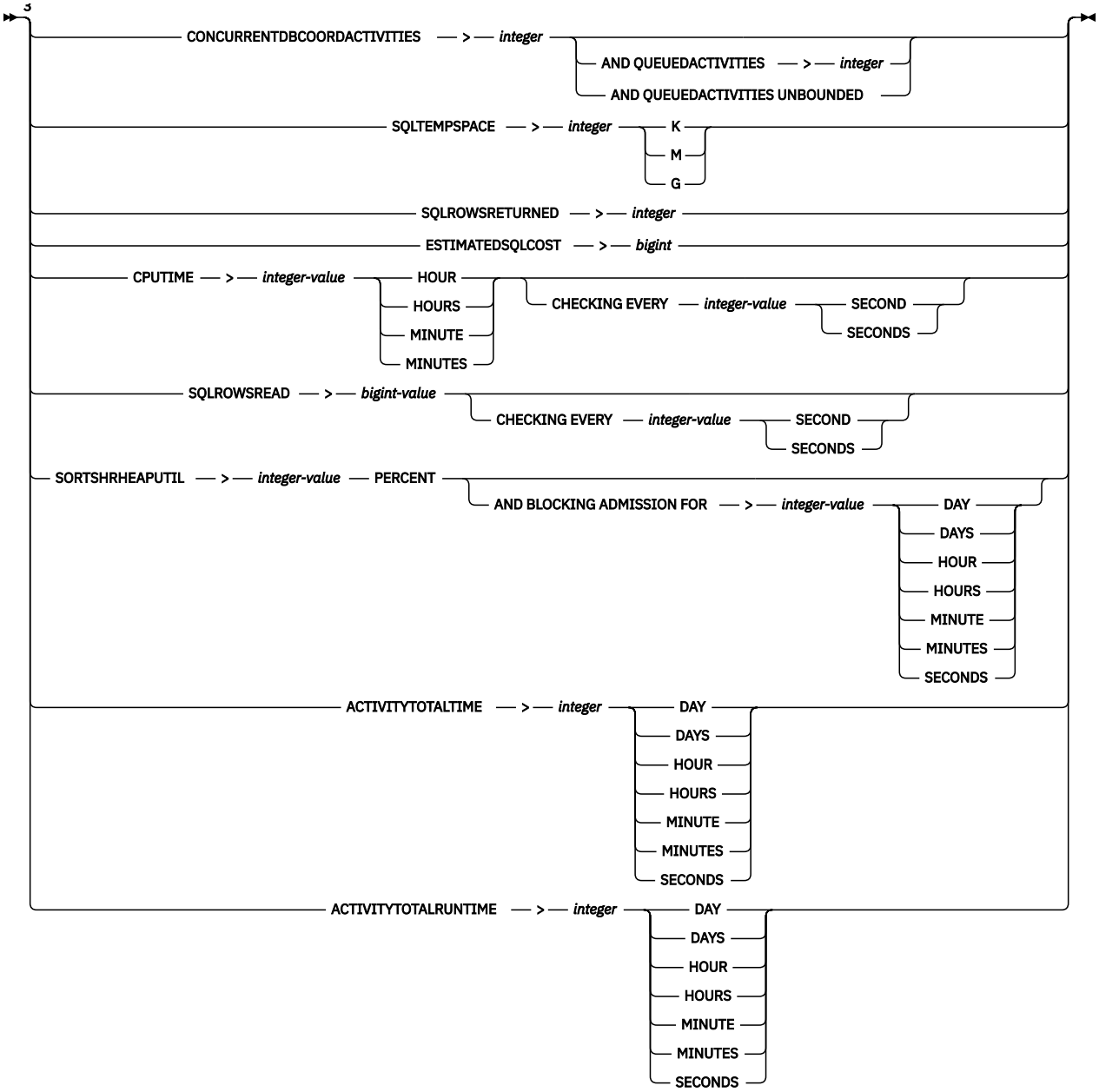
work-action-definition



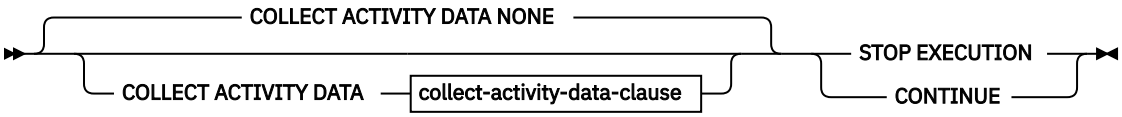
action-types-clause



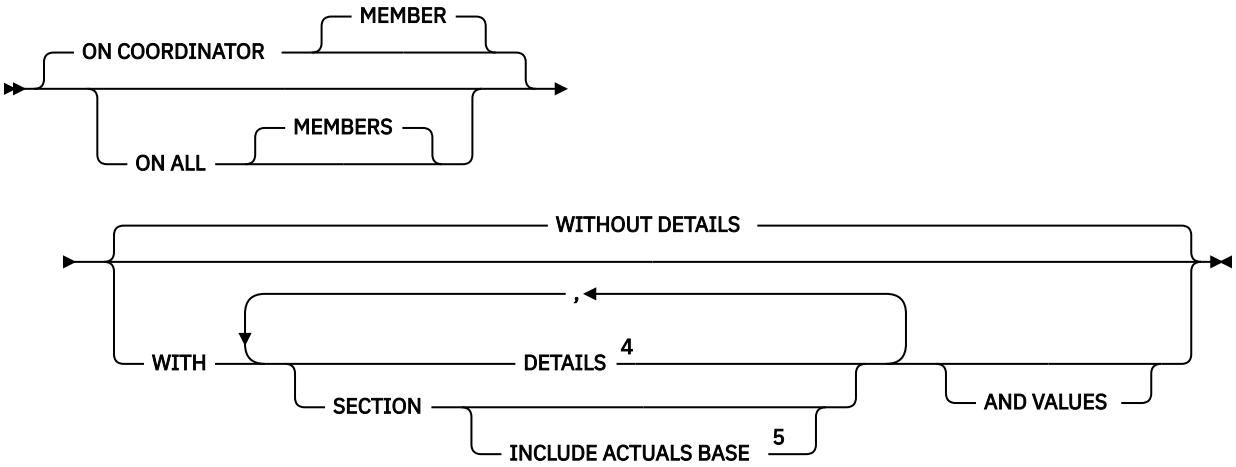
threshold-predicate-clause



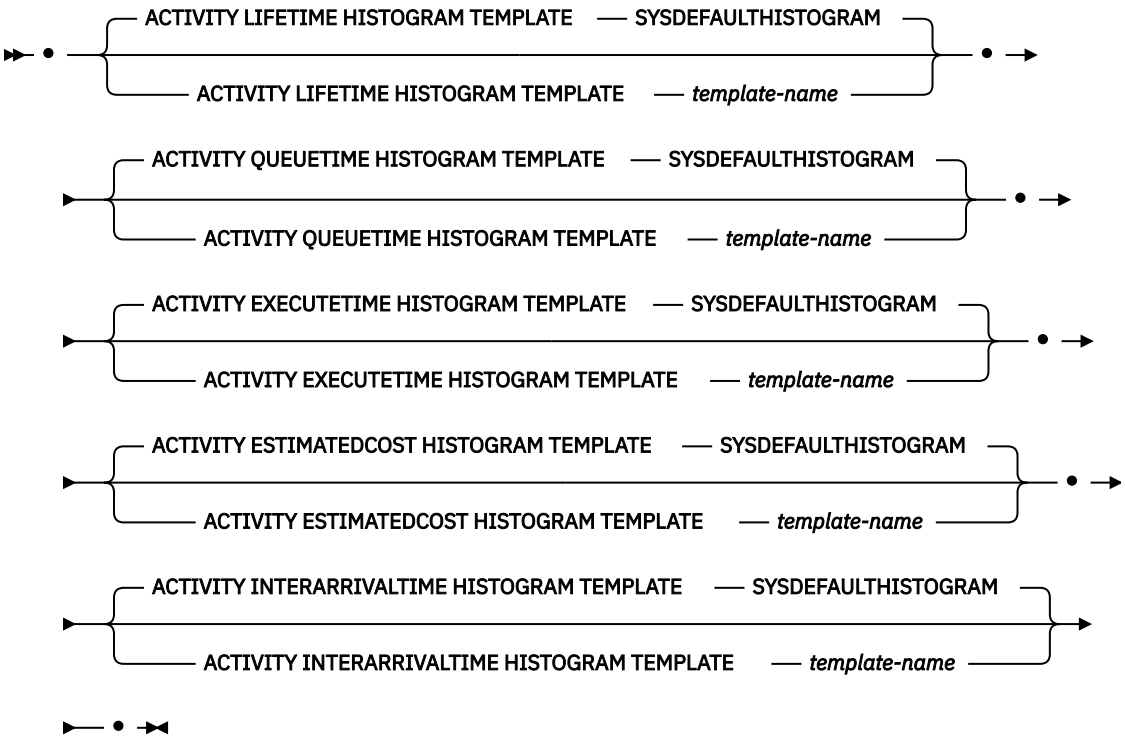
threshold-exceeded-actions



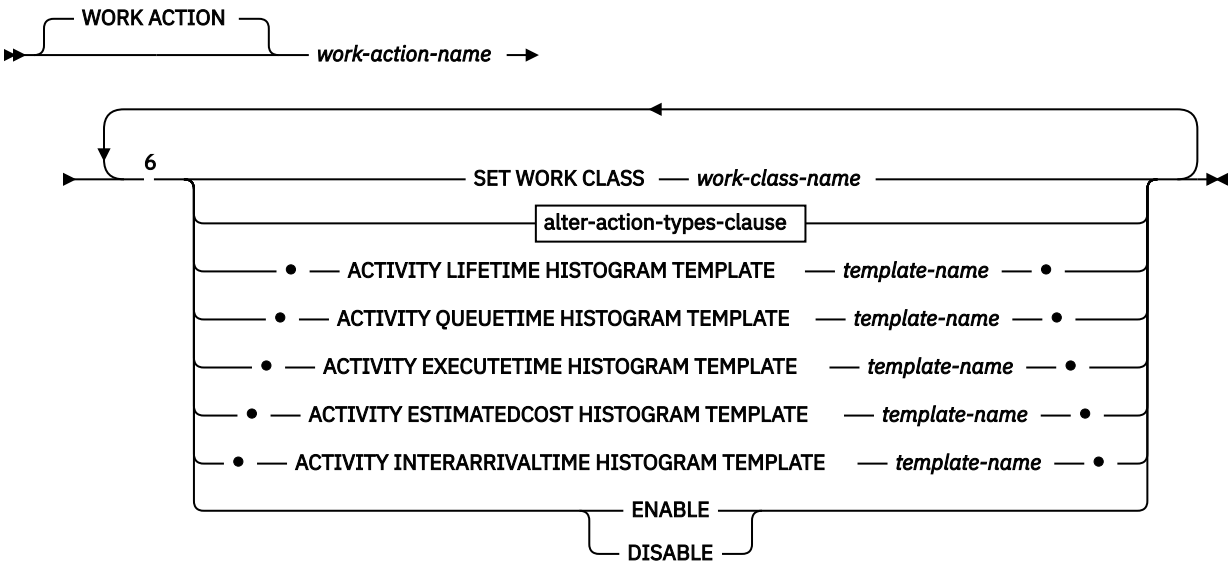
collect-activity-data-clause



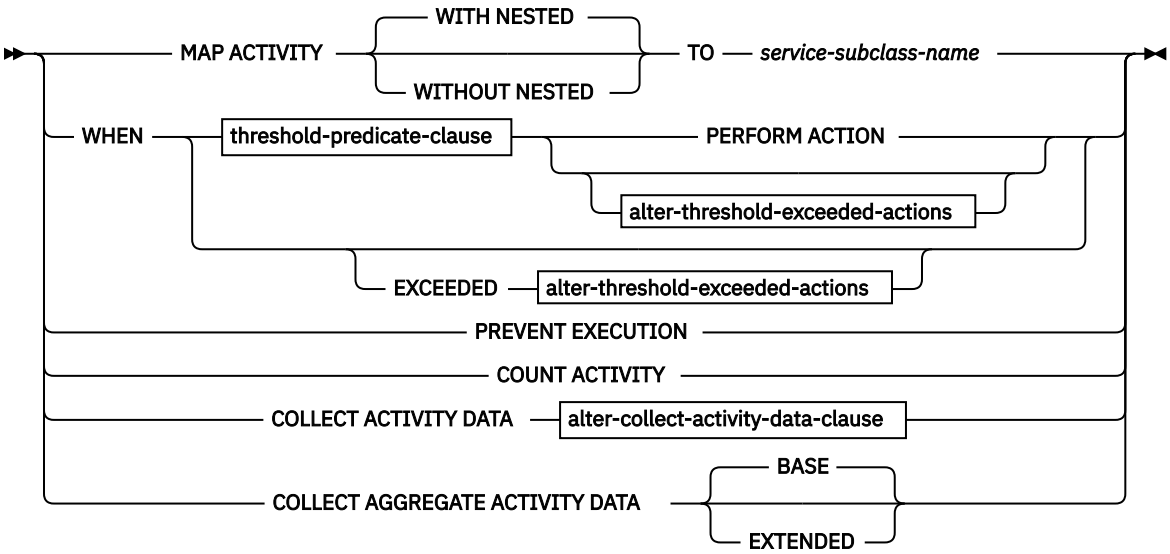
histogram-template-clause



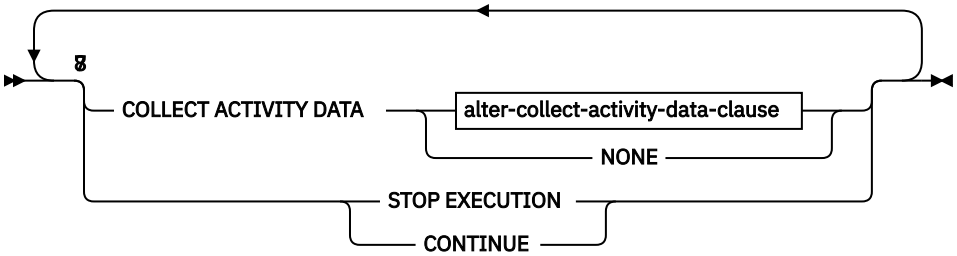
work-action-alteration



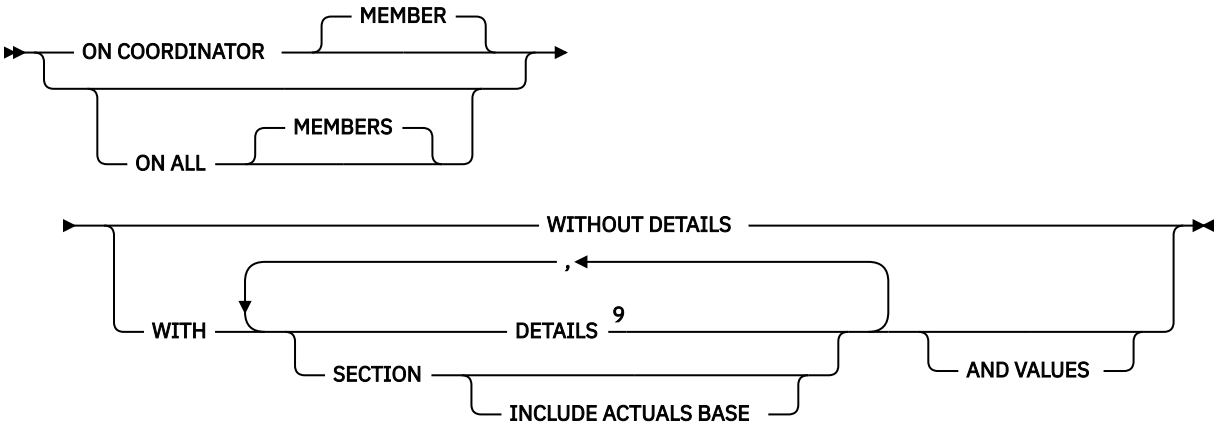
alter-action-types-clause



alter-threshold-exceeded-actions



alter-collect-activity-data-clause



Notes:

[1] The ADD, ALTER, and DROP clauses are processed in the order in which they are specified.

[2] The ENABLE or DISABLE clause can only be specified once in the same statement.

[3] Only one work action of the same threshold type can be applied to a single work class at a time. When altering a threshold work action, the threshold predicate cannot be changed.

[4] The DETAILS keyword is the minimum to be specified, followed by the option separated by a comma.

[5] This clause does not apply to thresholds.

[6] The same clause must not be specified more than once.

[7] The same clause must not be specified more than once.

[8] If an existing work action does not have a threshold-exceeded action defined for it and it is being altered to become a threshold work action, then either STOP EXECUTION or CONTINUE must be specified, and if COLLECT ACTIVITY DATA is not specified, then COLLECT ACTIVITY DATA NONE is the default.

[9] The DETAILS keyword is the minimum to be specified, followed by the option separated by a comma.

Description

work-action-set-name

Identifies the work action set that is to be altered. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *work-action-set-name* must identify a work action set that exists at the current server (SQLSTATE 42704).

ADD

Adds a work action to the work action set.

WORK ACTION *work-action-name*

Names the work action. The *work-action-name* must not identify a work action that already exists at the current server under this work action set (SQLSTATE 42710). The *work-action-name* cannot begin with 'SYS' (SQLSTATE 42939).

ON WORK CLASS *work-class-name*

Specifies the work class that identifies the database activities to which this work action will apply. The *work-class-name* must exist in the *work-class-set-name* at the current server (SQLSTATE 42704).

MAP ACTIVITY

Specifies a work action of mapping the activity. This action can only be specified if the object for which this work action set is defined is a service superclass (SQLSTATE 5U034).

WITH NESTED or WITHOUT NESTED

Specifies whether or not activities that are nested under this activity are mapped to the service subclass. The default is WITH NESTED.

WITH NESTED

All database activities that have a nesting level of zero that are classified under the work class, and all database activities nested under this activity, are mapped to the service

subclass; that is, activities with a nesting level greater than zero are run under the same service class as activities with a nesting level of zero.

WITHOUT NESTED
Only database activities that have a nesting level of zero that are classified under the work class are mapped to the service subclass. Database activities that are nested under this activity are handled according to their activity type.

TO *service-subclass-name*
Specifies the service subclass to which activities are to be mapped. The *service-subclass-name* must already exist in the *service-superclass-name* at the current server (SQLSTATE 42704). The *service-subclass-name* cannot be the default service subclass, SYSDEFAULTSUBCLASS (SQLSTATE 5U018).

WHEN
Specifies the threshold that will be applied to the database activity that is associated with the work class for which this work action is defined. A threshold can only be specified if the database manager object for which this work action set is defined is a database (SQLSTATE 5U034). None of these thresholds apply to internal database activities initiated by the database manager or to database activities generated by administrative SQL routines.

threshold-predicate-clause
For a description of valid threshold types, see the "CREATE THRESHOLD" statement.

threshold-exceeded-actions
For a description of valid threshold-exceeded actions, see the "CREATE THRESHOLD" statement.

PREVENT EXECUTION
Specifies that none of the database activities associated with the work class for which this work action is defined will be allowed to run (SQLSTATE 5U033).

COUNT ACTIVITY
Specifies that all of the database activities associated with the work class are to be run and that each time one is run, the counter for the work class will be incremented.

COLLECT ACTIVITY DATA
Specifies that data about each activity associated with the work class for which this work action is defined is to be sent to any active activities event monitor when the activity completes.

collect-activity-data-clause

ON COORDINATOR MEMBER
Specifies that the activity data is to be collected at only the coordinator member of the activity.

ON ALL MEMBERS
Specifies that the activity data is to be collected at all members on which the activity is processed. For predictive thresholds, activity information is collected at all members only if you also specify the CONTINUE action for exceeded thresholds. For reactive thresholds, the ON ALL MEMBERS clause has no effect and activity information is always collected only at the coordinator member. For both predictive and reactive thresholds, any input data values, section information, or values will be collected only at the coordinator member.

WITHOUT DETAILS
Specifies that data about each activity associated with the work class for which this work action is defined should be sent to any active activities event monitor, when the activity completes execution. Details about statement, compilation environment, and section environment data are not sent.

WITH

DETAILS
Specifies that statement and compilation environment data is to be sent to any active activities event monitor, for those activities that have them. Section environment data is not sent.

SECTION
Specifies that statement, compilation environment and section environment data is to be sent to any active activities event monitor for those activities that have them. DETAILS must be specified if SECTION is specified.

INCLUDE ACTUALS BASE
Specifies that section actuals should also be collected on any partition where the activity data is collected. For section actuals to be collected, either INCLUDE ACTUALS clause must be specified or the **section_actuals** database configuration parameter must be set.

The effective setting for the collection of section actuals is the combination of the INCLUDE ACTUALS clause, the **section_actuals** database configuration parameter, and the <collectsectionactuals> setting specified on the WLM_SET_CONN_ENV routine. For example, if INCLUDE ACTUALS BASE is specified, yet the **section_actuals** database configuration parameter value is NONE and <collectsectionactuals> is set to NONE, then the effective setting for the collection of section actuals is BASE.

BASE specifies that the following actuals should be enabled and collected during the activity's execution:

- Basic operator cardinality counts
- Statistics for each object referenced (DML statements only)

AND VALUES
Specifies that input data values are to be sent to any active activities event monitor, for those activities that have them.

NONE
Specifies that activity data should not be collected for each activity that is associated with the work class for which this work action is defined.

COLLECT AGGREGATE ACTIVITY DATA
Specifies that aggregate activity data is to be captured for activities that are associated with the work class for which this work action is defined and sent to the statistics event monitor, if one is active. This information is collected periodically on an interval that is specified by the **wlm_collect_int** database configuration parameter. The default is COLLECT AGGREGATE ACTIVITY DATA BASE. This clause cannot be specified for a work action defined in a work action set that is applied to a database.

BASE
Specifies that basic aggregate activity data should be captured for activities associated with the work class for which this work action is defined and sent to the statistics event monitor, if one is active. Basic aggregate activity data includes:

- Estimated activity cost high watermark
- Rows returned high watermark
- Temporary table space usage high watermark. Only activities that have an SQLTEMPSPACE threshold applied to them participate in this high watermark.
- Activity life time histogram
- Activity queue time histogram
- Activity execution time histogram

EXTENDED
Specifies that all aggregate activity data should be captured for activities associated with the work class for which this work action is defined and sent to the statistics event monitor, if one is active. This includes all basic aggregate activity data plus:

- Activity data manipulation language (DML) estimated cost histogram
- Activity DML inter-arrival time histogram

ENABLE or DISABLE
Specifies whether or not the work action is to be considered when database activities are submitted. The default is ENABLE.

ENABLE
Specifies that the work action is enabled and will be considered when database activities are submitted.

DISABLE
Specifies that the work action is disabled and will not be considered when database activities are submitted.

histogram-template-clause
Specifies histogram templates to use when collecting aggregate activity data for activities associated with the work class to which this work action is assigned. Aggregate activity data is only collected for the work class when the work action type is COLLECT AGGREGATE ACTIVITY DATA.

ACTIVITY LIFETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the duration, in milliseconds, of database activities running during a specific interval. The database activities are those associated with the work class to which this work action is assigned. This time includes both time queued and time executing. The default is SYSDEFAULTHISTOGRAM. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

ACTIVITY QUEUETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, that database activities are queued during a specific interval. The database activities are those associated with the work class to which this work action is assigned. The default is SYSDEFAULTHISTOGRAM. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

ACTIVITY EXECUTETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, that database activities are executing during a specific interval. The database activities are those associated with the work class to which this work action is assigned. This time does not include the time spent queued. Activity execution time is collected in this histogram at each member where the activity executes. On the activity's coordinator member, this is the end-to-end execution time (that is, the life time less the time spent queued). On non-coordinator members, this is the time that these members spend working on behalf of the activity. During the execution of a given activity, the database manager might present work to a non-coordinator member more than once, and each time the non-coordinator member will collect the execution time for that occurrence of the activity. Therefore, the counts in the execution time histogram might not represent the actual number of unique activities that executed on a member. The default is SYSDEFAULTHISTOGRAM. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

ACTIVITY ESTIMATEDCOST HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the estimated cost, in timerons, of DML activities associated with the work class to which this work action is assigned. The default is SYSDEFAULTHISTOGRAM. This information is

only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified with the EXTENDED option.

ACTIVITY INTERARRIVALTIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, between the arrival of one DML activity and the arrival of the next DML activity, for any activity associated with the work class to which this work action is assigned. The default is SYSDEFAULTHISTOGRAM. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified with the EXTENDED option.

ALTER
Alters the definition of the work action. You can change the work class to which this work action applies, and the action that is to be applied to the database activity that falls within the work class.

WORK ACTION *work-action-name*
Identifies the work action. The *work-action-name* must identify a work action that exists at the current server under this work action set (SQLSTATE 42704).

SET WORK CLASS *work-class-name*
Specifies the work class that identifies the database activities to which this work action will apply. The *work-class-name* must exist in the *work-class-set-name* at the current server (SQLSTATE 42704).

MAP ACTIVITY
Specifies a work action of mapping the activity. This action can only be specified if the object for which this work action set is defined is a service superclass (SQLSTATE 5U034).

WITH NESTED or WITHOUT NESTED
Specifies whether or not activities that are nested under this activity are mapped to the service subclass. The default is WITH NESTED.

WITH NESTED
All database activities that have a nesting level of zero that are classified under the work class, and all database activities nested under this activity are mapped to the service subclass.

WITHOUT NESTED
Only database activities that have a nesting level of zero that are classified under the work class are mapped to the service subclass. Database activities that are nested under this activity are handled according to their activity type.

TO *service-subclass-name*
Specifies the service subclass to which activities are to be mapped. The *service-subclass-name* must already exist in the *service-superclass-name* at the current server (SQLSTATE 42704). The *service-subclass-name* cannot be the default service subclass, SYSDEFAULTSUBCLASS (SQLSTATE 5U018).

WHEN
Specifies the threshold to be altered for the database activity that is associated with the work class for which this work action is defined.

threshold-predicate-clause
For a description of valid threshold types, see the "CREATE THRESHOLD" statement.

PERFORM ACTION
When altering the value of the threshold predicate condition, specifies that the threshold exceeded action is not changed. The work action must be a threshold (SQLSTATE 42613).

alter-threshold-exceeded-actions
For a description of valid alter-threshold-exceeded-actions, see threshold-exceeded-actions in the "CREATE THRESHOLD" statement.

EXCEEDED
Specifies to keep the same threshold predicate that was specified originally for this altered threshold. The work action must be a threshold (SQLSTATE 42613).

PREVENT EXECUTION
Specifies that none of the database activities associated with the work class for which this work action is defined will be allowed to run (SQLSTATE 5U033).

COUNT ACTIVITY
Specifies that all of the database activities associated with the work class are to be run and that each time one is run, the counter for the work class will be incremented.

COLLECT ACTIVITY DATA
Specifies that data about each activity associated with the work class for which this work action is defined is to be sent to any active activities event monitor when the activity completes.

alter-collect-activity-data-clause

ON COORDINATOR MEMBER
Specifies that the activity data is to be collected only at the coordinator member of the activity.

ON ALL MEMBERS
Specifies that activity data is to be collected at all members where the activity is processed. On remote members, a record for the activity may be captured multiple times as the activity comes and goes on those members. If the AND VALUES clause is specified, activity input values will be collected only for the members of the coordinator.

WITHOUT DETAILS
Specifies that data about each activity that is associated with the work class for which this work action is defined should be sent to any active activities event monitor when the activity completes execution. Details about statement, compilation environment, and section environment data are not sent.

WITH

DETAILS
Specifies that statement and compilation environment data is to be sent to any active activities event monitor, for those activities that have them. Section environment data is not sent.

SECTION
Specifies that statement, compilation environment, section environment data, and section actuals are to be sent to any active activities event monitor for those activities that have them. DETAILS must be specified if SECTION is specified. Section actuals will be collected on any member where the activity data is collected.

INCLUDE ACTUALS BASE
Specifies that section actuals should also be collected on any partition where the activity data is collected. For section actuals to be collected, either INCLUDE ACTUALS clause must be specified or the **section_actuals** database configuration parameter must be set.

The effective setting for the collection of section actuals is the combination of the INCLUDE ACTUALS clause, the **section_actuals** database configuration parameter, and the <collectsectionactuals> setting specified on the WLM_SET_CONN_ENV routine. For example, if INCLUDE ACTUALS BASE is specified, yet the **section_actuals** database configuration parameter value is NONE and <collectsectionactuals> is set to NONE, then the effective setting for the collection of section actuals is BASE.

BASE specifies that the following actuals should be enabled and collected during the activity's execution:

- Basic operator cardinality counts
- Statistics for each object referenced (DML statements only)

AND VALUES
Specifies that input data values are to be sent to any active activities event monitor, for those activities that have them.

NONE
Specifies that activity data should not be collected for each activity that is associated with the work class for which this work action is defined.

COLLECT AGGREGATE ACTIVITY DATA
Specifies that aggregate activity data is to be captured for activities that are associated with the work class for which this work action is defined and sent to the statistics event monitor, if one is active. This information is collected periodically on an interval that is specified by the **wlm_collect_int** database configuration parameter. The default is COLLECT AGGREGATE ACTIVITY DATA BASE. This clause cannot be specified for a work action defined in a work action set that is applied to a database.

BASE
Specifies that basic aggregate activity data should be captured for activities associated with the work class for which this work action is defined and sent to the statistics event monitor, if one is active. Basic aggregate activity data includes:

- Estimated activity cost high watermark
- Rows returned high watermark
- Temporary table space usage high watermark
- Activity life time histogram
- Activity queue time histogram
- Activity execution time histogram

EXTENDED
Specifies that all aggregate activity data should be captured for activities associated with the work class for which this work action is defined and sent to the statistics event monitor, if one is active. This includes all basic aggregate activity data plus:

- Activity DML estimated cost histogram
- Activity DML inter-arrival time histogram

ACTIVITY LIFETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the duration, in milliseconds, of database activities running during a specific interval. This time includes both time queued and time executing. The database activities are those associated with the work class to which this work action is assigned. The default is SYSDEFAULTHISTOGRAM. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

ACTIVITY QUEUETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, that database activities are queued during a specific interval. The database activities are those associated with the work class to which this work action is assigned. The default is SYSDEFAULTHISTOGRAM. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

ACTIVITY EXECUTETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, that database activities are executing during a specific interval. The database activities are those associated with the work class to which this work action is assigned. This time does not include the time spent queued. Activity execution time is collected in this histogram at each member where the activity executes. On the activity's coordinator member, this is the end-to-end execution time (that is, the life time less the time spent queued). On non-coordinator members, this is the time that these members spend working on behalf of the activity. During the execution of a given activity, the database manager might present work to a non-coordinator member more than once, and each time the non-coordinator member will collect

the execution time for that occurrence of the activity. Therefore, the counts in the execution time histogram might not represent the actual number of unique activities that executed on a member. The default is SYSDEFAULTHISTOGRAM. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

ACTIVITY ESTIMATEDCOST HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the estimated cost, in timerons, of data manipulation language (DML) activities associated with the work class to which this work action is assigned. The default is SYSDEFAULTHISTOGRAM. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified with the EXTENDED option.

ACTIVITY INTERARRIVALTIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, between the arrival of one DML activity and the arrival of the next DML activity, for any activity associated with the work class to which this work action is assigned. The default is SYSDEFAULTHISTOGRAM. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified with the EXTENDED option.

ENABLE or DISABLE
Specifies whether or not the work action is to be considered when database activities are submitted.

ENABLE
Specifies that the work action is enabled and will be considered when database activities are submitted.

DISABLE
Specifies that the work action is disabled and will not be considered when database activities are submitted.

DROP *work-action-name*
Drops the work action from the work action set. The *work-action-name* must identify a work action that exists at the current server under this work action set (SQLSTATE 42704).

A threshold created as part of a work action set cannot be manipulated directly. You must first disable the work action in order to disable the threshold. You can then drop the work action once the threshold is not being used. For more information, see "Dropping a work action" in the *Db2 Workload Management Guide and Reference*.

ENABLE or DISABLE
Specifies whether or not the work action set is to be considered when database activities are submitted.

ENABLE
Specifies that the work action set is enabled and will be considered when database activities are submitted.

DISABLE
Specifies that the work action set is disabled and will not be considered when database activities are submitted.

Rules

- A workload management (WLM)-exclusive SQL statement must be followed by a COMMIT or a ROLLBACK statement (SQLSTATE 5U021). WLM-exclusive SQL statements are:
  - CREATE HISTOGRAM TEMPLATE, ALTER HISTOGRAM TEMPLATE, or DROP (histogram template)
  - CREATE SERVICE CLASS, ALTER SERVICE CLASS, or DROP (service class)
  - CREATE THRESHOLD, ALTER THRESHOLD, or DROP (threshold)
  - CREATE WORK ACTION SET, ALTER WORK ACTION SET, or DROP (work action set)
  - CREATE WORK CLASS SET, ALTER WORK CLASS SET, or DROP (work class set)

  - CREATE WORKLOAD, ALTER WORKLOAD, or DROP (workload)
  - GRANT (workload privileges) or REVOKE (workload privileges)
- A WLM-exclusive SQL statement cannot be issued within a global transaction (SQLSTATE 51041) such as, for example, an XA transaction.

Notes

- Changes are written to the system catalog, but do not take effect until they are committed, even for the connection that issues the statement.
- Thresholds with a queue, for example CONCURRENTDBCOORDACTIVITIES, must be disabled before they can be dropped.
- Only one uncommitted WLM-exclusive SQL statement at a time is allowed across all partitions. If an uncommitted WLM-exclusive SQL statement is executing, subsequent WLM-exclusive SQL statements will wait until the current WLM-exclusive SQL statement commits or rolls back.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - DATABASE PARTITION can be specified in place of MEMBER, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
  - DATABASE PARTITIONS can be specified in place of MEMBERS, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.

Examples

- *Example 1:* Alter the DATABASE_ACTIONS work action set and add two work actions using the work class LARGE_SELECTS. For the work action ONE_CONCURRENT_SELECT, apply a concurrency threshold of 1 to control the number of activities that can run at one time, and allow a maximum of 3 to be queued. For work action BIG_ROWS_RETURNED, limit the number of rows that can be returned by database activities that fall within that class to 1 000 000.

```
ALTER WORK ACTION SET DATABASE_ACTIONS
  ADD WORK ACTION ONE_CONCURRENT_SELECT ON WORK CLASS LARGE_SELECTS
     WHEN CONCURRENTDBCOORDACTIVITIES > 1
        AND QUEUEDACTIVITIES > 3 STOP EXECUTION
  ADD WORK ACTION BIG_ROWS_RETURNED ON WORK CLASS LARGE_SELECTS
     WHEN SQLROWSRETURNED > 1000000 STOP EXECUTION
```

- *Example 2:* Alter the ADMIN_APPS_ACTIONS work action set to alter the MAP_SELECTS work action to map all activities that run in super service class ADMIN_APPS under the work class SELECT_CLASS to the service subclass ALL_SELECTS. Also add a new work action called MAP_UPDATES that maps all activities that would run in the work class UPDATE_CLASS to the service subclass ALL_SELECTS.

```
ALTER WORK ACTION SET ADMIN_APPS_ACTIONS
  ALTER WORK ACTION MAP_SELECTS MAP ACTIVITY TO ALL_SELECTS
  ADD WORK ACTION MAP_UPDATES ON WORK CLASS UPDATE_CLASS
    MAP ACTIVITY TO ALL_SELECTS
```

ALTER WORK CLASS SET

The ALTER WORK CLASS SET statement adds, alters, or drops work classes within a work class set.

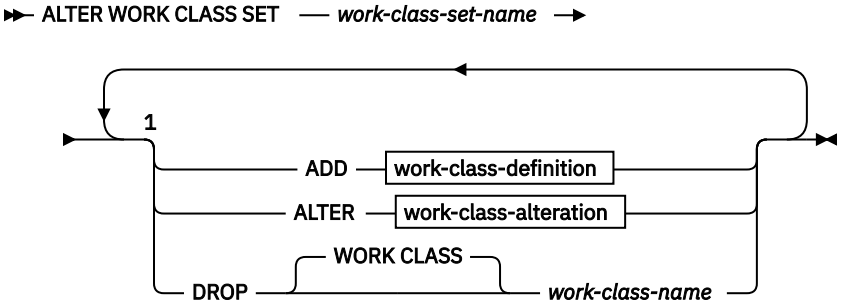
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

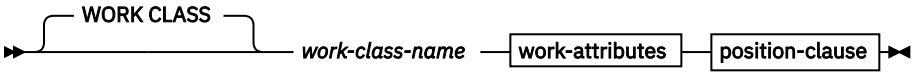
Authorization

The privileges held by the authorization ID of the statement must include WLMADM or DBADM authority.

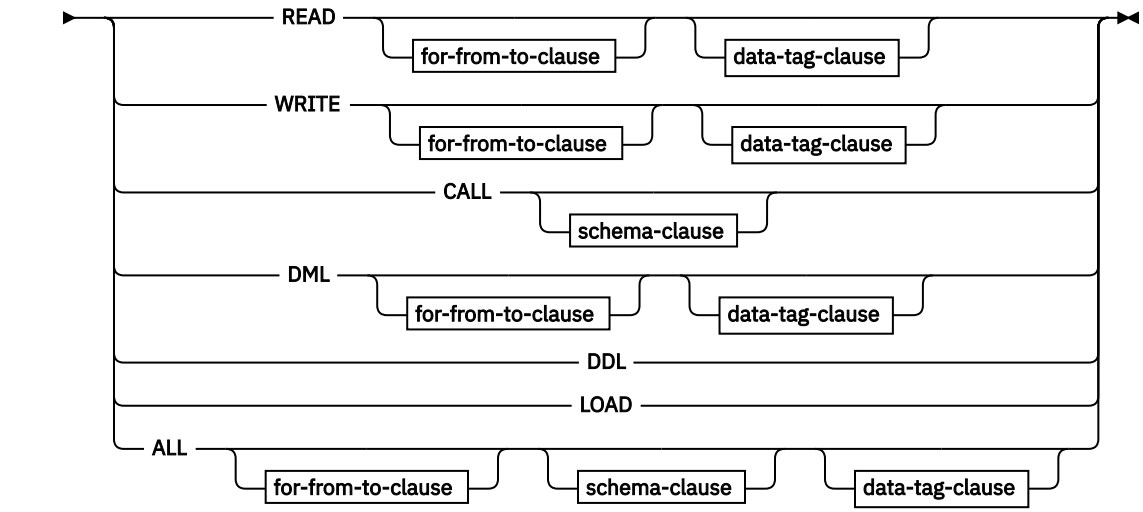
Syntax



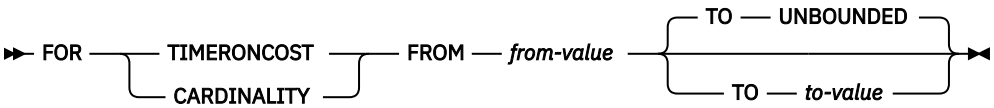
work-class-definition



work-attributes

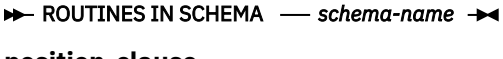


for-from-to-clause

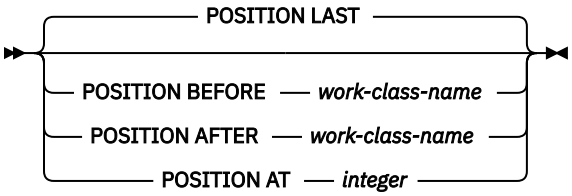


data-tag-clause

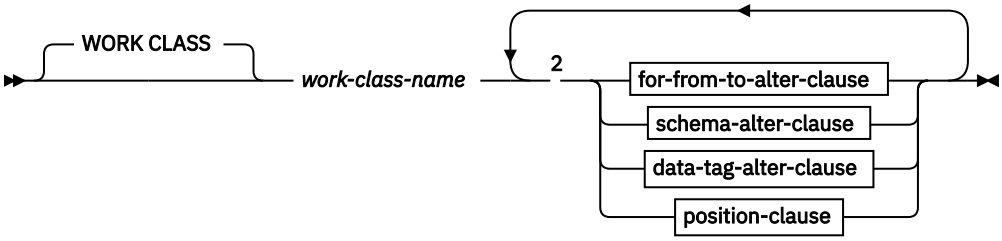
schema-clause



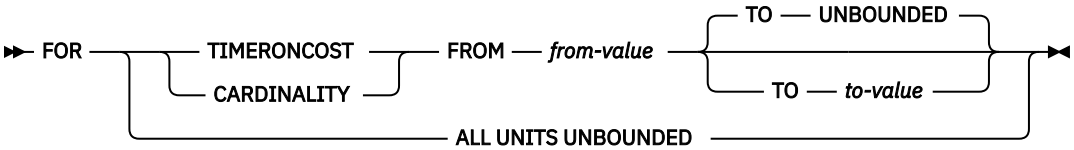
position-clause



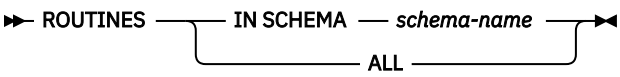
work-class-alteration



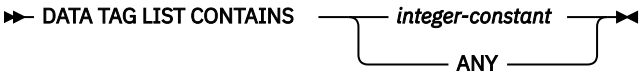
for-from-to-alter-clause



schema-alter-clause



data-tag-alter-clause



Notes:

[1] The ADD, ALTER, and DROP clauses are processed in the order in which they are specified.
[2] The same clause must not be specified more than once.

Description

work-class-set-name
Identifies the work class set that is to be altered. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *work-class-set-name* must identify a work class set that exists at the current server (SQLSTATE 42704).

ADD
Adds a work class to the work class set. For details, see "CREATE WORK CLASS SET".

ALTER
Alters the database activity attributes and the position of a specific work class within the work class set.

WORK CLASS *work-class-name*
Identifies the work class to be altered. The *work-class-name* must identify a work class that exists within the work class set at the current server (SQLSTATE 42704).

DROP
Drops the work class from the work class set.

WORK CLASS *work-class-name*
Identifies the work class to be dropped. The *work-class-name* must identify a work class that exists within the work class set at the current server (SQLSTATE 42704). A work class cannot be

dropped if there is a work action in any of the work action sets associated with this work class set that is dependent on it (SQLSTATE 42893).

for-to-from-alter-clause

FOR
Indicates the type of information that is being specified in the FROM *from-value* TO *to-value* clause. The FOR clause is only used for the following work types:

- ALL
- DML
- READ
- WRITE

TIMERONCOST
The estimated cost of the work, in timerons. This value is used to determine whether the work falls within the range specified in the FROM *from-value* TO *to-value* clause.

CARDINALITY
The estimated cardinality of the work. This value is used to determine whether the work falls within the range specified in the FROM *from-value* TO *to-value* clause.

FROM *from-value* TO UNBOUNDED or FROM *from-value* TO *to-value*
Specifies the range of either timeron value (for estimated cost) or cardinality within which the database activity must fall if it is to be part of this work class. The range is inclusive of *from-value* and *to-value*. This range is only used for the following work types:

- ALL
- DML
- READ
- WRITE

FROM *from-value* TO UNBOUNDED
The *from-value* must be zero or a positive DOUBLE value (SQLSTATE 5U019). The range has no upper bound.

FROM *from-value* TO *to-value*
The *from-value* must be zero or a positive DOUBLE value and the *to-value* must be a positive DOUBLE value. The *from-value* must be smaller than or equal to the *to-value* (SQLSTATE 5U019).

ALL UNITS UNBOUNDED
Indicates that no range is to be specified in the FROM *from-value* TO *to-value* clause, and that all work that falls within the specified work type is to be included.

schema-alter-clause

ROUTINES
This clause is only used if the work type is CALL or ALL and the database activity is a CALL statement.

IN SCHEMA *schema-name*
Specifies the schema name of the procedure that the CALL statement will be calling.

ALL
Specifies that all schemas are included.

data-tag-alter-clause

DATA TAG LIST CONTAINS *integer-constant*
Specifies the value of the tag given to any data which the database activity might touch if it is to be part of this work class. If the clause is not specified for the work class, all work that falls within the specified work type, regardless of what data it might touch, will be included (that is, the default is to ignore the data tag). This clause is used only if the work type is READ, WRITE,

DML, or ALL and the database activity is a DML statement. Valid values for *integer-constant* are integers from 1 to 9.

DATA TAG LIST CONTAINS ANY
Indicates that any data tag setting, including no data tag, is valid for the work class. All work that falls within the specified work type is to be included, regardless of the data tag.

position-clause

POSITION
Specifies where this work class is to be placed within the work class set, which determines the order in which work classes are evaluated. When performing work class assignment at run time, the database manager first determines the work class set that is associated with the object, either the database or a service superclass. The first matching work class within that work class set is then selected. If this keyword is not specified, the work class is placed in the last position.

LAST
Specifies that the work class is to be placed last in the ordered list of work classes within the work class set.

BEFORE *work-class-name*
Specifies that the work class is to be placed before work class *work-class-name* in the list. The *work-class-name* must identify a work class in the work class set that exists at the current server (SQLSTATE 42704).

AFTER *work-class-name*
Specifies that the work class is to be placed after work class *work-class-name* in the list. The *work-class-name* must identify a work class in the work class set that exists at the current server (SQLSTATE 42704).

AT *position*
Specifies the absolute position at which the work class is to be placed within the work class set in the ordered list of work classes. This value can be any positive integer (not zero) (SQLSTATE 42615). If *position* is greater than the number of existing work classes plus one, the work class is placed at the last position within the work class set.

Rules

- A workload management (WLM)-exclusive SQL statement must be followed by a COMMIT or a ROLLBACK statement (SQLSTATE 5U021). WLM-exclusive SQL statements are:
  - CREATE HISTOGRAM TEMPLATE, ALTER HISTOGRAM TEMPLATE, or DROP (HISTOGRAM TEMPLATE)
  - CREATE SERVICE CLASS, ALTER SERVICE CLASS, or DROP (SERVICE CLASS)
  - CREATE THRESHOLD, ALTER THRESHOLD, or DROP (THRESHOLD)
  - CREATE WORK ACTION SET, ALTER WORK ACTION SET, or DROP (WORK ACTION SET)
  - CREATE WORK CLASS SET, ALTER WORK CLASS SET, or DROP (WORK CLASS SET)
  - CREATE WORKLOAD, ALTER WORKLOAD, or DROP (WORKLOAD)
  - GRANT (Workload Privileges) or REVOKE (Workload Privileges)
- A WLM-exclusive SQL statement cannot be issued within a global transaction (SQLSTATE 51041) such as, for example, an XA transaction.

Notes

- Changes are written to the system catalog, but do not take effect until they are committed, even for the connection that issues the statement.
- Only one uncommitted WLM-exclusive SQL statement at a time is allowed across all partitions. If an uncommitted WLM-exclusive SQL statement is executing, subsequent WLM-exclusive SQL statements will wait until the current WLM-exclusive SQL statement commits or rolls back.

Examples

- *Example 1:* Alter work class set LARGE_QUERIES and set the two existing work classes to have each range starting at 100 000, keeping the range unbounded. Add a third work class for all SELECT statements that have an estimated timeron cost greater than or equal to 10 000, and position this work class to take priority over the existing two work classes.

```
ALTER WORK CLASS SET LARGE_QUERIES
  ALTER WORK CLASS LARGE_ESTIMATED_COST
   FOR TIMERONCOST FROM 100000 TO UNBOUNDED
  ALTER WORK CLASS LARGE_CARDINALITY
   FOR CARDINALITY FROM 100000 TO UNBOUNDED
  ADD WORK CLASS LARGE_SELECTS WORK TYPE READ
   FOR TIMERONCOST FROM 10000 TO UNBOUNDED POSITION AT 1
```

- *Example 2:* Alter a work class set named DML_STATEMENTS to add a work class that represents all DML SELECT statements that contain a DELETE, INSERT, MERGE, or UPDATE statement.

```
ALTER WORK CLASS SET DML_STATEMENTS
  ADD WORK CLASS UPDATE_CLASS WORK TYPE WRITE
```

ALTER WORKLOAD

The ALTER WORKLOAD statement alters a workload.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

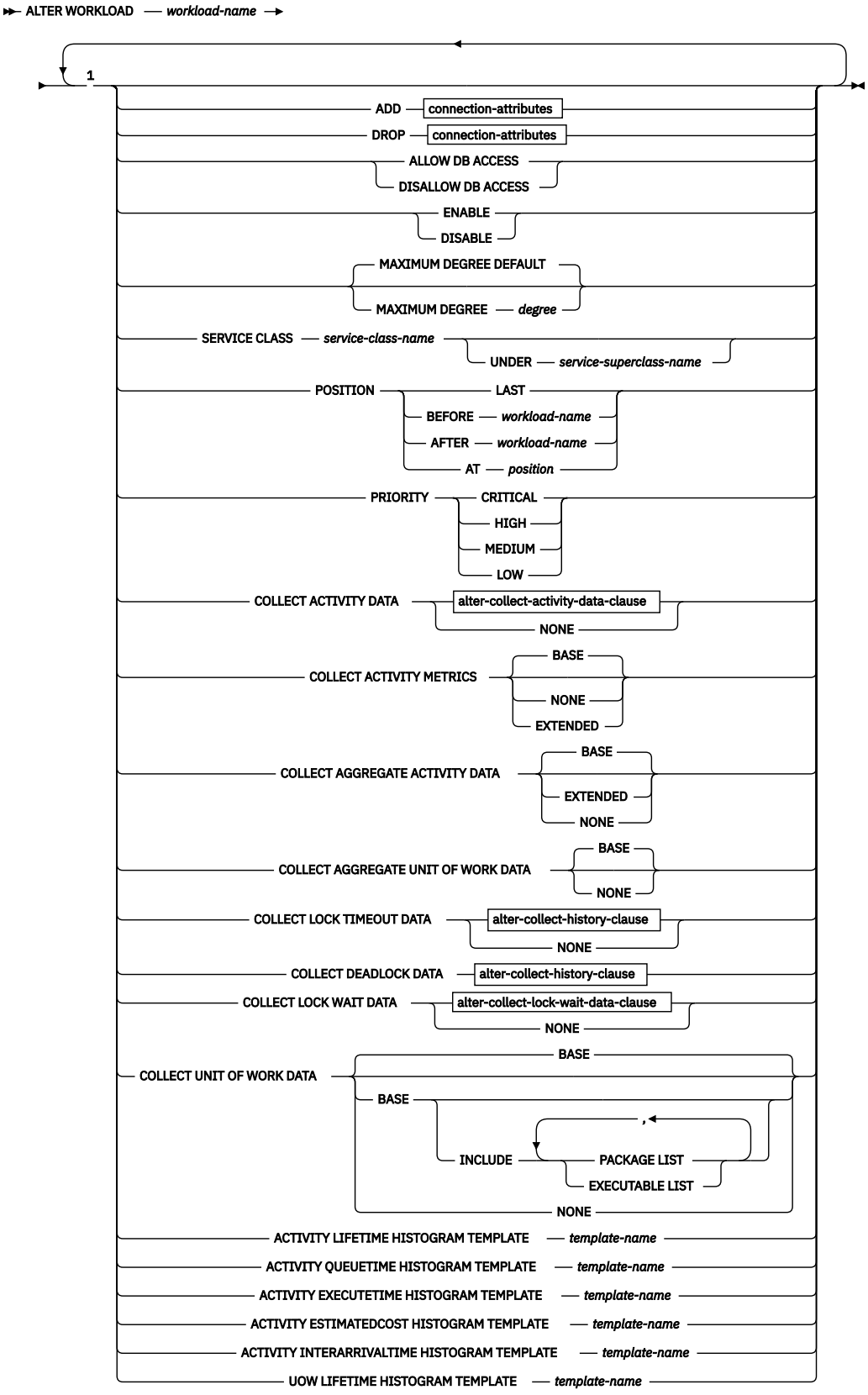
Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

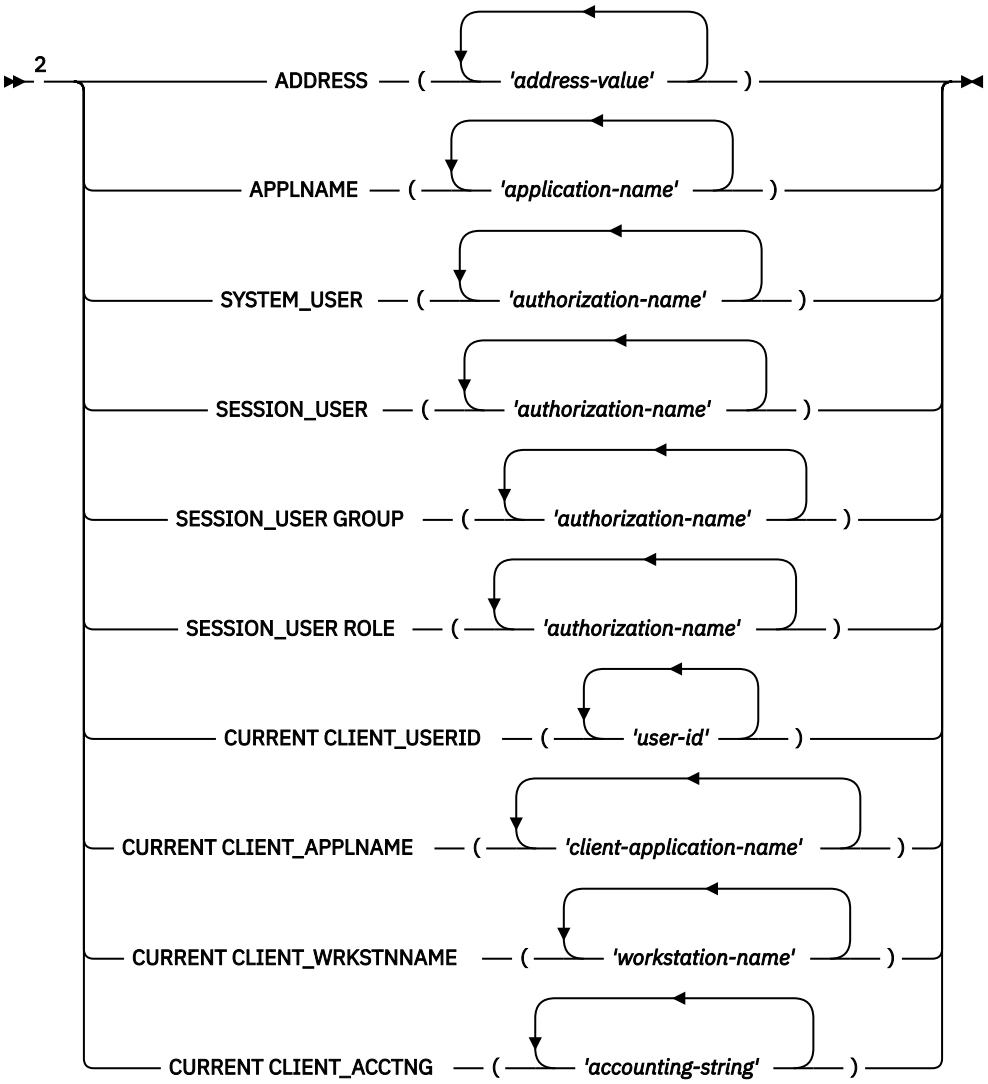
- SQLADM authority, only if every alteration clause is a COLLECT clause
- WLMADM authority
- DBADM authority

To specify any clause other than a COLLECT clause, the authorization ID of the statement must include DBADM or WLMADM authority.

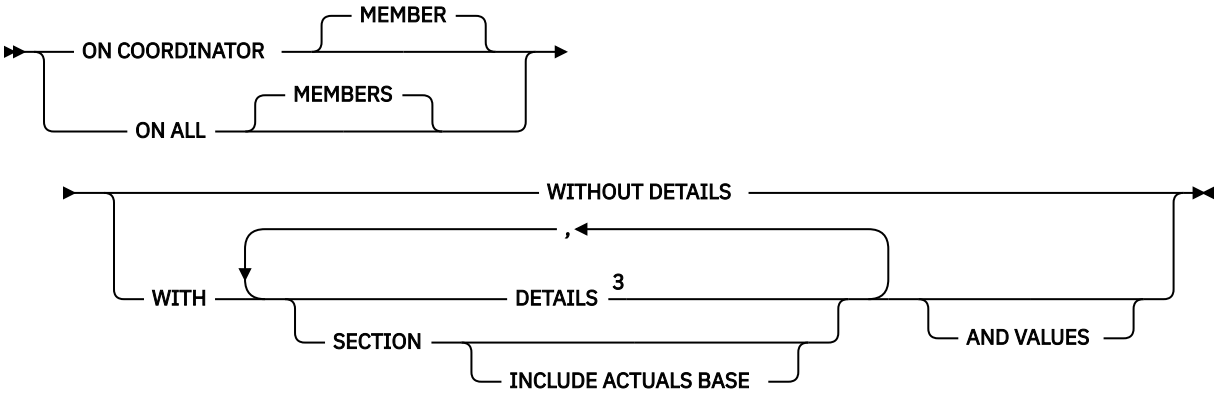
Syntax



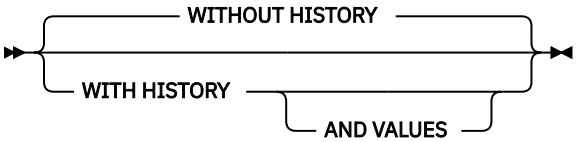
connection-attributes



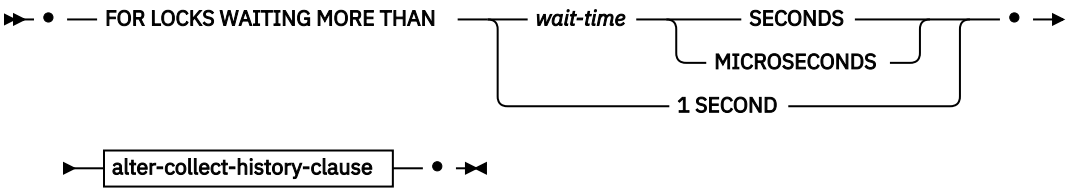
alter-collect-activity-data-clause



alter-collect-history-clause



alter-collect-lock-wait-data-clause



Notes:

[1] The same clause must not be specified more than once.

[2] Each connection attribute clause can only be specified once.

[3] The DETAILS keyword is the minimum to be specified, followed by the option separated by a comma.

Description

workload-name

Identifies the workload that is to be altered. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *workload-name* must identify a workload that exists at the current server (SQLSTATE 42704).

ADD *connection-attributes*

Adds one or more connection attribute values to the definition of the workload. Each specified connection attribute value must not already be defined for the workload (SQLSTATE 5U039). The ADD option cannot be specified if *workload-name* is 'SYSDEFAULTUSERWORKLOAD' or 'SYSDEFAULTADMWORKLOAD' (SQLSTATE 42832).

DROP *connection-attributes*

Drops one or more connection attribute values from the definition of the workload. Each specified connection attribute value must be defined for the workload (SQLSTATE 5U040). The DROP option cannot be specified if *workload-name* is 'SYSDEFAULTUSERWORKLOAD' or 'SYSDEFAULTADMWORKLOAD' (SQLSTATE 42832). There must be at least one defined connection attribute value. The last connection attribute value cannot be dropped (SQLSTATE 5U022).

connection-attributes

Specifies connection attribute values for the workload. All connection attributes are case sensitive, except for ADDRESS.

ADDRESS ('*address-value*', ...)

Specifies one or more IPv4 addresses, IPv6 addresses, or secure domain names for the ADDRESS connection attribute. An address value cannot appear more than once in the list (SQLSTATE 42713). The only supported protocol is TCP/IP. Each address value must be an IPv4 address, an IPv6 address, or a secure domain name.

An IPv4 address must not contain leading spaces and is represented as a dotted decimal address. An example of an IPv4 address is `9.112.46.111`. The value `localhost` or its equivalent representation `127.0.0.1` will not result in a match; the real IPv4 address of the host must be specified instead. An IPv6 address must not contain leading spaces and is represented as a colon hexadecimal address. An example of an IPv6 address is `2001:0DB8:0000:0000:0008:0800:200C:417A`. IPv4-mapped IPv6 addresses (`::ffff:192.0.2.128`, for example) will not result in a match. Similarly, `localhost` or its IPv6 short representation `::1` will not result in a match. A domain name is converted to an IP address by the domain name server where a resulting IPv4 or IPv6 address is determined. An example of a domain name is `corona.torolab.ibm.com`. When a domain name is converted to an IP address, the result of this conversion could be a set of one or more IP addresses. In this case, an incoming connection is said to match the ADDRESS attribute of a workload object if the IP address from which the connection originates matches any of the IP addresses to which the domain name was converted.

When creating a workload object, you should specify domain name values for the ADDRESS attribute instead of static IP addresses, particularly in Dynamic Host Configuration Protocol (DHCP) environments where a device can have a different IP address each time it connects to the network.

APPLNAME ('*application-name*', ...)
Specifies one or more applications for the APPLNAME connection attribute. An application name cannot appear more than once in the list (SQLSTATE 42713). If *application-name* does not contain a single asterisk character (*), is equivalent to the value shown in the "Application name" field in system monitor output and in output from the LIST APPLICATIONS command. If *application-name* does contain a single asterisk character (*), the value is used as an expression to represent a set of application names, where the asterisk (*) represents a string of zero or more characters. If the expression needs to include an asterisk character in the application name, use a sequence of two asterisk characters (**).

SYSTEM_USER ('*authorization-name*', ...)
Specifies one or more authorization IDs for the SYSTEM USER connection attribute. An authorization ID cannot appear more than once in the list (SQLSTATE 42713).

SESSION_USER ('*authorization-name*', ...)
Specifies one or more authorization IDs for the SESSION USER connection attribute. An authorization ID cannot appear more than once in the list (SQLSTATE 42713).

SESSION_USER GROUP ('*authorization-name*', ...)
Specifies one or more authorization IDs for the SESSION_USER GROUP connection attribute. An authorization ID cannot appear more than once in the list (SQLSTATE 42713).

SESSION_USER ROLE ('*authorization-name*', ...)
Specifies one or more authorization IDs for the SESSION_USER ROLE connection attribute. The roles of a session authorization ID in this context refer to all the roles that are available to the session authorization ID, regardless of how the roles were obtained. An authorization ID cannot appear more than once in the list (SQLSTATE 42713).

CURRENT CLIENT_USERID ('*user-id*', ...)
Specifies one or more client user IDs for the CURRENT CLIENT_USERID connection attribute. A client user ID cannot appear more than once in the list (SQLSTATE 42713). If *user-id* contains a single asterisk character (*), the value is used as an expression to represent a set of user IDs, where the asterisk (*) represents a string of zero or more characters. If the expression needs to include an asterisk character in the user ID, use a sequence of two asterisk characters (**).

CURRENT CLIENT_APPLNAME ('*client-application-name*', ...)
Specifies one or more applications for the CURRENT CLIENT_APPLNAME connection attribute. An application name cannot appear more than once in the list (SQLSTATE 42713). If *client-application-name* does not contain a single asterisk character (*), is equivalent to the value shown in the "TP Monitor client application name" field in system monitor output. If *client-application-name* does contain a single asterisk character (*), the value is used as an expression to represent a set of application names, where the asterisk (*) represents a string of zero or more characters. If the expression needs to include an asterisk character in the application name, use a sequence of two asterisk characters (**).

CURRENT CLIENT_WRKSTNNAME ('*workstation-name*', ...)
Specifies one or more client workstation names for the CURRENT CLIENT_WRKSTNNAME connection attribute. A client workstation name cannot appear more than once in the list (SQLSTATE 42713). If *workstation-name* contains a single asterisk character (*), the value is used as an expression to represent a set of workstation names, where the asterisk (*) represents a string of zero or more characters. If the expression needs to include an asterisk character in the workstation name, use a sequence of two asterisk characters (**).

CURRENT CLIENT_ACCTNG ('*accounting-string*', ...)
Specifies one or more client accounting strings for the CURRENT CLIENT_ACCTNG connection attribute. A client accounting string cannot appear more than once in the list (SQLSTATE 42713). If *accounting-string* contains a single asterisk character (*), the value is used as an expression to represent a set of accounting strings, where the asterisk (*) represents a string of zero or more characters. If the expression needs to include an asterisk character in the accounting string, use a sequence of two asterisk characters (**).

ALLOW DB ACCESS or DISALLOW DB ACCESS
Specifies whether or not a workload occurrence associated with this workload is allowed access to the database.

ALLOW DB ACCESS
Specifies that workload occurrences associated with this workload are allowed access to the database.

DISALLOW DB ACCESS
Specifies that workload occurrences associated with this workload are not allowed access to the database. The next unit of work associated with this workload will be rejected (SQLSTATE 5U020). Workload occurrences that are already running are allowed to complete. This option cannot be specified if *workload-name* is 'SYSDEFAULTADMWORKLOAD' (SQLSTATE 42832).

ENABLE or DISABLE
Specifies whether or not this workload will be considered when a workload is chosen.

ENABLE
Specifies that the workload is enabled and will be considered when a workload is chosen.

DISABLE
Specifies that the workload is disabled and will not be considered when a workload is chosen. This option cannot be specified if *workload-name* is SYSDEFAULTUSERWORKLOAD or SYSDEFAULTADMWORKLOAD (SQLSTATE 42832).

MAXIMUM DEGREE
Specifies the maximum runtime degree of parallelism for this workload. The MAXIMUM DEGREE attribute can not be altered if *workload-name* is SYSDEFAULTADMWORKLOAD.

DEFAULT
If `DB2_WORKLOAD=ANALYTICS`, this setting enables intrapartition parallelism for this workload. Otherwise, this setting specifies that this workload inherits the intrapartition parallelism setting from the database manager configuration parameter **intra_parallel**. When **intra_parallel** is set to `NO`, this workload runs with intrapartition parallelism disabled. When **intra_parallel** is set to YES, this workload runs with intrapartition parallelism enabled. This workload does not specify a maximum runtime degree for assigned applications. Therefore, the actual runtime degree is determined as the lower of the value of **max_querydegree** configuration parameter, the MAXIMUM DEGREE set on the query service class, the value set by SET RUNTIME DEGREE command, and the SQL statement compilation degree.

degree
Specifies the maximum degree of parallelism for this workload. Valid values are 1 to 32,767. With value 1, the associated requests run with intrapartition parallelism disabled. With value 2 to 32,767, the associated requests run with intrapartition parallelism enabled. The actual runtime degree is determined as the lower of this *degree*, the value of **max_querydegree** configuration parameter, the MAXIMUM DEGREE set on the query service class, the value set by SET RUNTIME DEGREE command and the SQL statement compilation degree.

SERVICE CLASS *service-class-name*
Specifies that requests associated with this workload are to be executed in the service class *service-class-name*. The *service-class-name* must identify a service class that exists at the current server (SQLSTATE 42704). The *service-class-name* cannot be 'SYSDEFAULTSUBCLASS', 'SYSDEFAULTSYSTEMCLASS', or 'SYSDEFAULTMAINTENANCECLASS' (SQLSTATE 5U032). This option cannot be specified if *workload-name* is 'SYSDEFAULTADMWORKLOAD' (SQLSTATE 42832).

UNDER *service-superclass-name*
This clause is used when specifying a service subclass. The *service-superclass-name* identifies the service superclass of *service-class-name*. The *service-superclass-name* must identify a service superclass that exists at the current server (SQLSTATE 42704). The *service-superclass-name* cannot be 'SYSDEFAULTSYSTEMCLASS' or 'SYSDEFAULTMAINTENANCECLASS' (SQLSTATE 5U032).

POSITION
Specifies where this workload is to be placed within the ordered list of workloads. At run time, this list is searched in order for the first workload that matches the required connection attributes. This option cannot be specified if *workload-name* is 'SYSDEFAULTUSERWORKLOAD' or 'SYSDEFAULTADMWORKLOAD' (SQLSTATE 42832).

LAST
Specifies that the workload is to be last in the list, before the default workloads SYSDEFAULTUSERWORKLOAD and SYSDEFAULTADMWORKLOAD.

BEFORE *relative-workload-name*
Specifies that the workload is to be placed before workload *relative-workload-name* in the list. The *relative-workload-name* must identify a workload that exists at the current server (SQLSTATE 42704). The BEFORE option cannot be specified if *relative-workload-name* is 'SYSDEFAULTUSERWORKLOAD' or 'SYSDEFAULTADMWORKLOAD' (SQLSTATE 42832).

AFTER *relative-workload-name*
Specifies that the workload is to be placed after workload *relative-workload-name* in the list. The *relative-workload-name* must identify a workload that exists at the current server (SQLSTATE 42704). The AFTER option cannot be specified if *relative-workload-name* is 'SYSDEFAULTUSERWORKLOAD' or 'SYSDEFAULTADMWORKLOAD' (SQLSTATE 42832).

AT *position*
Specifies the absolute position at which the workload is to be placed in the list. This value can be any positive integer (not zero) (SQLSTATE 42615). If *position* is greater than the number of existing workloads plus one, the workload is placed at the last position, just before SYSDEFAULTUSERWORKLOAD and SYSDEFAULTADMWORKLOAD.

PRIORITY
Specifies the priority of the work from this workload compared to that of the work in other workloads in the same service superclass. Within a service superclass priority is used to prioritize more important jobs over less important jobs. Work scheduling across superclasses does not use the priority for scheduling, but instead uses only resource-based scheduling.

COLLECT ACTIVITY DATA
Specifies that data about each activity associated with this workload is to be sent to any active activities event monitor when the activity completes.

alter-collect-activity-data-clause

ON COORDINATOR MEMBER
Specifies that activity data is to be collected only at the coordinator member of the activity.

ON ALL MEMBERS
Specifies that activity data is to be collected at all members where the activity is processed. On remote members, a record for the activity may be captured multiple times as the activity comes and goes on those members. If the AND VALUES clause is specified, activity input values will be collected only for the members of the coordinator.

WITHOUT DETAILS
Specifies that data about each activity that is associated with this workload is to be sent to any active activities event monitor, when the activity completes execution. Details about statement, compilation environment, and section environment data are not sent.

WITH

DETAILS
Specifies that statement and compilation environment data is to be sent to any active activities event monitor, for those activities that have them. Section environment data is not sent.

SECTION
Specifies that statement, compilation environment, section environment data, and section actuals are to be sent to any active activities event monitor for those activities that have them. DETAILS must be specified if SECTION is specified. Section actuals will be collected on any member where the activity data is collected.

INCLUDE ACTUALS BASE
Specifies that section actuals should also be collected on any partition where the activity data is collected. For section actuals to be collected, either INCLUDE ACTUALS clause must be specified or the **section_actuals** database configuration parameter must be set.

The effective setting for the collection of section actuals is the combination of the INCLUDE ACTUALS clause, the **section_actuals** database configuration parameter, and the <collectsectionactuals> setting specified on the WLM_SET_CONN_ENV routine. For example, if INCLUDE ACTUALS BASE is specified, yet the **section_actuals** database configuration parameter value is NONE and <collectsectionactuals> is set to NONE, then the effective setting for the collection of section actuals is BASE.

BASE specifies that the following actuals should be enabled and collected during the activity's execution:

- Basic operator cardinality counts
- Statistics for each object referenced (DML statements only)

AND VALUES
Specifies that input data values are to be sent to any active activities event monitor, for those activities that have them. This data does not include SQL statements that are compiled by using the REOPT ALWAYS bind option.

NONE
Specifies that activity data is not collected for each activity that is associated with this workload.

COLLECT ACTIVITY METRICS
Specifies that monitor metrics should be collected for an activity submitted by an occurrence of the workload. The default is COLLECT ACTIVITY METRICS NONE.

The effective activity metrics collection setting is the combination of the attribute specified by the COLLECT ACTIVITY METRICS clause on the workload submitting the activity, and the MON_ACT_METRICS database configuration parameter. If either the workload attribute or the configuration parameter has a value other than NONE, metrics will be collected for the activity.

NONE
Specifies that no metrics will be collected for any activity submitted by an occurrence of the workload.

BASE
Specifies that basic metrics will be collected for any activity submitted by an occurrence of the workload.

EXTENDED
Specifies that basic metrics will be collected for any activity submitted by an occurrence of the workload. In addition, specifies that the values for the following monitor elements should be determined with additional granularity:

- **total_section_time**
- **total_section_proc_time**
- **total_routine_user_code_time**
- **total_routine_user_code_proc_time**
- **total_routine_time**

COLLECT AGGREGATE ACTIVITY DATA
Specifies that aggregate activity data about the activities associated with this workload is to be sent to the statistics event monitor, if one is active. This information is collected periodically on an interval that is specified by the **wlm_collect_int** database configuration parameter. The default when COLLECT AGGREGATE ACTIVITY DATA is specified is COLLECT AGGREGATE ACTIVITY DATA BASE.

BASE
Specifies that basic aggregate activity data about the activities associated with this workload is to be sent to the statistics event monitor, if one is active. Basic aggregate activity data includes:

- Estimated activity cost high watermark
- Rows returned high watermark
- Temporary table space usage high watermark. Only activities that have an SQLTEMPSPACE threshold applied to them participate in this high watermark.
- Activity life time histogram
- Activity queue time histogram
- Activity execution time histogram

EXTENDED
Specifies that all aggregate activity data about the activities associated with this workload is to be sent to the statistics event monitor, if one is active. This includes all basic aggregate activity data plus:

- Activity data manipulation language (DML) estimated cost histogram
- Activity DML inter-arrival time histogram

NONE
Specifies that no aggregate activity data is to be collected for this workload.

COLLECT AGGREGATE UNIT OF WORK DATA
Specifies that aggregate unit of work data about the units of work associated with this workload is to be sent to the statistics event monitor, if one is active. This information is collected periodically on an interval that is specified by the **wlm_collect_int** database configuration parameter. The default, when COLLECT AGGREGATE UNIT OF WORK DATA is specified, is COLLECT AGGREGATE UNIT OF WORK DATA BASE.

BASE
Specifies that basic aggregate unit of work data about the units of work associated with this workload is to be sent to the statistics event monitor, if one is active. Basic aggregate unit of work data includes:

- Unit of work lifetime histogram

NONE
Specifies that no aggregate unit of work data is to be collected for this workload.

COLLECT LOCK TIMEOUT DATA
Specifies that data about lock timeout events that occur within this workload is sent to any active locking event monitor when the lock event occurs. The lock timeout data is collected on all members. This setting works in conjunction with the **MON_LOCKTIMEOUT** database configuration setting. The setting that produces the most detailed output is honored.

alter-collect-history-clause

WITHOUT HISTORY
Specifies that data about lock events that occur within this workload is sent to any active locking event monitor when the lock event occurs. Past activity history and input values are not sent to the event monitor.

WITH HISTORY
Specifies to collect past activity history in the current unit of work for all of this type of lock events. The activity history buffer will wrap after the maximum size limit is used.

The default limit on the number of past activities to be kept by any one application is 250. If the number of past activities is greater than the limit, only the newest activities are reported. This default value can be overridden using the registry variable DB2_MAX_INACT_STMTS to specify a different value. You can choose a different value for the limit to increase or reduce the amount of system monitor heap used for past activity information.

AND VALUES
Specifies that input data values are to be sent to any active locking event monitor for those activities that have them. These data values will not include LOB data, LONG VARCHAR data, LONG VARGRAPHIC data, structured type data, or XML data. For SQL statements compiled using the REOPT ALWAYS bind option, there will be no REOPT compilation or statement execution data values provided in the event information.

NONE
Specifies that lock timeout data for the workload is not collected at any member.

COLLECT DEADLOCK DATA
Specifies that data about deadlock events that occur within this workload is sent to any active locking event monitor when the lock event occurs. The deadlock data is collected on all members. This setting is only honored if the **MON_DEADLOCK** database configuration parameter is not set to NONE.

alter-collect-history-clause

WITHOUT HISTORY
Specifies that data about lock events that occur within this workload is sent to any active locking event monitor when the lock event occurs. Past activity history and input values are not sent to the event monitor.

WITH HISTORY
Specifies to collect past activity history in the current unit of work for all of these type of lock events. The activity history buffer will wrap after the maximum size limit is used.

The default limit on the number of past activities to be kept by any one application is 250. If the number of past activities is greater than the limit, only the newest activities are reported. This default value can be overridden using the registry variable DB2_MAX_INACT_STMTS to specify a different value. You can choose a different value for the limit to increase or reduce the amount of system monitor heap used for past activity information.

AND VALUES
Specifies that input data values are to be sent to any active locking event monitor for those activities that have them. These data values will not include LOB data, LONG VARCHAR data, LONG VARGRAPHIC data, structured type data, or XML data. For SQL statements compiled using the REOPT ALWAYS bind option, there will be no REOPT compilation or statement execution data values provided in the event information.

COLLECT LOCK WAIT DATA
Specifies that data about lock wait events that occur within this workload is sent to any active locking even monitor when the lock has not been acquired within *wait-time*. This setting works in conjunction with the **mon_lockwait** and **mon_lw_thresh** database configuration parameters. The setting that produces the most detailed output is honored.

alter-collect-lock-wait-data-clause

FOR LOCKS WAITING MORE THAN *wait-time* SECONDS | MICROSECONDS) | 1 SECOND
Specifies that data about lock wait events that occur within this workload is sent to the applicable event monitor when the lock has not been acquired within *wait-time*.

This value can be any non-negative integer. Use a valid duration keyword to specify an appropriate unit of time for *wait-time*. The minimum valid value for the *wait-time* parameter is 1000 microseconds.

WITH HISTORY
Specifies to collect past activity history in the current unit of work for all of this type of lock events. The activity history buffer will wrap after the maximum size limit is used.

The default limit on the number of past activities to be kept by any one application is 250. If the number of past activities is greater than the limit, only the newest activities are reported. This default value can be overridden using the registry variable DB2_MAX_INACT_STMTS to specify a different value. You can choose a different value for the limit to increase or reduce the amount of system monitor heap used for past activity information.

AND VALUES
Specifies that input data values are to be sent to any active locking event monitor for those activities that have them. These data values will not include LOB data, LONG VARCHAR data, LONG VARGRAPHIC data, structured type data, or XML data. For SQL statements compiled using the REOPT ALWAYS bind option, there will be no REOPT compilation or statement execution data values provided in the event information.

NONE
Specifies that the lock wait event for the workload is not collected at any member.

COLLECT UNIT OF WORK DATA
Specifies that data about each unit of work, also referred to as a transaction, associated with this workload is to be sent to the unit of work event monitors, if any have been created, when the unit of work ends. The default is COLLECT UNIT OF WORK BASE. If the **mon_uow_data** database configuration parameter is set to BASE, it takes precedence over the COLLECT UNIT OF WORK DATA parameter. A value of NONE for the **mon_uow_data** indicates that the COLLECT UNIT OF WORK DATA parameters of individual workloads is used.

BASE
Specifies that the base level of data for transactions, associated with this workload, is sent to the unit of work event monitors.

Some of the information reported in a unit of work event are system level request metrics. The collection of these metrics is controlled independently from the collection of the unit of work data. The request metrics are controlled with the COLLECT REQUEST METRICS clause on superclass, or using the **mon_req_metrics** database configuration parameter. The service super class which the workload is associated with, or the service super class of the service subclass which the workload is associated with, must have the collection of request metrics enabled in order for the request metrics to be present in the unit of work event. If the request metrics collection is not enabled, the value of the request metrics will be zero.

INCLUDE PACKAGE LIST
Specifies that base level of data and the package list for transactions associated with this workload are sent to the unit of work event monitor.

The size of the collected package list is determined by the value of the **mon_pkglist_sz** database configuration parameter. If this value is 0, then the package list is not collected even if the PACKAGE LIST option is specified.

In a partitioned database environment, the package list is only available on the coordinator member. The BASE level will be collected on remote members.

Some of the information reported in a unit of work event are system level request metrics. The collection of these metrics is controlled independently from the collection of the unit of work data. The request metrics are controlled with the COLLECT REQUEST METRICS clause on superclass, or using the **mon_req_metrics** database configuration parameter. The service super class which the workload is associated with, or the service super class of the service subclass which the workload is associated with, must have the collection of request metrics enabled in order for the request metrics to be present in the unit of work event. If the request metrics collection is not enabled, the value of the request metrics will be zero.

INCLUDE EXECUTABLE LIST
Specifies that executable ID list will be collected for a unit of work together with base level of data and sent to the unit of work event monitor.

NONE
Specifies that no unit of work data for transactions associated with this workload is sent to the unit of work event monitor.

ACTIVITY LIFETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the duration, in milliseconds, of database activities running in the workload during a specific interval. This time

includes both time queued and time executing. This information is collected only when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

ACTIVITY QUEUETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, that database activities running in the workload are queued during a specific interval. This information is collected only when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

ACTIVITY EXECUTETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, that database activities running in the workload are executing during a specific interval. This time does not include the time spent queued. Activity execution time is collected in this histogram at the coordinator member only. The time does not include idle time. Idle time is the time between the execution of requests belonging to the same activity when no work is being done. An example of idle time is the time between the end of opening a cursor and the start of fetching from that cursor. This information is collected only when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

ACTIVITY ESTIMATEDCOST HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the estimated cost, in timerons, of DML activities running in the workload. This information is collected only when the COLLECT AGGREGATE ACTIVITY DATA clause is specified with the EXTENDED option.

ACTIVITY INTERARRIVALTIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, between the arrival of one DML activity into this workload and the arrival of the next DML activity into this workload. This information is collected only when the COLLECT AGGREGATE ACTIVITY DATA clause is specified with the EXTENDED option.

UOW LIFETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the duration, in milliseconds, of units of work running in the workload during a specific interval. The default is SYSDEFAULTHISTOGRAM. This information is collected only when the COLLECT AGGREGATE UNIT OF WORK DATA clause is specified with the BASE option.

Rules

- A workload management (WLM)-exclusive SQL statement must be followed by a COMMIT or a ROLLBACK statement (SQLSTATE 5U021). WLM-exclusive SQL statements are:
  - CREATE HISTOGRAM TEMPLATE, ALTER HISTOGRAM TEMPLATE, or DROP (HISTOGRAM TEMPLATE)
  - CREATE SERVICE CLASS, ALTER SERVICE CLASS, or DROP (SERVICE CLASS)
  - CREATE THRESHOLD, ALTER THRESHOLD, or DROP (THRESHOLD)
  - CREATE WORK ACTION SET, ALTER WORK ACTION SET, or DROP (WORK ACTION SET)
  - CREATE WORK CLASS SET, ALTER WORK CLASS SET, or DROP (WORK CLASS SET)
  - CREATE WORKLOAD, ALTER WORKLOAD, or DROP (WORKLOAD)
  - GRANT (Workload Privileges) or REVOKE (Workload Privileges)
- A WLM-exclusive SQL statement cannot be issued within a global transaction (SQLSTATE 51041) such as, for example, an XA transaction.

Notes

- Changes are written to the system catalog, but do not take effect until they are committed, even for the connection that issues the statement. For newly submitted workload occurrences, changes take effect after the ALTER WORKLOAD statement commits. For active workload occurrences, changes take effect at the beginning of the next unit of work.

- Only one uncommitted WLM-exclusive SQL statement at a time is allowed across all partitions. If an uncommitted WLM-exclusive SQL statement is executing, subsequent WLM-exclusive SQL statements will wait until the current WLM-exclusive SQL statement commits or rolls back.
- If the DISABLE option is specified, the workload is disabled after the statement commits. The workload is not considered the next time that a workload is chosen. If there is an active workload occurrence associated with this workload when the ALTER WORKLOAD statement commits, it continues to run until the end of the current unit of work. At the beginning of the next unit of work, a workload re-evaluation takes place, and the connection becomes associated with a different workload.
- ***Privileges***: The USAGE privilege is not granted to any user, group, or role when a workload is created. To enable use of a workload, grant USAGE privilege on that workload to a user, a group, or a role using the GRANT USAGE ON WORKLOAD statement.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - DATABASE PARTITION can be specified in place of MEMBER, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
  - DATABASE PARTITIONS can be specified in place of MEMBERS, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
  - COLLECT UNIT OF WORK DATA PACKAGE LIST can be specified in place of COLLECT UNIT OF WORK DATA BASE INCLUDE PACKAGE LIST.

Examples

- *Example 1:* The workload PAYROLL is currently positioned such that the workload INVENTORY is considered first when the database manager chooses a workload at run time. Alter the evaluation order so that PAYROLL will be considered first.

```
ALTER WORKLOAD PAYROLL
  POSITION BEFORE INVENTORY
```

- *Example 2:* Alter the evaluation order so that the workload BENCHMARK is evaluated by the database manager before any other workload in the catalog.

```
ALTER WORKLOAD BENCHMARK
  POSITION AT 1
```

- *Example 3:* The workload REPORTS was created with APPLNAME set to appl1, appl2, and appl3, and SYSTEM_USER set to BOB and MARY. Alter the workload to add a new application, appl4 to the application name list, and remove appl2, because it should no longer be mapped to REPORTS.

```
ALTER WORKLOAD REPORTS
  ADD APPLNAME ('appl4')
  DROP APPLNAME ('appl2')
```

- *Example 4:* Assuming a lock event monitor called LOCK exists and is active, create lock event records with statement history for lock timeout events that occur within the workload APP.

```
ALTER WORKLOAD APP
  COLLECT LOCK TIMEOUT DATA WITH HISTORY
```

- *Example 5:* Assuming a lock event monitor called LOCK exists and is active, create lock event records for only deadlock and lock timeout events that occur within the workload PAYROLL on all partitions.

```
ALTER WORKLOAD PAYROLL
  COLLECT DEADLOCK DATA
  COLLECT LOCK TIMEOUT DATA WITHOUT HISTORY
```

- *Example 6:* Assuming a lock event monitor called LOCK exists and is active, create lock event records with statement history and values for deadlock events that occur within the workload INVOICE.

```
ALTER WORKLOAD INVOICE
  COLLECT DEADLOCK DATA WITH HISTORY AND VALUES
```

- *Example 7:* Assuming a lock event monitor called LOCK exists and is active, create lock event records with statement history and values for locks acquired after waiting for more than 150 milliseconds that occur within the workload INVOICE.

```
ALTER WORKLOAD INVOICE
  COLLECT LOCK WAIT DATA FOR LOCKS WAITING MORE THAN 150000
  MICROSECONDS WITH HISTORY AND VALUES
```

- *Example 8:* Alter the workload REPORTS to collect unit of work data and send it to the unit of work event monitor:

```
ALTER WORKLOAD REPORTS
  COLLECT UNIT OF WORK DATA BASE
```

ALTER WRAPPER

The ALTER WRAPPER statement is used to update the properties of a wrapper.

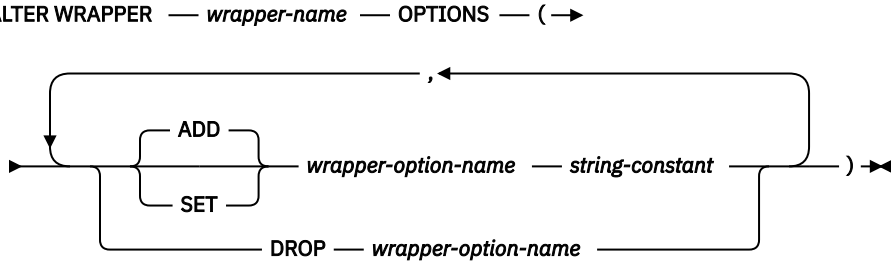
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include DBADM authority.

Syntax



Description

wrapper-name
Specifies the name of the wrapper.

OPTIONS
Indicates what wrapper options are to be enabled, reset, or dropped.

ADD
Enables a new wrapper option.

SET
Changes the setting of a wrapper option.

wrapper-option-name
The wrapper option that is to be added or reset. Which options you can specify depends on the data source of the object for which a wrapper is being created. For a list of data sources and the wrapper options that apply to each, see Data source options.

string-constant
The wrapper option setting as a character string constant enclosed in single quotation marks.

DROP *wrapper-option-name*
Drops a wrapper option.

Notes

- Execution of the ALTER WRAPPER statement does not include checking the validity of wrapper-specific options.
- An ALTER WRAPPER statement within a given unit of work (UOW) cannot be processed (SQLSTATE 55007) if the UOW already includes one of the following items:
  - A SELECT statement that references a nickname that belongs to the wrapper.
  - An open cursor on a nickname that belongs to the wrapper.
  - An INSERT, DELETE, or UPDATE statement issued against a nickname that belongs to the wrapper.

Example

Set the DB2_FENCED option on for wrapper NET8.

```
ALTER WRAPPER NET8 OPTIONS (SET DB2_FENCED 'Y')
```

ALTER XSROBJECT

This statement is used to either enable or disable the decomposition support for a specific XML schema. Annotated XML schemas can be used to decompose XML documents into relational tables, if decomposition has been enabled for those XML schemas.

Invocation

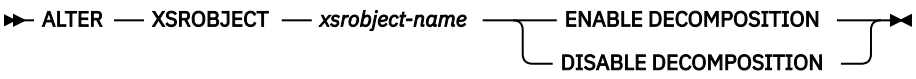
The ALTER XSROBJECT statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if the DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

One of the following authorities is required:

- DBADM
- ALTERIN on the SQL schema
- SCHEMAADM authority on the SQL schema
- Ownership of the XSR object to be altered

Syntax



Description

xsrobject-name
Identifies the XSR object to be altered. The *xsrobject-name*, including the implicit or explicit schema qualifier, must uniquely identify an existing XSR object at the current server. If no XSR object with this identifier exists, an error is returned (SQLSTATE 42704).

ENABLE DECOMPOSITION or DISABLE DECOMPOSITION
Enables or disables the use of the XSR object for decomposition. The identified XSR object must be an XML schema (SQLSTATE 42809). In order to enable decomposition, the XML schema needs to be annotated with decomposition rules (SQLSTATE 225DE) and the objects referenced by the decomposition rules must exist at the current server (SQLSTATE 42704).

Notes

- When decomposition for an XSR object is disabled, all related catalog entries are removed.
- Decomposition support for an XSR object will be disabled if any objects the XSR object depends on (such as tables) are dropped or altered to become incompatible with the XSR object.
- In a partitioned database environment, you can issue this statement by connecting to any partition.

ASSOCIATE LOCATORS

The ASSOCIATE LOCATORS statement gets the result set locator value for each result set returned by a procedure.

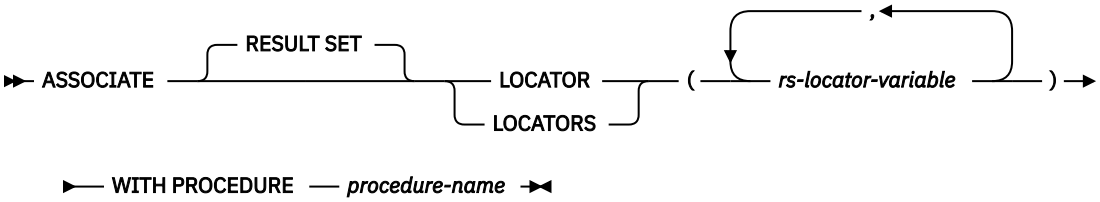
Invocation

This statement can only be embedded in an SQL procedure. It is not an executable statement and cannot be dynamically prepared.

Authorization

None required.

Syntax



Description

rs-locator-variable
Specifies a result set locator variable that has been declared in a compound SQL (Procedure) statement.

WITH PROCEDURE
Identifies the procedure that returns result set locators by the specified procedure name.

procedure-name
A procedure name is a qualified or unqualified name.

A fully qualified procedure name is a two-part name. The first part is an identifier that contains the schema name of the procedure. The last part is an identifier that contains the name of the procedure. A period must separate each of the parts. Any or all of the parts can be a delimited identifier.

If the procedure name is unqualified, it has only one name because the implicit schema name is not added as a qualifier to the procedure name. Successful execution of the ASSOCIATE LOCATOR statement only requires that the unqualified procedure name in the statement be the same as the procedure name in the most recently executed CALL statement that was specified with an unqualified procedure name. The implicit schema name for the unqualified name in the CALL statement is not considered in the match. The rules for how the procedure name must be specified are described in the following paragraph.

When the ASSOCIATE LOCATORS statement is executed, the procedure name or specification must identify a procedure that the requester has already invoked using the CALL statement. The procedure name in the ASSOCIATE LOCATORS statement must be specified the same way that it was specified on the CALL statement. For example, if a two-part name was specified on the CALL statement, you must use a two-part name in the ASSOCIATE LOCATORS statement.

Notes

- If the number of result set locator variables that are listed in the ASSOCIATE LOCATORS statement is less than the number of locators returned by the procedure, all variables in the statement are assigned a value, and a warning is issued.
- If the number of result set locator variables that are listed in the ASSOCIATE LOCATORS statement is greater than the number of locators returned by the procedure, the extra variables are assigned a value of 0.
- If a procedure is called more than once from the same caller, only the most recent result sets are accessible.
- Result set locator values are available for a procedure that is called using an EXECUTE statement executing the CALL statement that was previously prepared by the PREPARE statement. Result set locator values, however, are not available for a procedure that is called using an EXECUTE IMMEDIATE statement.
- Module-procedure names referenced in an ASSOCIATE LOCATORS statement can only be 1-part or 2-part qualified name references. A 3-part name reference is not allowed (SQLSTATE 42601). Any CALL statement that references a module-procedure that was referenced in an ASSOCIATE LOCATORS statement, must specify the module-procedure with the same 1-part or 2-part qualified name used in the ASSOCIATE LOCATORS statement.

Examples

The statements in the following examples are assumed to be embedded in SQL Procedures.

- *Example 1*: Use result set locator variables LOC1 and LOC2 to get the result set locator values for the two result sets returned by procedure P1. Assume that the procedure is called with a one-part name.

```
CALL P1;
ASSOCIATE RESULT SET LOCATORS (LOC1, LOC2)
  WITH PROCEDURE P1;
```

- *Example 2*: Repeat the scenario in Example 1, but use a two-part name to specify an explicit schema name for the procedure to ensure that procedure P1 in schema MYSCHEMA is used.

```
CALL MYSCHEMA.P1;
ASSOCIATE RESULT SET LOCATORS (LOC1, LOC2)
  WITH PROCEDURE MYSCHEMA.P1;
```

AUDIT

The AUDIT statement determines the audit policy that is to be used for a particular database or database object at the current server. Whenever the object is in use, it is audited according to that policy.

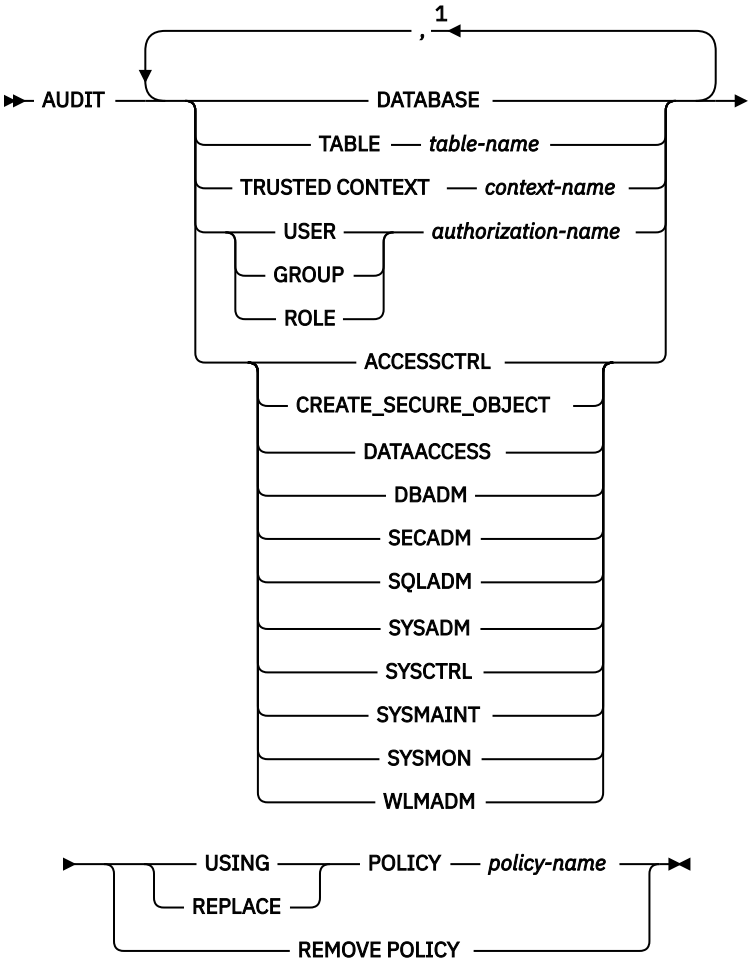
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include SECADM authority.

Syntax



Notes:

[1] Each clause (with the same object name, if applicable) can be specified at most once (SQLSTATE 42713).

Description

ACCESSCTRL, CREATE_SECURE_OBJECT, DATAACCESS, DBADM, SECADM, SQLADM, SYSADM, SYSCTRL, SYSMAINT, SYSMON, or WLMADM
Specifies that an audit policy is to be associated with or removed from the specified authority. All auditable events that are initiated by a user who holds the specified authority, even if that authority is not required for the event, will be audited according to the associated audit policy.

DATABASE
Specifies that an audit policy is to be associated with or removed from the database at the current server. All auditable events that occur within the database are audited according to the associated audit policy.

TABLE *table-name*
Specifies that an audit policy is to be associated with or removed from *table-name*. The *table-name* must identify a table, materialized query table (MQT), or nickname that exists at the current server (SQLSTATE 42704). It cannot be a view, a catalog table, a created temporary table, a declared temporary table (SQLSTATE 42995), or a typed table (SQLSTATE 42997). Only EXECUTE category audit events, with or without data, will be generated when the table is accessed, even if the policy indicates that other categories should be audited.

TRUSTED CONTEXT *context-name*
Specifies that an audit policy is to be associated with or removed from *context-name*. The *context-name* must identify a trusted context that exists at the current server (SQLSTATE 42704). All auditable events that happen within the trusted connection defined by the trusted context *context-name* will be audited according to the associated audit policy.

USER *authorization-name*
Specifies that an audit policy is to be associated with or removed from the user with authorization ID *authorization-name*. All auditable events that are initiated by *authorization-name* will be audited according to the associated audit policy.

GROUP *authorization-name*
Specifies that an audit policy is to be associated with or removed from the group with authorization ID *authorization-name*. All auditable events that are initiated by users who are members of *authorization-name* will be audited according to the associated audit policy. If user membership in a group cannot be determined, the policy will not apply to that user.

ROLE *authorization-name*
Specifies that an audit policy is to be associated with or removed from the role with authorization ID *authorization-name*. The *authorization-name* must identify a role that exists at the current server (SQLSTATE 42704). All auditable events that are initiated by users who are members of *authorization-name* will be audited according to the associated audit policy. Indirect role membership through other roles or groups is valid.

USING, REMOVE, or REPLACE
Specifies whether the audit policy should be used, removed, or replaced for the specified object.

USING
Specifies that the audit policy is to be used for the specified object. An existing audit policy must not already be defined for the object (SQLSTATE 5U041). If an audit policy already exists, it must be removed or replaced.

REMOVE
Specifies that the audit policy is to be removed from the specified object. Use of the object will no longer be audited according to the audit policy. The association is deleted from the catalog when the audit policy is removed from the object.

REPLACE
Specifies that the audit policy is to replace an existing audit policy for the specified object. This combines both REMOVE and USING options into one step to ensure that there is no period of time in which an audit policy does not apply to the specified object. If a policy was not in use for the specified object, REPLACE is equivalent to USING.

POLICY *policy-name*

Specifies the audit policy that is to be used to determine audit settings. The *policy-name* must identify an existing audit policy at the current server (SQLSTATE 42704).

Rules

- An AUDIT-exclusive SQL statement must be followed by a COMMIT or ROLLBACK statement (SQLSTATE 5U021). AUDIT-exclusive SQL statements are:
  - AUDIT
  - CREATE AUDIT POLICY, ALTER AUDIT POLICY, or DROP (AUDIT POLICY)
  - DROP (ROLE or TRUSTED CONTEXT if it is associated with an audit policy)
- An AUDIT-exclusive SQL statement cannot be issued within a global transaction (SQLSTATE 51041) such as, for example, an XA transaction.
- An object can be associated with no more than one policy (SQLSTATE 5U042).

Notes

- Packages that are dependent on the table that is being audited become invalid when an audit policy is added or removed from that table.
- Changes are written to the catalog, but do not take effect until after a COMMIT statement executes.
- Changes do not take effect until the next unit of work that references the object to which the audit policy applies. For example, if the audit policy is in use for the database, no current units of work will begin auditing according to the policy until after a COMMIT or a ROLLBACK statement completes.
- Views accessing a table that is associated with an audit policy are audited according to the underlying table's policy.
- The audit policy that applies to a table does not apply to a materialized query table (MQT) based on that table. It is recommended that if you associate an audit policy with a table, you also associate that policy with any MQT based on that table. The compiler might automatically use an MQT, even though an SQL statement references the base table; however, the audit policy in use for the base table will still be in effect.
- When a switch user operation is performed within a trusted context, all audit policies are re-evaluated according to the new user, and no policies from the old user are used for the current session. This applies specifically to audit policies associated directly with the user, the user's group or role memberships, and the user's authorities. For example, if the current session was audited because the previous user was a member of an audited role, and the switched-to user is not a member of that role, that policy no longer applies to the session.
- When a SET SESSION USER statement is executed, the audit policies associated with the original user (and that user's group and role memberships and authorities) are combined with the policies that are associated with the user specified in the SET SESSION USER statement. The audit policies associated with the original user are still in effect, as are the policies for the user specified in the SET SESSION USER statement. If multiple SET SESSION USER statements are issued within a session, only the audit policies associated with the original user and the current user are considered.
- If the object with which an audit policy is associated is dropped, the association to the audit policy is removed from the catalog and no longer exists. If that object is recreated at some later time, the object will not be audited according to the policy that was associated with it when the object was dropped.
- When multiple objects are specified, the policy applies to each of them individually. The policy applies to the individual objects within the statement, and not their intersection. It is not possible to combine policies with an "AND" type configuration.

Examples

- *Example 1:* Use the audit policy DBAUDPRF to determine the audit settings for the database at the current server.

```
AUDIT DATABASE USING POLICY DBAUDPRF
```

- *Example 2:* Remove the audit policy from the EMPLOYEE table.

```
AUDIT TABLE EMPLOYEE REMOVE POLICY
```

- *Example 3:* Use the audit policy POWERUSERS to determine the audit settings for the authorities SYSADM, DBADM, and SECADM, as well as the group DBAS.

```
AUDIT SYSADM, DBADM, SECADM, GROUP DBAS USING POLICY POWERUSERS
```

- *Example 4:* Replace the audit policy for the role TELLER with the new policy TELLERPRF.

```
AUDIT ROLE TELLER REPLACE POLICY TELLERPRF
```

- *Example 5:* Specifying multiple objects

```
AUDIT TABLE EMPLOYEE, ROLE TELLER USING POLICY TABLEAUDIT
```

The above statement is equivalent to these separate statements:

```
AUDIT TABLE EMPLOYEE USING POLICY TABLEAUDIT
AUDIT ROLE TELLER USING POLICY TABLEAUDIT
```

BEGIN DECLARE SECTION

The BEGIN DECLARE SECTION statement marks the beginning of a host variable declare section.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in REXX.

Authorization

None required.

Syntax

Description

The BEGIN DECLARE SECTION statement may be coded in the application program wherever variable declarations can appear in accordance with the rules of the host language. It is used to indicate the beginning of a host variable declaration section. A host variable section ends with an END DECLARE SECTION statement.

Rules

- The BEGIN DECLARE SECTION and the END DECLARE SECTION statements must be paired and may not be nested.
- SQL statements cannot be included within the declare section.
- Variables referenced in SQL statements must be declared in a declare section in all host languages other than REXX. Furthermore, the section must appear before the first reference to the variable.

Generally, host variables are not declared in REXX with the exception of LOB locators and file reference variables. In this case, they are not declared within a BEGIN DECLARE SECTION.

- Variables declared outside a declare section should not have the same name as variables declared within a declare section.
- LOB data types must have their data type and length preceded with the SQL TYPE IS keywords.

Examples

- *Example 1:* Define the host variables hv_smint (smallint), hv_vchar24 (varchar(24)), hv_double (double), hv_blob_50k (blob(51200)), hv_struct (of structured type "struct_type" as blob(10240)) in a C program.

```
EXEC SQL  BEGIN DECLARE SECTION;
  short  hv_smint;
  struct {
  short hv_vchar24_len;
  char  hv_vchar24_value[24];
 }                              hv_vchar24;
  double hv_double;
  SQL TYPE IS BLOB(50K) hv_blob_50k;
  SQL TYPE IS struct_type AS BLOB(10k) hv_struct;
EXEC SQL  END DECLARE SECTION;
```

- *Example 2:* Define the host variables HV-SMINT (smallint), HV-VCHAR24 (varchar(24)), HV-DEC72 (dec(7,2)), and HV-BLOB-50k (blob(51200)) in a COBOL program.

```
WORKING-STORAGE SECTION.
      EXEC SQL BEGIN DECLARE SECTION  END-EXEC.
01  HV-SMINT              PIC S9(4)       COMP-4.
01  HV-VCHAR24.
    49 HV-VCHAR24-LENGTH  PIC S9(4)       COMP-4.
    49 HV-VCHAR24-VALUE   PIC X(24).
01  HV-DEC72              PIC S9(5)V9(2)  COMP-3.
01 HV-BLOB-50K            USAGE SQL TYPE IS BLOB(50K).
      EXEC SQL END DECLARE SECTION  END-EXEC.
```

- *Example 3:* Define the host variables HVSMINT (smallint), HVVCHAR24 (char(24)), HVDOUBLE (double), and HVBLOB50k (blob(51200)) in a Fortran program.

```
EXEC SQL  BEGIN DECLARE SECTION
  INTEGER*2      HVSMINT
  CHARACTER*24   HVVCHAR24
  REAL*8         HVDOUBLE
  SQL TYPE IS BLOB(50K)  HVBLOB50K
EXEC SQL  END DECLARE SECTION
```

Note: In Fortran, if the expected value is greater than 254 bytes, then a CLOB host variable should be used.

- *Example 4:* Define the host variables HVSMINT (smallint), HVBLOB50K (blob(51200)), and HVCLOBLOC (a CLOB locator) in a REXX program.

```
DECLARE :HVCLOBLOC  LANGUAGE TYPE CLOB LOCATOR
call sqlexec 'FETCH c1 INTO :HVSMINT, :HVBLOB50K'
```

Note that the variables HVSMINT and HVBLOB50K were implicitly defined by using them in the FETCH statement.

CALL

The CALL statement calls a procedure or a foreign procedure.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

When invoked using the command line processor, there are some additional rules for specifying arguments of the procedure.

For more information, refer to "Using command line SQL statements and XQuery statements" in *Command Reference*.

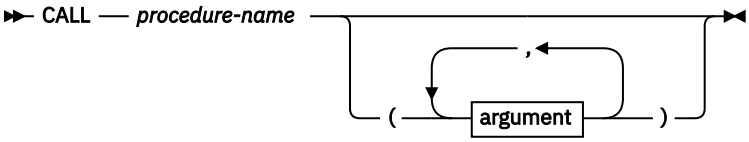
Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

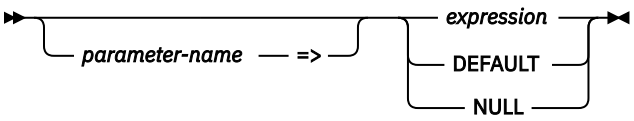
- EXECUTE privilege on the procedure
- EXECUTEIN privilege on the schema containing the procedure
- DATAACCESS authority on the schema containing the procedure
- DATAACCESS authority

If a matching procedure exists that the authorization ID of the statement is not authorized to execute, an error is returned (SQLSTATE 42501).

Syntax



argument



Description

procedure-name
Specifies the procedure that is to be called. It must be a procedure that is described in the catalog or that is declared in the scope of the compound SQL (compiled) statement that includes the CALL statement. The specific procedure to invoke is chosen using procedure resolution. (For more details, see the "Notes" section of this statement.)

argument

parameter-name
Name of the parameter to which the argument is assigned. When an argument is assigned to a parameter by name, then all the arguments that follow it must also be assigned by name (SQLSTATE 4274K).

A named argument must be specified only once (implicitly or explicitly) (SQLSTATE 4274K).

Named arguments are not supported on the call to an uncataloged procedure (SQLSTATE 4274K).

***expression* or DEFAULT or NULL**
Each specification of *expression*, the DEFAULT keyword, or the NULL keyword is an argument of the CALL. The *n*th unnamed argument of the CALL statement corresponds to the *n*th parameter defined in the CREATE PROCEDURE statement for the procedure.

Named arguments correspond to the same named parameter, regardless of the order in which they are specified.

If the DEFAULT keyword is specified, the default as defined in the CREATE PROCEDURE statement is used if it exists; otherwise the null value is used as the default.

If the NULL keyword is specified, the null value is passed as the parameter value.

Each argument of the CALL must be compatible with the corresponding parameter in the procedure definition as follows:

- IN parameter
  - The argument must be assignable to the parameter.
  - The assignment of a string argument uses the storage assignment rules.
- OUT parameter
  - The argument must be a single variable or parameter marker (SQLSTATE 42886).
  - The argument must be assignable to the parameter.
  - The assignment of a string argument uses the retrieval assignment rules.
- INOUT parameter
  - The argument must be a single variable or parameter marker (SQLSTATE 42886).
  - The argument must be assignable to the parameter.
  - The assignment of a string argument uses the storage assignment rules on invocation and the retrieval assignment rules on return.

Notes

- ***Parameter assignments:*** When the CALL statement is executed, the value of each of its arguments is assigned (using storage assignment) to the corresponding parameter of the procedure. A parameter value that is defined to have a default value can be omitted from the argument list when invoking the procedure.

  When the CALL statement is executed, control is passed to the procedure according to the calling conventions of the host language. When execution of the procedure is complete, the value of each parameter of the procedure is assigned (using storage assignment) to the corresponding argument of the CALL statement defined as OUT or INOUT. If an error is returned by the procedure, OUT arguments are undefined and INOUT arguments are unchanged. For details on the assignment rules, see "Assignments and comparisons".

  When the CALL statement is in an SQL procedure and is calling another SQL procedure, assignment of XML parameters is done by reference. When an XML argument is passed by reference, the input node trees, if any, are used directly from the XML argument, preserving all properties, including document order, the original node identities, and all parent properties.
- ***Procedure signatures:*** A procedure is identified by its schema, a procedure name, and the number of parameters. This is called a procedure signature, which must be unique within the database. There can be more than one procedure with the same name in a schema, provided that the number of parameters is different for each procedure.
- ***SQL path:*** A procedure can be invoked by referring to a qualified name (schema and procedure name), followed by an optional list of arguments enclosed by parentheses. A procedure can also be invoked without the schema name, resulting in a choice of possible procedures in different schemas with the same number of parameters. In this case, the SQL path is used to assist in procedure resolution. The SQL path is a list of schemas that is searched to identify a procedure with the same name and number of parameters. For static CALL statements, SQL path is specified using the FUNCPATH bind option. For dynamic CALL statements, SQL path is the value of the CURRENT PATH special register.
- ***Procedure resolution:*** Given a procedure invocation, the database manager must decide which of the possible procedures with the same name to execute.

  Local scope procedure resolution is used when a procedure is invoked from within a compound SQL (compiled) statement and either of the following criteria exist:

  - A procedure with the same name as the invoked procedure is declared in the same compound SQL (compiled) statement

– A procedure with the same name as the invoked procedure is declared in a compound SQL (compiled) statement within which the compound SQL (compiled) statement that invoked the procedure is nested

Local scope procedure resolution means that only declared procedures within the scope of the compound SQL (compiled) statement that invoked the procedure are considered during procedure resolution regardless of the existence of possible matching built-in procedures, schema procedures, or module procedures. *Global scope procedure resolution* is used in all other cases and considers candidates from schemas and modules depending on the context of the invocation and the qualification of the procedure name.

– Let *A* be the number of arguments in a procedure invocation.
– Let *P* be the number of parameters in a procedure signature.
– Let *N* be the number of parameters without a default.

Candidate procedures for resolution of a procedure invocation are selected based on the following criteria:

– Each candidate procedure has a matching name and an applicable number of parameters. An applicable number of parameters satisfies the condition $N \leq A \leq P$.
– Each candidate procedure has parameters such that for each named argument in the CALL statement there exists a parameter with a matching name that does not already correspond to a positional (or unnamed) argument.
– Each parameter of a candidate procedure that does not have a corresponding argument in the CALL statement, specified by either position or name, is defined with a default.
– Each candidate procedure from a set of one or more schemas has the EXECUTE privilege associated with the authorization ID of the statement invoking the function.
– Each candidate procedure from a schema has the EXECUTEIN privilege or DATAACCESS authority on the schema associated with the authorization ID of the statement invoking the function.

In addition, the set of candidate procedures depends on the environment where the procedure is invoked and how the procedure name is qualified.

– If the procedure name is unqualified, procedure resolution is done using the steps that follow:

  1. If the procedure is invoked from within a compound SQL (compiled) statement and a declared procedure with the same name exists in the nested scope, search the set of compound SQL (compiled) statements within which the CALL statement is nested for candidate procedures. If no candidate procedures are found, an error is returned (SQLSTATE 42884). If a single candidate procedure is found, resolution is complete. If there are multiple candidate procedures, determine the candidate procedure with the lowest number of parameters and eliminate candidate procedures with a higher number of parameters.
  2. If the procedure is invoked from within a module object, search within the module for candidate procedures. If one or more candidate procedures are found in the context module, then these candidate procedures are included with any candidate procedures from the schemas in the SQL path (see next item).
  3. Search all schema procedures with a schema in the SQL path for candidate procedures. If one or more candidate procedures are found in the schemas of the SQL path, then these candidate procedures are included with any candidate procedures from the context module (see previous item). If a single candidate procedure remains, resolution is complete. If there are multiple candidate procedures, choose the procedure from the context module if still a candidate and otherwise choose the procedure whose schema is earliest in the SQL path. If there are still multiple candidate procedures, determine the candidate procedure with the lowest number of parameters and eliminate candidate procedures with a higher number of parameters.

  If there are no candidate procedures remaining after step 3, an error is returned (SQLSTATE 42884).

– If the procedure name is qualified, procedure resolution is done using the steps that follow:

1. If the procedure is invoked from within a compound SQL (compiled) statement and a declared procedure with the same name exists where the qualifier matches the label of the compound SQL (compiled) statement from the set of compound SQL (compiled) statements within which the CALL statement is nested, search that compound SQL (compiled) statement with the matching label for candidate procedures. If no candidate procedures are found, an error is returned (SQLSTATE 42884). If a single candidate procedure is found, resolution is complete. If there are multiple candidate procedures, determine the candidate procedure with the lowest number of parameters and eliminate candidate procedures with a higher number of parameters.
2. If the procedure is invoked from within a module and the qualifier matches the name of the module from within which the procedure is invoked, search within the module for candidate procedures. If the qualifier is a single identifier, then the schema name of the module is ignored when matching the module name. If the qualifier is a two part identifier, then it is compared to the schema-qualified module name when determining a match. If a single candidate procedure exists, resolution is complete. If there are multiple candidate procedures, choose the candidate procedure with the least number of parameters. If the qualifier does not match or there are no candidate procedures, then continue with the next step.
3. Consider the qualifier as a schema name and search within that schema for candidate procedures. If a single candidate procedure exists, resolution is complete. If there are multiple candidate procedures, choose the candidate procedure with the least number of parameters and resolution is complete. If the schema does not exist or there are no authorized candidate procedures, and the qualifier matched the name of the module in the first step, then return an error. Otherwise, continue to the next step.
4. Consider the qualifier as a module name, without considering EXECUTE privilege on modules.
   - If the module name is qualified with a schema name, then search published procedures within this module for candidate procedures.
   - If the module name is not qualified with a schema name, then the schema for the module is the first schema in the SQL path that has a matching module name. If found, then search published procedures within this module for candidate procedures.
   - If the module is not found using the SQL path, check for a module public alias that matches the name of the procedure qualifier. If found, then search published procedures within this module for candidate procedures.

   If a matching module is not found or there are no candidate procedures in the matching module, then a procedure not found error is returned (SQLSTATE 42884). If there are multiple candidate procedures, choose the candidate procedure with the least number of parameters. Resolution is complete if the authorization ID of the CALL statement has EXECUTE privilege on the module, or EXECUTEIN privilege on the schema containing the module, of the remaining candidate procedure, otherwise an authorization error is returned (SQLSTATE 42501).

- ***Retrieving the DB2_RETURN_STATUS from an SQL procedure:*** If an SQL procedure successfully issues a RETURN statement with a status value, this value is returned in the first SQLERRD field of the SQLCA. If the CALL statement is issued in an SQL procedure, use the GET DIAGNOSTICS statement to retrieve the DB2_RETURN_STATUS value. The value is -1 if the SQLSTATE indicates an error. The values is 0 if no error is returned and the RETURN statement was not specified in the procedure.
- ***Returning result sets from procedures:*** If the calling program is written using CLI, JDBC, or SQLJ, or the caller is an SQL procedure, result sets can be returned directly to the caller. The procedure indicates that a result set is to be returned by declaring a cursor on that result set, opening a cursor on the result set, and leaving the cursor open when exiting the procedure.

  At the end of a procedure:

  – For every cursor that has been left open, a result set is returned to the caller or (for WITH RETURN TO CLIENT cursors) directly to the client.
  – Only unread rows are passed back. For example, if the result set of a cursor has 500 rows, and 150 of those rows have been read by the procedure at the time the procedure is terminated, rows 151 through 500 will be returned to the caller or application (as appropriate).

If the procedure was invoked from CLI or JDBC, and more than one cursor is left open, the result sets can only be processed in the order in which the cursors were opened.

- ***Improving performance:*** The values of all arguments are passed from the application to the procedure. To improve the performance of this operation, host variables that correspond to OUT parameters and have lengths of more than a few bytes should be set to the null value before the CALL statement is executed.
- ***Nesting CALL statements:*** Procedures can be called from routines as well as application programs. When a procedure is called from a routine, the call is considered to be nested.

  If a procedure returns any query result sets, the result sets are returned as follows:

  – RETURN TO CALLER result sets are visible only to the program that is at the previous nesting level.
  – RETURN TO CLIENT results sets are visible only if the procedure was invoked from a set of nested procedures. If a function or method occurs anywhere in the call chain, the result set is not visible. If the result set is visible, it is only visible to the client application that made the initial procedure call.

  Consider the following example:

```
Client program:
EXEC SQL CALL PROCA;

  PROCA:
  EXEC SQL CALL PROCB;

    PROCB:
    EXEC SQL DECLARE B1 CURSOR WITH RETURN TO CLIENT ...;
    EXEC SQL DECLARE B2 CURSOR WITH RETURN TO CALLER ...;
    EXEC SQL DECLARE B3 CURSOR FOR SELECT UDFA FROM T1;

      UDFA:
      EXEC SQL CALL PROCC;

        PROCC:
        EXEC SQL DECLARE C1 CURSOR WITH RETURN TO CLIENT ...;
        EXEC SQL DECLARE C2 CURSOR WITH RETURN TO CALLER ...;
```

  From procedure PROCB:

  – Cursor B1 is visible in the client application, but not visible in procedure PROCA.
  – Cursor B2 is visible in PROCA, but not visible to the client.

  From procedure PROCC:

  – Cursor C1 is visible to neither UDFA nor to the client application. (Because UDFA appears in the call chain between the client and PROCC, the result set is not returned to the client.)
  – Cursor C2 is visible in UDFA, but not visible to any of the higher procedures.
- ***Nesting procedures within triggers, compound statements, functions, or methods:*** When a procedure is called within a trigger, compound statement, function, or method:

  – The procedure must not issue a COMMIT or a ROLLBACK statement.
  – Result sets returned from the procedure cannot be accessed.
  – If the procedure is defined as READS SQL DATA or MODIFIES SQL DATA, no statement in the procedure can access a table that is being modified by the statement that invoked the procedure (SQLSTATE 57053). If the procedure is defined as MODIFIES SQL DATA, no statement in the procedure can modify a table that is being read or modified by the statement that invoked the procedure (SQLSTATE 57053).

    When a procedure is called within a function or method:

    - The procedure has the same table access restrictions as the invoking function or method.
    - Savepoints defined before the function or method was invoked will not be visible to the procedure, and savepoints defined inside the procedure will not be visible outside the function or method.
    - RETURN TO CLIENT result sets returned from the procedure cannot be accessed from the client.

- ***Compilation of CALL statements from Db2 for IBM i and Db2 for z/OS:*** The compilation of CALL statements from Db2 for IBM i and Db2 for z/OS implicitly behave as if CALL_RESOLUTION DEFERRED was specified. When CALL statements are compiled with CALL_RESOLUTION DEFERRED, all arguments must be provided via host variables, and expressions are not allowed.
- ***Syntax alternatives:*** There is an older form of the CALL statement that can be embedded in an application by precompiling the application with the CALL_RESOLUTION DEFERRED option. This option is not available for SQL procedures and federated procedures.

Examples

- *Example 1:* A Java procedure is defined in the database using the following statement:

```
CREATE PROCEDURE PARTS_ON_HAND (IN PARTNUM INTEGER,
                                OUT COST DECIMAL(7,2),
                                OUT QUANTITY INTEGER)
               EXTERNAL NAME 'parts!onhand'
               LANGUAGE JAVA
               PARAMETER STYLE DB2GENERAL;
```

  A Java application calls this procedure using the following code fragment:

```
...
CallableStatement stpCall;

String sql = "CALL PARTS_ON_HAND (?, ?, ?)";

stpCall = con.prepareCall(sql); /*con is the connection */

stpCall.setInt(1, hvPartnum);
stpCall.setBigDecimal(2, hvCost);
stpCall.setInt(3, hvQuantity);

stpCall.registerOutParameter(2, Types.DECIMAL, 2);
stpCall.registerOutParameter(3, Types.INTEGER);

stpCall.execute();

hvCost = stpCall.getBigDecimal(2);
hvQuantity = stpCall.getInt(3);
...
```

  This application code fragment will invoke the Java method onhand in class `parts`, because the procedure name specified on the CALL statement is found in the database and has the external name `parts!onhand`.

- *Example 2:* There are six FOO procedures, in four different schemas, registered as follows (note that not all required keywords appear):

```
CREATE PROCEDURE AUGUSTUS.FOO (INT) SPECIFIC FOO_1 ...
CREATE PROCEDURE AUGUSTUS.FOO (DOUBLE, DECIMAL(15, 3)) SPECIFIC FOO_2 ...
CREATE PROCEDURE JULIUS.FOO (INT) SPECIFIC FOO_3 ...
CREATE PROCEDURE JULIUS.FOO (INT, INT, INT) SPECIFIC FOO_4 ...
CREATE PROCEDURE CAESAR.FOO (INT, INT) SPECIFIC FOO_5 ...
CREATE PROCEDURE NERO.FOO (INT,INT) SPECIFIC FOO_6 ...
```

  The procedure reference is as follows (where I1 and I2 are INTEGER values):

```
CALL FOO(I1, I2)
```

  Assume that the application making this reference has an SQL path established as:

```
"JULIUS", "AUGUSTUS", "CAESAR"
```

  Following through the algorithm...

  The procedure with specific name FOO_6 is eliminated as a candidate, because the schema "NERO" is not included in the SQL path. FOO_1, FOO_3, and FOO_4 are eliminated as candidates, because they have the wrong number of parameters. The remaining candidates are considered in order, as determined by the SQL path. Note that the types of the arguments and parameters are ignored.

The parameters of FOO_5 exactly match the arguments in the CALL, but FOO_2 is chosen because "AUGUSTUS" appears before "CAESAR" in the SQL path.

- *Example 3:* Assume the following procedure exists.

```
CREATE PROCEDURE update_order(
                  IN IN_POID BIGINT,
                  IN IN_CUSTID BIGINT DEFAULT GLOBAL_CUST_ID,
                  IN NEW_STATUS VARCHAR(10) DEFAULT NULL,
                  IN NEW_ORDERDATE DATE DEFAULT NULL,
                  IN NEW_COMMENTS VARCHAR(1000)DEFAULT NULL)...
```

Also assume that the global variable GLOBAL_CUST_ID is set to the value 1002. Call the procedure to change the status of order 5000 for customer 1002 to 'Shipped'. Leave the rest of the order data as it is by allowing the rest of the arguments to default to the null value.

```
CALL update_order (5000, NEW_STATUS => 'Shipped')
```

The customer with ID 1001 has called and indicated that they received their shipment for purchase order 5002 and are satisfied. Update their order.

```
CALL update_order (5002,
          IN_CUSTID => 1001,
          NEW_STATUS => 'Received',
          NEW_COMMENTS => 'Customer satisfied with the order.')
```

- *Example 4:* The following example illustrates procedure resolution, given two procedures named *p1*:

```
CREATE PROCEDURE p1(i1 INT)...
CREATE PROCEDURE p1(i1 INT DEFAULT 0, i2 INT DEFAULT 0)...
CALL p1(i2=>1)
```

The argument names are taken into consideration during the candidate selection process. Therefore, only the second version of *p1* will be considered a candidate. Furthermore, it can be successfully called because *i1* in this version of *p1* is defined with a default, so only specifying *i2* on the call to *p1* is valid.

- *Example 5:* The following example is another illustration of procedure resolution, given two procedures named *p1*:

```
CREATE PROCEDURE p1(i1 INT, i2 INT DEFAULT 0)...
CREATE PROCEDURE p1(i1 INT DEFAULT 0, i2 INT DEFAULT 0, i3 INT DEFAULT 0)...
CALL p1(i2=>1)
```

One of the criteria for a procedure parameter which does not have a corresponding argument in the CALL statement (specified by either position or name) is that the parameter is defined with a default value. Therefore, the first version of *p1* is not considered a candidate.

CASE

The CASE statement selects an execution path based on multiple conditions. This statement should not be confused with the CASE expression, which allows an expression to be selected based on the evaluation of one or more conditions.

Invocation

This statement can be embedded in:

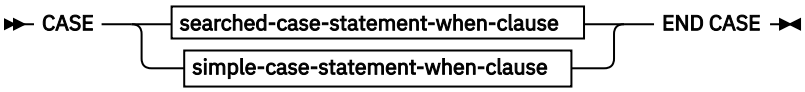
- An SQL procedure definition
- A compound SQL (compiled) statement
- A compound SQL (inlined) statement

The compound SQL statements can be embedded in an SQL procedure definition, SQL function definition, or SQL trigger definition. The CASE statement is not an executable statement and cannot be dynamically prepared.

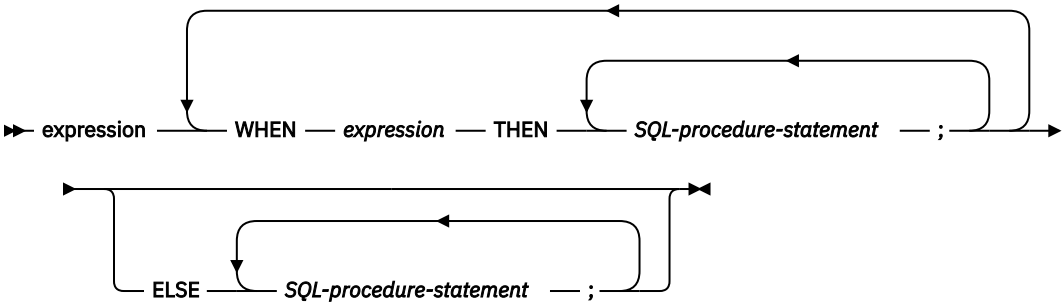
Authorization

No privileges are required to invoke the CASE statement. However, the privileges held by the authorization ID of the statement must include all necessary privileges to invoke the SQL statements and expressions that are embedded in the CASE statement.

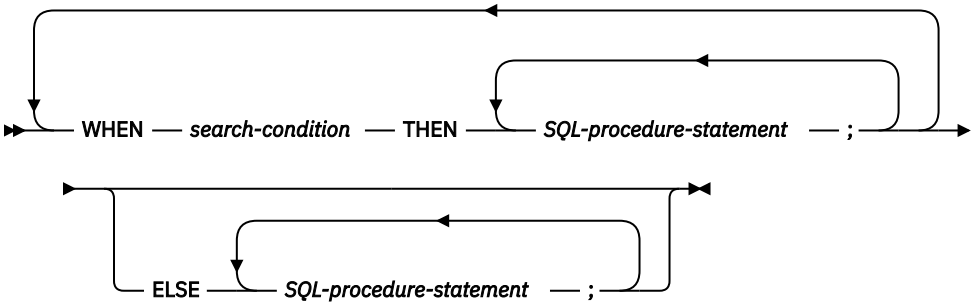
Syntax



simple-case-statement-when-clause



searched-case-statement-when-clause



Description

CASE
Begins a *case-statement*.

simple-case-statement-when-clause
The value of the *expression* before the first WHEN keyword is tested for equality with the value of each *expression* that follows the WHEN keyword. If the search condition is true, the THEN statement is executed. If the result is unknown or false, processing continues to the next search condition. If the result does not match any of the search conditions, and an ELSE clause is present, the statements in the ELSE clause are processed.

searched-case-statement-when-clause
The *search-condition* following the WHEN keyword is evaluated. If it evaluates to true, the statements in the associated THEN clause are processed. If it evaluates to false, or unknown, the next *search-condition* is evaluated. If no *search-condition* evaluates to true and an ELSE clause is present, the statements in the ELSE clause are processed.

SQL-procedure-statement
Specifies a statement that should be invoked. See *SQL-procedure-statement* in "Compound SQL (compiled)" statement.

END CASE
Ends a *case-statement*.

Notes

- If none of the conditions specified in the WHEN are true, and an ELSE clause is not specified, an error is issued at runtime, and the execution of the case statement is terminated (SQLSTATE 20000).
- Ensure that your CASE statement covers all possible execution conditions.

Examples

Depending on the value of SQL variable `v_workdept`, update column DEPTNAME in table DEPARTMENT with the appropriate name.

- *Example 1*: The following example shows how to do this using the syntax for a *simple-case-statement-when-clause*:

```
CASE v_workdept
  WHEN'A00'
    THEN UPDATE department
    SET deptname = 'DATA ACCESS 1';
  WHEN 'B01'
    THEN UPDATE department
    SET deptname = 'DATA ACCESS 2';
  ELSE UPDATE department
    SET deptname = 'DATA ACCESS 3';
END CASE
```

- *Example 2*: The following example shows how to do this using the syntax for a *searched-case-statement-when-clause*:

```
CASE
  WHEN v_workdept = 'A00'
    THEN UPDATE department
    SET deptname = 'DATA ACCESS 1';
  WHEN v_workdept = 'B01'
    THEN UPDATE department
    SET deptname = 'DATA ACCESS 2';
  ELSE UPDATE department
    SET deptname = 'DATA ACCESS 3';
END CASE
```

CLOSE

The CLOSE statement closes a cursor. If a result table was created when the cursor was opened, that table is destroyed.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that cannot be dynamically prepared. When invoked using the command line processor, some options cannot be specified. For more information, refer to "Using command line SQL statements and XQuery statements".

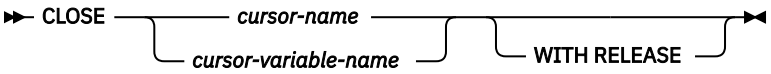
Authorization

If the *cursor-variable-name* references a global variable, then the privileges held by the authorization ID of the statement must include one of the following authorities:

- READ privilege on the global variable that is not defined in a module
- EXECUTE privilege on the module of the global variable that is defined in a module
- EXECUTEIN privilege on the schema containing the module of the global variable that is defined in a module
- DATAACCESS authority on the schema containing the module of the global variable that is defined in a module

For the authorization required to use a cursor, see "DECLARE CURSOR".

Syntax



Description

cursor-name
Identifies the cursor to be closed. The *cursor-name* must identify a declared cursor as explained in the DECLARE CURSOR statement. When the CLOSE statement is executed, the cursor must be in the open state.

cursor-variable-name
Identifies the cursor to be closed. The *cursor-variable-name* must identify a cursor variable. When the CLOSE statement is executed, the underlying cursor of *cursor-variable-name* must be in the open state (SQLSTATE 24501). A CLOSE statement using *cursor-variable-name* can only be used within a compound SQL (compiled) statement.

WITH RELEASE
The release of all locks that have been held for the cursor is attempted. Note that not all of the locks are necessarily released; these locks may be held for other operations or activities.

Notes

- At the end of a unit of work, all cursors that belong to an application process and that were declared without the WITH HOLD option are implicitly closed.
- An underlying cursor of a cursor variable is implicitly closed when it becomes an orphaned cursor. An underlying cursor becomes orphaned when it is no longer an underlying cursor of any cursor variable. For example, this could occur if all the cursor variables for an underlying cursor are in the same scope and all of them go out of scope at the same time.
- The WITH RELEASE clause has no effect when closing cursors defined in functions or methods. The clause also has no effect when closing cursors defined in procedures called from functions or methods.
- The WITH RELEASE clause has no effect for cursors that are operating under isolation levels CS or UR. When specified for cursors that are operating under isolation levels RS or RR, WITH RELEASE terminates some of the guarantees of those isolation levels. Specifically, if the cursor is opened again, an RS cursor may experience the 'nonrepeatable read' phenomenon and an RR cursor may experience either the 'nonrepeatable read' or 'phantom' phenomenon.

  If a cursor that was originally either RR or RS is reopened after being closed using the WITH RELEASE clause, new locks will be acquired.
- Special rules apply to cursors within a procedure that have not been closed before returning to the calling program.
- While a cursor is open (that is, it has not been closed yet), any changes to sequence values as a result of statements involving that cursor (for example, a FETCH or an UPDATE using the cursor that includes a NEXT VALUE expression for a sequence) will not result in an update to PREVIOUS VALUE for those sequences as seen by that cursor. The PREVIOUS VALUE values for these affected sequences are updated when the cursor is closed explicitly with the CLOSE statement. In a partitioned database environment, if a cursor is closed implicitly by a commit or a rollback, the PREVIOUS VALUE may not be updated with the most recently generated value for the sequence.

Example

A cursor is used to fetch one row at a time into the C program variables `dnum`, `dname`, and `mnum`. Finally, the cursor is closed. If the cursor is reopened, it is again located at the beginning of the rows to be fetched.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO
```

```
  FROM TDEPT
  WHERE ADMRDEPT = 'A00';

EXEC SQL OPEN C1;

while (SQLCODE==0) {                                                    .
  EXEC SQL FETCH C1 INTO :dnum, :dname, :mnum;
     .
     .
}
EXEC SQL CLOSE C1;
```

COMMENT

The COMMENT statement adds or replaces comments in the catalog descriptions of various objects.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

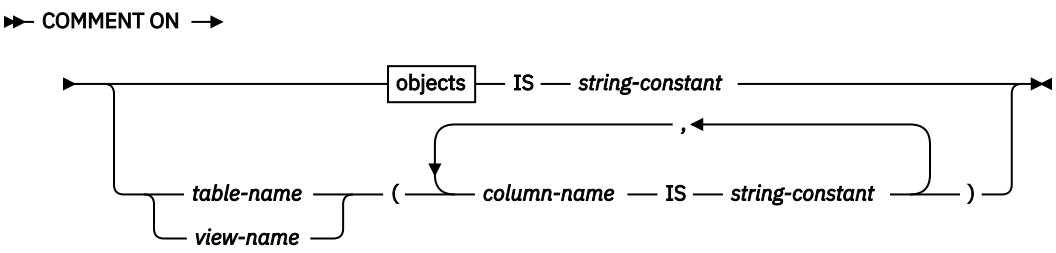
Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

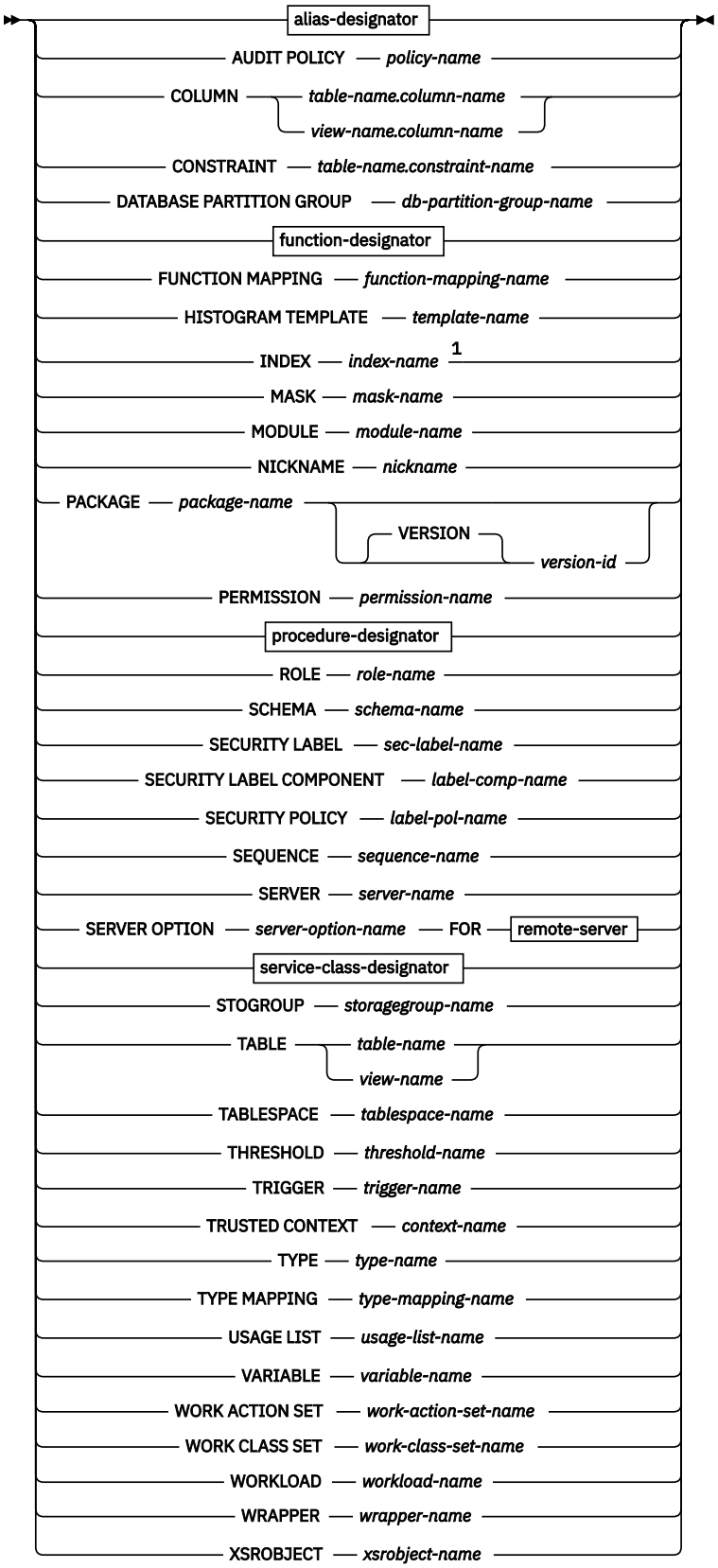
- Owner of the object (underlying table for column or constraint), as recorded in the OWNER column of the catalog view for the object
- ALTERIN privilege on the schema (applicable only to objects that allow more than one-part names)
- CONTROL privilege on the object (applicable only to index, package, table, or view objects)
- ALTER privilege on the object (applicable only to table objects)
- CREATE_SECURE_OBJECT authority (applicable only to secure functions or secure triggers)
- The WITH ADMIN OPTION (applicable only to roles)
- SCHEMAADM authority on the schema (applicable only to objects that allow more than one-part names)
- WLMADM authority (applicable only to workload manager objects)
- SECADM authority (applicable only to audit policy, column mask, role, row permission, secure function, secure trigger, security label, security label component, security policy, or trusted context objects; also applicable to tables for which row level access control or column level access control has been activated)
- DBADM authority (applicable to all objects except audit policy, role, security label, security label component, security policy, or trusted context objects)

Note that for table space, storage group, or database partition group, and bufferpools, the authorization ID must have SYSCTRL or SYSADM authority.

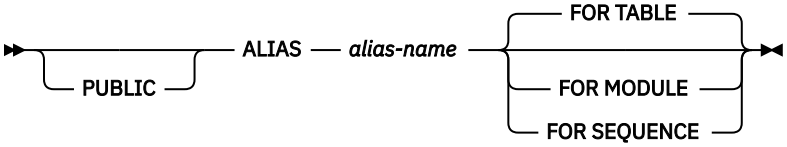
Syntax



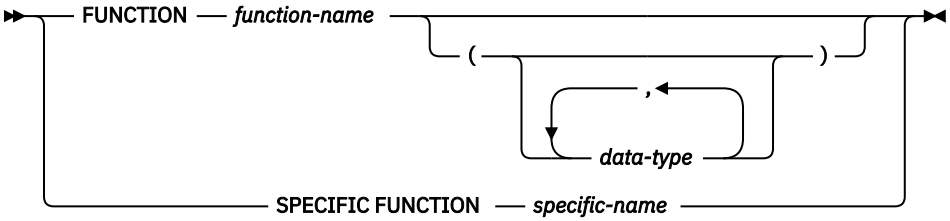
objects



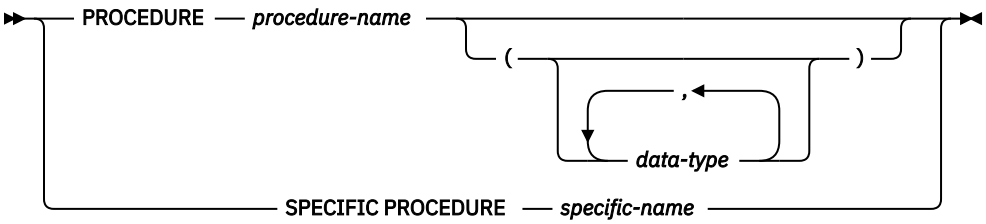
alias-designator



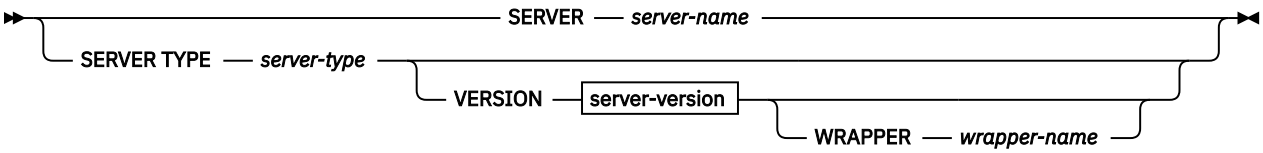
function-designator



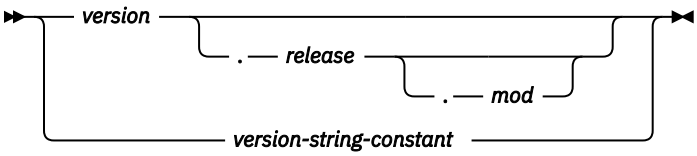
procedure-designator



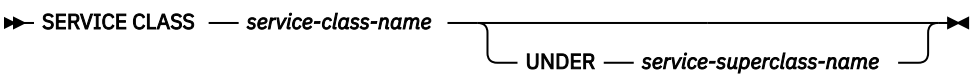
remote-server



server-version



service-class-designator



Notes:

[1] *Index-name* can be the name of either an index or an index specification.

Description

alias-designator

ALIAS *alias-name*

Indicates a comment will be added or replaced for an alias. The *alias-name* must identify an alias that exists at the current server (SQLSTATE 42704).

FOR TABLE, FOR MODULE, or FOR SEQUENCE

Specifies the object type for the alias.

FOR TABLE

The alias is for a table, view, or nickname. The comment replaces the value of the REMARKS column of the SYSCAT.TABLES catalog view for the row that describes the alias.

FOR MODULE
The alias is for a module. The comment replaces the value of the REMARKS column of the SYSCAT.MODULES catalog view for the row that describes the alias.

FOR SEQUENCE
The alias is for a sequence. The comment replaces the value of the REMARKS column of the SYSCAT.SEQUENCES catalog view for the row that describes the alias.

If PUBLIC is specified, the *alias-name* must identify a public alias that exists at the current server (SQLSTATE 42704).

AUDIT POLICY *policy-name*
Indicates a comment will be added or replaced for an audit policy. The *policy-name* must identify an audit policy that exists at the current server (SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.AUDITPOLICIES catalog view for the row that describes the audit policy.

COLUMN *table-name.column-name* or *view-name.column-name*
Indicates that a comment for a column will be added or replaced. The *table-name.column-name* or *view-name.column-name* combination must identify a column and table combination that exists at the current server (SQLSTATE 42704), but must not identify a global temporary table (SQLSTATE 42995). The comment replaces the value of the REMARKS column of the SYSCAT.COLUMNS catalog view for the row that describes the column.

CONSTRAINT *table-name.constraint-name*
Indicates a comment will be added or replaced for a constraint. The *table-name.constraint-name* combination must identify a constraint and the table that it constrains; they must exist at the current server (SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.TABCONST catalog view for the row that describes the constraint.

DATABASE PARTITION GROUP *db-partition-group-name*
Indicates a comment will be added or replaced for a database partition group. The *db-partition-group-name* must identify a distinct database partition group that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.DBPARTITIONGROUPS catalog view for the row that describes the database partition group.

function-designator
Indicates a comment will be added or replaced for a function. For more information, see "Function, method, and procedure designators" on page 745.

It is not possible to comment on a function that is in the SYSIBM, SYSIBMADM, SYSFUN, or SYSPROC schema (SQLSTATE 42832).

The comment replaces the value of the REMARKS column of the SYSCAT.ROUTINES catalog view for the row that describes the function.

FUNCTION MAPPING *function-mapping-name*
Indicates a comment will be added or replaced for a function mapping. The *function-mapping-name* must identify a function mapping that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.FUNCMAPPINGS catalog view for the row that describes the function mapping.

HISTOGRAM TEMPLATE *template-name*
Indicates a comment will be added or replaced for a histogram template. The *template-name* must identify a histogram template that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.HISTOGRAMTEMPLATES catalog view for the row that describes the histogram template.

INDEX *index-name*
Indicates a comment will be added or replaced for an index or index specification. The *index-name* must identify either a distinct index or an index specification that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.INDEXES catalog view for the row that describes the index or index specification.

MASK *mask-name*
Identifies the column mask to which the comment applies. *mask-name* must identify a column mask that exists at the current server (SQLSTATE 42704). The comment is placed in the REMARKS column of the SYSCAT.CONTROLS catalog table for the row that describes the mask.

MODULE *module-name*
Indicates a comment will be added or replaced for a module. The *module-name* must identify a module that exists at the current server (SQLSTATE 42704). The specified name must not be an alias for a module (SQLSTATE 560CT). The comment replaces the value for the REMARKS column of the SYSCAT.MODULES catalog view for the row that describes the module.

NICKNAME *nickname*
Indicates a comment will be added or replaced for a nickname. The *nickname* must be a nickname that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.TABLES catalog view for the row that describes the nickname.

PACKAGE *package-name*
Indicates that a comment will be added or replaced for a package. The package name must identify a package that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.PACKAGES catalog view for the row that describes the package.

VERSION *version-id*
Identifies which package version is to be commented on. If a value is not specified, the version defaults to the empty string. If multiple packages with the same package name but different versions exist, only one package version can be commented on in one invocation of the COMMENT statement. Delimit the version identifier with double quotation marks when it:

- Is generated by the VERSION(AUTO) precompiler option
- Begins with a digit
- Contains lowercase or mixed-case letters

If the statement is invoked from an operating system command prompt, precede each double quotation mark delimiter with a back slash character to ensure that the operating system does not strip the delimiters.

PERMISSION *permission-name*
Identifies the row permission to which the comment applies. *permission-name* must identify a row permission that exists at the current server (SQLSTATE 42704, SQLCODE -204). The comment is placed in the REMARKS column of the SYSCAT.CONTROLS catalog table for the row that describes the permission.

procedure-designator
Indicates a comment will be added or replaced for a procedure. For more information, see "Function, method, and procedure designators" on page 745.

It is not possible to comment on a procedure that is in the SYSIBM, SYSIBMADM, SYSFUN, or SYSPROC schema (SQLSTATE 42832).

The comment replaces the value of the REMARKS column of the SYSCAT.ROUTINES catalog view for the row that describes the procedure.

ROLE *role-name*
Indicates a comment will be added or replaced for a role. The *role-name* must identify a role that exists at the current server (SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.ROLES catalog view for the row that describes the role.

SCHEMA *schema-name*
Indicates a comment will be added or replaced for a schema. The *schema-name* must identify a schema that exists at the current server (SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.SCHEMATA catalog view for the row that describes the schema.

SECURITY LABEL *sec-label-name*
Indicates that a comment will be added or replaced for the security label named *sec-label-name*. The name must be qualified with a security policy and must identify a security label that exists at the

current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.SECURITYLABELS catalog view for the row that describes the security label.

SECURITY LABEL COMPONENT *label-comp-name*
Indicates that a comment will be added or replaced for the security label component named *label-comp-name*. The *label-comp-name* must identify a security label component that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.SECURITYLABELCOMPONENTS catalog view for the row that describes the security label component.

SECURITY POLICY *label-pol-name*
Indicates that a comment will be added or replaced for the security policy named *label-pol-name*. The *label-pol-name* must identify a security policy that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.SECURITYPOLICIES catalog view for the row that describes the security policy.

SEQUENCE *sequence-name*
Indicates a comment will be added or replaced for a sequence. The *sequence-name* must identify a sequence that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.SEQUENCES catalog view for the row that describes the sequence.

SERVER *server-name*
Indicates a comment will be added or replaced for a data source. The *server-name* must identify a data source that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.SERVERS catalog view for the row that describes the data source.

SERVER OPTION *server-option-name* FOR *remote-server*
Indicates a comment will be added or replaced for a server option.

server-option-name
Identifies a server option. This option must be one that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.SERVEROPTIONS catalog view for the row that describes the server option.

remote-server
Describes the data source to which the *server-option* applies.

SERVER *server-name*
Names the data source to which the *server-option* applies. The *server-name* must identify a data source that exists at the current server.

TYPE *server-type*
Specifies the type of data source (such as Db2 for z/OS or Oracle) to which the *server-option* applies. The *server-type* can be specified in either lower- or uppercase; it will be stored in uppercase in the catalog.

VERSION
Specifies the version of the data source identified by *server-name*.

version
Specifies the version number. *version* must be an integer.

release
Specifies the number of the release of the version denoted by *version*. *release* must be an integer.

mod
Specifies the number of the modification of the release denoted by *release*. *mod* must be an integer.

version-string-constant
Specifies the complete designation of the version. The *version-string-constant* can be a single value (for example, '8i'); or it can be the concatenated values of *version*, *release*, and, if applicable, *mod* (for example, '8.0.3').

WRAPPER *wrapper-name*
Identifies the wrapper that is used to access the data source referenced by *server-name.*

service-class-designator

SERVICE CLASS *service-class-name*
Indicates a comment will be added or replaced for a service class. The *service-class-name* must identify a service class that exists at the current server (SQLSTATE 42704). To add or replace a comment for a service subclass, the *service-superclass-name* must be specified using the UNDER clause. The comment replaces the value for the REMARKS column of the SYSCAT.SERVICECLASSES catalog view for the row that describes the service class.

UNDER *service-superclass-name*
Specifies the service superclass of the service subclass when adding or replacing a comment for a service subclass. The *service-superclass-name* must identify a service superclass that exists at the current server (SQLSTATE 42704).

STOGROUP *storagegroup-name*
Indicates a comment will be added or replaced for a storage group. The *storagegroup-name* must identify a distinct storage group that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.STOGROUPS catalog view for the row that describes the storage group.

TABLE *table-name* or *view-name*
Indicates a comment will be added or replaced for a table or view. The *table-name* or *view-name* must identify a table or view (not an alias or nickname) that exists at the current server (SQLSTATE 42704) and must not identify a declared temporary table (SQLSTATE 42995). The comment replaces the value for the REMARKS column of the SYSCAT.TABLES catalog view for the row that describes the table or view.

TABLESPACE *tablespace-name*
Indicates a comment will be added or replaced for a table space. The *tablespace-name* must identify a distinct table space that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.TABLESPACES catalog view for the row that describes the table space.

THRESHOLD *threshold-name*
Indicates a comment will be added or replaced for a threshold. The *threshold-name* must identify a threshold that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.THRESHOLDS catalog view for the row that describes the threshold.

TRIGGER *trigger-name*
Indicates a comment will be added or replaced for a trigger. The *trigger-name* must identify a distinct trigger that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.TRIGGERS catalog view for the row that describes the trigger.

TRUSTED CONTEXT *context-name*
Indicates a comment will be added or replaced for a trusted context. The *context-name* must identify a trusted context that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.CONTEXTS catalog view for the row that describes the trusted context.

TYPE *type-name*
Indicates a comment will be added or replaced for a user-defined type. The *type-name* must identify a user-defined type that exists at the current server (SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.DATATYPES catalog view for the row that describes the user-defined type.

In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

TYPE MAPPING *type-mapping-name*
Indicates a comment will be added or replaced for a user-defined data type mapping. The *type-mapping-name* must identify a data type mapping that exists at the current server (SQLSTATE 42704).

The comment replaces the value for the REMARKS column of the SYSCAT.TYPEMAPPINGS catalog view for the row that describes the mapping.

USAGE LIST *usage-list-name*
Indicates a comment will be added or replaced for a usage list. The *usage-list-name* must identify a usage list that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.USAGELISTS catalog view for the row that describes the usage list.

VARIABLE *variable-name*
Indicates a comment will be added or replaced for a global variable. The *variable-name* must identify a global variable that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.VARIABLES catalog view for the row that describes the variable.

WORK ACTION SET *work-action-set-name*
Indicates a comment will be added or replaced for a work action set. The *work-action-set-name* must identify a work action set that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.WORKACTIONSETS catalog view for the row that describes the work action set.

WORK CLASS SET *work-class-set-name*
Indicates a comment will be added or replaced for a work class set. The *work-class-set-name* must identify a work class set that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.WORKCLASSSETS catalog view for the row that describes the work class set.

WORKLOAD *workload-name*
Indicates that a comment will be added or replaced for a workload. The *workload-name* must identify a workload that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.WORKLOADS catalog view for the row that describes the workload.

WRAPPER *wrapper-name*
Indicates a comment will be added or replaced for a wrapper. The *wrapper-name* must identify a wrapper that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.WRAPPERS catalog view for the row that describes the wrapper.

XSROBJECT *xsrobject-name*
Indicates a comment will be added or replaced for an XSR object. The *xsrobject-name* must identify an XSR object that exists at the current server (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.XSROBJECTS catalog view for the row that describes the XSR object.

IS *string-constant*
Specifies the comment to be added or replaced. The *string-constant* can be any character string constant of up to 254 bytes. (Carriage return and line feed each count as 1 byte.)

***table-name*|*view-name* ({ *column-name* IS *string-constant* } ...)**
This form of the COMMENT statement provides the ability to specify comments for multiple columns of a table or view. The column names must not be qualified, each name must identify a column of the specified table or view, and the table or view must exist at the current server. The *table-name* cannot be a declared temporary table (SQLSTATE 42995).

A comment cannot be made on a column of an inoperative view (SQLSTATE 51024).

Notes

- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - NODEGROUP can be specified in place of DATABASE PARTITION GROUP
  - DISTINCT TYPE *type-name* can be specified in place of TYPE *type-name*
  - DATA TYPE *type-name* can be specified in place of TYPE *type-name*

– SYNONYM can be specified in place of ALIAS

Examples

- *Example 1:* Add a comment for the EMPLOYEE table.

```
COMMENT ON TABLE EMPLOYEE
  IS 'Reflects first quarter reorganization'
```

- *Example 2:* Add a comment for the EMP_VIEW1 view.

```
COMMENT ON TABLE EMP_VIEW1
  IS 'View of the EMPLOYEE table without salary information'
```

- *Example 3:* Add a comment for the EDLEVEL column of the EMPLOYEE table.

```
COMMENT ON COLUMN EMPLOYEE.EDLEVEL
  IS 'highest grade level passed in school'
```

- *Example 4:* Add comments for two different columns of the EMPLOYEE table.

```
COMMENT ON EMPLOYEE
  (WORKDEPT IS 'see DEPARTMENT table for names',
  EDLEVEL IS 'highest grade level passed in school' )
```

- *Example 5:* Pellow wants to comment on the CENTER function, which he created in his PELLOW schema, using the signature to identify the specific function to be commented on.

```
COMMENT ON FUNCTION CENTER (INT,FLOAT)
  IS 'Frank''s CENTER fctn, uses Chebychev method'
```

- *Example 6:* McBride wants to comment on another CENTER function, which she created in the PELLOW schema, using the specific name to identify the function instance to be commented on:

```
COMMENT ON SPECIFIC FUNCTION PELLOW.FOCUS92 IS
  'Louise''s most triumphant CENTER function, uses the
       Brownian fuzzy-focus technique'
```

- *Example 7:* Comment on the function ATOMIC_WEIGHT in the CHEM schema, where it is known that there is only one function with that name:

```
COMMENT ON FUNCTION CHEM.ATOMIC_WEIGHT
  IS 'takes atomic nbr, gives atomic weight'
```

- *Example 8:* Eigler wants to comment on the SEARCH procedure, which he created in his EIGLER schema, using the signature to identify the specific procedure to be commented on.

```
COMMENT ON PROCEDURE SEARCH (CHAR,INT)
  IS 'Frank''s mass search and replace algorithm'
```

- *Example 9:* Macdonald wants to comment on another SEARCH function, which he created in the EIGLER schema, using the specific name to identify the procedure instance to be commented on:

```
COMMENT ON SPECIFIC PROCEDURE EIGLER.DESTROY IS
  'Patrick''s mass search and destroy algorithm'
```

- *Example 10:* Comment on the procedure OSMOSIS in the BIOLOGY schema, where it is known that there is only one procedure with that name:

```
COMMENT ON PROCEDURE BIOLOGY.OSMOSIS
  IS 'Calculations modelling osmosis'
```

- *Example 11:* Comment on an index specification named INDEXSPEC.

```
COMMENT ON INDEX INDEXSPEC
  IS 'An index specification that indicates to the optimizer
  that the table referenced by nickname NICK1 has an index.'
```

- *Example 12:* Comment on the wrapper whose default name is NET8.

```
COMMENT ON WRAPPER NET8
  IS 'The wrapper for data sources associated with
      Oracle's Net8 client software.'
```

- *Example 13:* Create a comment on the XML schema HR.EMPLOYEE.

```
COMMENT ON XSROBJECT HR.EMPLOYEE
  IS 'This is the base XML Schema for employee data.'
```

- *Example 14:* Create a comment for trusted context APPSERVER.

```
COMMENT ON TRUSTED CONTEXT APPSERVER
  IS 'WebSphere Server'
```

- *Example 15:* Create a comment for column mask M1.

```
COMMENT ON MASK M1 IS 'Column mask for column EMP.SALARY'
```

COMMIT

The COMMIT statement terminates a unit of work and commits the database changes that were made by that unit of work.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax

Description

The unit of work in which the COMMIT statement is executed is terminated and a new unit of work is initiated. All changes made by the following statements executed during the unit of work are committed: ALTER, COMMENT, CREATE, DROP, GRANT, LOCK TABLE, REVOKE, SET INTEGRITY, and the data change statements (INSERT, DELETE, MERGE, UPDATE), including those nested in a query.

The following statements, however, are not under transaction control and changes made by them are independent of the COMMIT statement:

- SET CONNECTION
- SET ENCRYPTION PASSWORD
- SET EVENT MONITOR STATE
- SET PASSTHRU

  Note: Although the SET PASSTHRU statement is not under transaction control, the passthru session initiated by the statement is under transaction control.
- SET SERVER OPTION
- SET variable
- Assignments to updatable special registers

All locks acquired by the unit of work subsequent to its initiation are released, except necessary locks for open cursors that are declared WITH HOLD. All open cursors not defined WITH HOLD are closed. Open cursors defined WITH HOLD remain open, and the cursor is positioned before the next logical row of the result table. (A FETCH must be performed before a positioned UPDATE or DELETE statement is issued.) All LOB locators are freed. Note that this is true even when the locators are associated with LOB values retrieved via a cursor that has the WITH HOLD property.

Dynamic SQL statements prepared in a package bound with the KEEPDYNAMIC YES option are kept in the SQL context after a COMMIT statement. This is the default behavior. The statement might be implicitly prepared again, as a result of DDL operations that are rolled back within the unit of work. Inactive dynamic SQL statements prepared in a package bound with KEEPDYNAMIC NO are removed from the SQL context after a COMMIT. The statement must be prepared again before it can be executed in a new transaction.

All savepoints set within the transaction are released.

The following statements behave differently than other data definition language (DDL) and data control language (DCL) statements. Changes made by these statements do not take effect until the statement is committed, even for the current connection that issues the statement. Only one of these statements can be issued by any application at a time, and only one of these statements is allowed within any one unit of work. Each statement must be followed by a COMMIT or a ROLLBACK statement before another one of these statements can be issued.

- CREATE SERVICE CLASS, ALTER SERVICE CLASS, or DROP (SERVICE CLASS)
- CREATE THRESHOLD, ALTER THRESHOLD, or DROP (THRESHOLD)
- CREATE WORK ACTION, ALTER WORK ACTION, or DROP (WORK ACTION)
- CREATE WORK CLASS, ALTER WORK CLASS, or DROP (WORK CLASS)
- CREATE WORKLOAD, ALTER WORKLOAD, or DROP (WORKLOAD)
- GRANT (Workload Privileges) or REVOKE (Workload Privileges)

Notes

- It is strongly recommended that each application process explicitly ends its unit of work before terminating. If the application program ends normally without a COMMIT or ROLLBACK statement then the database manager attempts a commit or rollback depending on the application environment.
- For information about the impact of COMMIT on cached dynamic SQL statements, see "EXECUTE".
- For information about potential impacts of COMMIT on created temporary tables, see "CREATE GLOBAL TEMPORARY TABLE".
- For information about potential impacts of COMMIT on declared temporary tables, see "DECLARE GLOBAL TEMPORARY TABLE".
- The following dynamic SQL statements may be active during COMMIT:
  - Open WITH HOLD cursor
  - COMMIT statement
  - CALL statements under which the COMMIT statement was executed

Example

Commit alterations to the database made since the last commit point.

```
COMMIT WORK
```

Compound SQL

A compound SQL statement is a sequence of individual SQL statements enclosed by BEGIN and END keywords.

There are three types of compound SQL statements:

- Inlined: A compound SQL (inlined) statement is a compound SQL statement that is inlined at run time within another SQL statement. Compound SQL (inlined) statements have the property of being atomically executed; if the execution of any of the statements raises an error, the full statement is rolled back.
- Embedded: Combines one or more other SQL statements (*sub-statements*) into an executable block.
- Compiled: A compound SQL (compiled) statement can contain SQL control statements and SQL statements. Compound SQL (compiled) statements can be used to implement procedural logic through a sequence of SQL statements with a local scope for variables, conditions, cursors, and handlers.

Compound SQL (inlined)

A compound SQL (inlined) statement is a compound SQL statement that is inlined at run time within another SQL statement. Compound SQL (inlined) statements have the property of being atomically executed; if the execution of any of the statements raises an error, the full statement is rolled back.

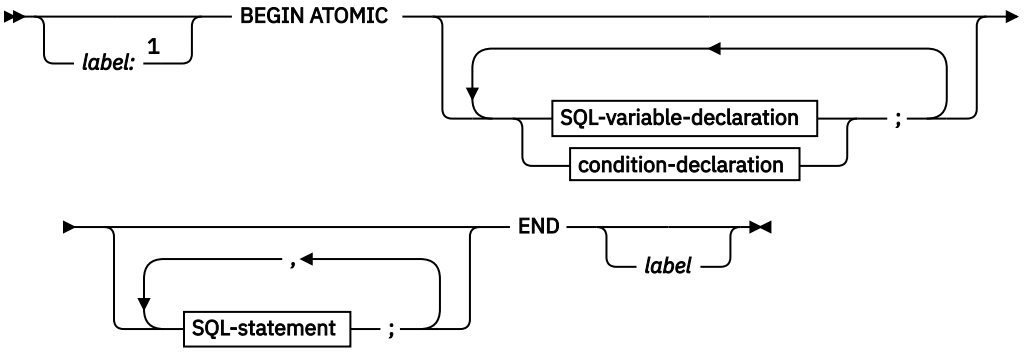
Invocation

This statement can be embedded in a trigger, SQL function, or SQL method, or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

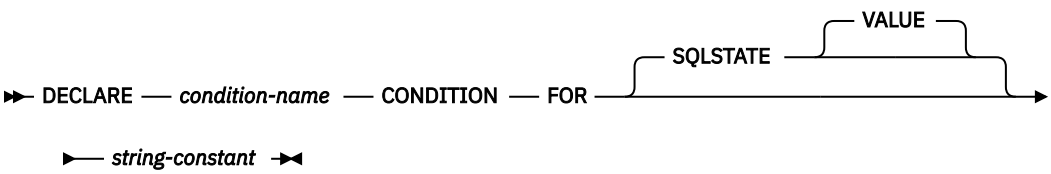
The privileges held by the authorization ID of the statement must also include all of the privileges necessary to invoke the SQL statements that are specified in the compound statement.

Syntax

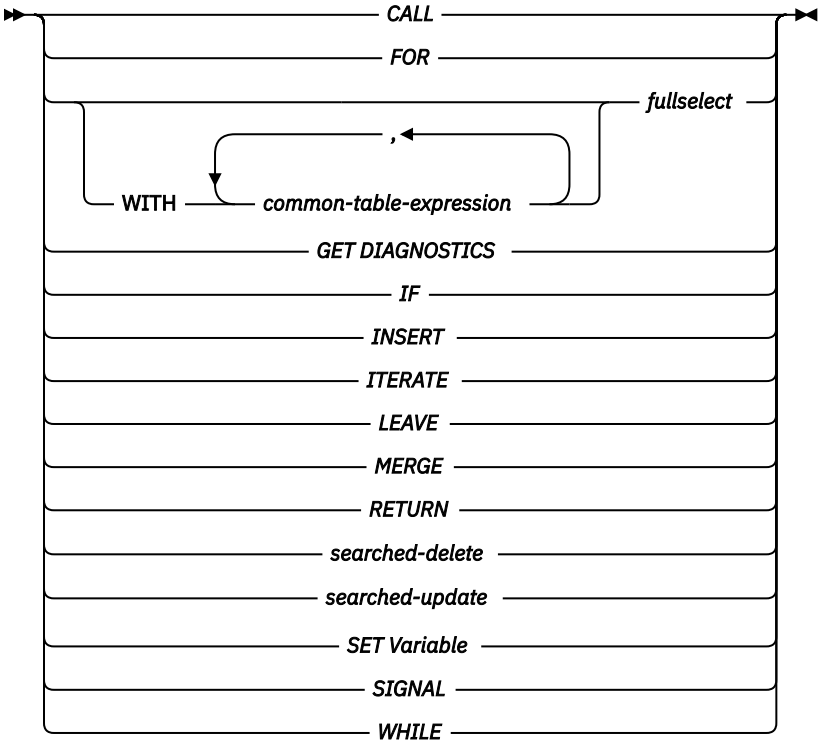


SQL-variable-declaration

condition-declaration



SQL-statement



Notes:

[1] A label can only be specified when the statement is in a function, method, or trigger definition.

Description

label
Defines the label for the code block. If the beginning label is specified, it can be used to qualify SQL variables declared in the compound SQL (inlined) statement and can also be specified on a LEAVE statement. If the ending label is specified, it must be the same as the beginning label.

ATOMIC
ATOMIC indicates that, if an error occurs in the compound statement, all SQL statements in the compound statement will be rolled back, and any remaining SQL statements in the compound statement are not processed.

If the ATOMIC keyword is specified in an SQL function in a module or an SQL procedure, the compound statement is processed as a compound SQL (compiled) statement.

SQL-statement
Specifies an SQL statement to be executed within the compound SQL (inlined) statement.

SQL-variable-declaration
Declares a variable that is local to the compound SQL (inlined) statement.

SQL-variable-name
Defines the name of a local variable. SQL variable names are converted to uppercase. The name cannot be the same as:

- Another SQL variable within the compound statement
- A parameter name

If an SQL statement contains an identifier with the same name as an SQL variable and a column reference, the identifier is interpreted as a column.

data-type
Specifies the data type of the variable. The XML data type is not supported in a compound SQL (inlined) statement used in a trigger, in a method, or as a stand-alone statement (SQLSTATE 429BB). The XML data type is supported when the compound SQL (inlined) statement is used in an SQL function body.

DEFAULT
Defines the default for the SQL variable. The variable is initialized when the compound SQL (inlined) statement is executed. The default value must be assignment-compatible with the data type of the variable. If a default value is not specified, the default for the SQL variable is initialized to the null value.

NULL
Specifies NULL as the default for the SQL variable.

constant
Specifies a constant as the default for the SQL variable.

condition-declaration
Declares a condition name and corresponding SQLSTATE value.

condition-name
Specifies the name of the condition. The condition name must be unique within the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement (SQLSTATE 42734). A condition name can only be referenced within the compound statement in which it is declared, including any compound statements that are nested within that compound statement (SQLSTATE 42737).

FOR SQLSTATE *string-constant*
Specifies the SQLSTATE associated with the condition. The *string-constant* must be specified as five characters enclosed by single quotation marks, and the SQLSTATE class (the first two characters) must not be '00'.

Notes

- Compound SQL (inlined) statements are compiled as one single statement. This statement is effective for short scripts involving little control flow logic but significant data flow. For larger constructs with requirements for nested control flow or condition handling, a better choice is to use the compound SQL (compiled) statement or an SQL procedure.
- A procedure called within a compound statement must not issue a COMMIT or a ROLLBACK statement (SQLSTATE 42985).
- ***Table access restrictions:*** If a procedure is defined as READS SQL DATA or MODIFIES SQL DATA, no statement in the procedure can access a table that is being modified by the compound statement that invoked the procedure (SQLSTATE 57053). If the procedure is defined as MODIFIES SQL DATA, no statement in the procedure can modify a table that is being read or modified by the compound statement that invoked the procedure (SQLSTATE 57053).
- ***XML assignments:*** Assignment to parameters and variables of data type XML is done by reference in SQL function bodies.

  When XML values are passed by reference, any input node trees are used directly. This direct usage preserves all properties, including document order, the original node identities, and all parent properties.
- ***Isolation level:*** If a *select-statement*, *fullselect*, or *subselect* specifies an *isolation-clause*, the clause is ignored and a warning is returned.

Example

This example illustrates how inline SQL PL can be used in a data warehousing scenario for data cleansing.

The example introduces three tables. The TARGET table contains the cleansed data. The EXCEPT table stores rows that cannot be cleansed (exceptions) and the SOURCE table contains the raw data to be cleansed.

A simple SQL function called DISCRETIZE is used to classify and modify the data. It returns the null value for all bad data. The compound SQL (inlined) statement then cleanses the data. It walks all rows of the SOURCE table in a FOR-loop and decides whether the current row gets inserted into the TARGET or the EXCEPT table, depending on the result of the DISCRETIZE function. More elaborate mechanisms (multistage cleansing) are possible with this technique.

The same code can be written using an SQL Procedure or any other procedure or application in a host language. However, the compound SQL (inlined) statement offers a unique advantage in that the FOR-loop does not open a cursor and the single row inserts are not really single row inserts. In fact, the logic is effectively a multi-table insert from a shared select.

This is achieved by compilation of the compound SQL (inlined) statement as a single statement. Similar to a view whose body is integrated into the query that uses it and then is compiled and optimized as a whole within the query context, the database optimizer compiles and optimizes both the control and data flow together. The whole logic is therefore executed within the runtime environment of the database. No data is moved outside of the core database engine, as would be done for a procedure.

The first step is to create the required tables:

```
CREATE TABLE TARGET
  (PK INTEGER NOT NULL
  PRIMARY KEY, C1 INTEGER)
```

This creates a table called TARGET to contain the cleansed data.

```
CREATE TABLE EXCEPT
  (PK INTEGER NOT NULL
  PRIMARY KEY, C1 INTEGER)
```

This creates a table called EXCEPT to contain the exceptions.

```
CREATE TABLE SOURCE
  (PK INTEGER NOT NULL
  PRIMARY KEY, C1 INTEGER)
```

This creates a table called SOURCE to hold the data that is to be cleansed.

Next, a function named DISCRETIZE is created to cleanse the data by throwing out all values outside [0..1000] and aligning them to steps of 10.

```
CREATE FUNCTION DISCRETIZE(RAW INTEGER) RETURNS INTEGER
  RETURN CASE
    WHEN RAW < 0 THEN CAST(NULL AS INTEGER)
    WHEN RAW > 1000 THEN NULL
    ELSE ((RAW / 10) * 10) + 5
  END
```

Then the values are inserted:

```
INSERT INTO SOURCE (PK, C1)
  VALUES (1,   -5),
         (2, NULL),
         (3, 1200),
         (4,   23),
         (5,   10),
         (6,  876)
```

Invoke the function:

```
  BEGIN ATOMIC
    FOR ROW AS
      SELECT PK, C1, DISCRETIZE(C1) AS D FROM SOURCE
    DO
      IF ROW.D IS NULL THEN
```

```
        INSERT INTO EXCEPT VALUES(ROW.PK, ROW.C1);
      ELSE
        INSERT INTO TARGET VALUES(ROW.PK, ROW.D);
      END IF;
    END FOR;
  END
```

And test the results:

```
SELECT * FROM EXCEPT ORDER BY 1
PK          C1
----------- -----------
          1          -5
          2           -
          3        1200
  3 record(s) selected.

SELECT * FROM TARGET ORDER BY 1
PK          C1
----------- -----------
          4          25
          5          15
          6         875
  3 record(s) selected.
```

The final step is to clean up:

```
DROP FUNCTION DISCRETIZE
DROP TABLE SOURCE
DROP TABLE TARGET
DROP TABLE EXCEPT
```

Compound SQL (embedded)

Combines one or more other SQL statements (*sub-statements*) into an executable block.

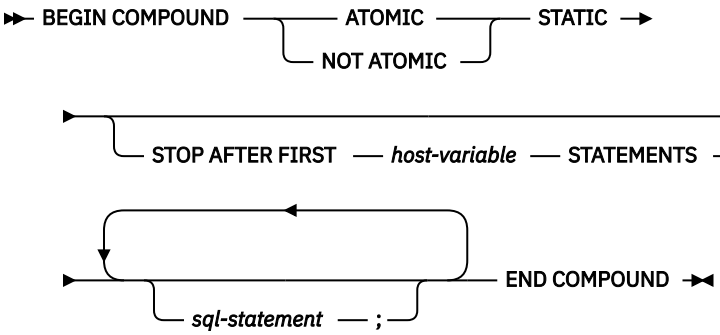
Invocation

This statement can only be embedded in an application program. The entire compound SQL (embedded) statement construct is an executable statement that cannot be dynamically prepared. The statement is not supported in REXX.

Authorization

No privileges are required to invoke an compound SQL (embedded). However, the privileges held by the authorization ID of the statement must include all necessary privileges to invoke the SQL statements that are embedded in the compound statement.

Syntax



Description

ATOMIC
Specifies that, if any of the sub-statements within the compound SQL (embedded) statement fail, then all changes made to the database by any of the sub-statements, including changes made by successful sub-statements, are undone.

NOT ATOMIC
Specifies that, regardless of the failure of any sub-statements, the compound SQL (embedded) statement will not undo any changes made to the database by the other sub-statements.

STATIC
Specifies that input variables for all sub-statements retain their original value. For example, if

```
SELECT ... INTO :abc ...
```

is followed by:

```
UPDATE T1 SET C1 = 5 WHERE C2 = :abc
```

the UPDATE statement will use the value that :abc had at the start of the execution of the compound SQL (embedded) statement, not the value that follows the SELECT INTO.

If the same variable is set by more than one sub-statement, the value of that variable following the compound SQL (embedded) statement is the value set by the last sub-statement.

Note: Non-static behavior is not supported. This means that the sub-statements should be viewed as executing non-sequentially and sub-statements should not have interdependencies.

STOP AFTER FIRST
Specifies that only a certain number of sub-statements will be executed.

host-variable
A small integer that specifies the number of sub-statements to be executed.

STATEMENTS
Completes the STOP AFTER FIRST *host-variable* clause.

sql-statement
All executable statements except the following can be contained within an embedded static compound SQL (embedded) statement:

```
CALL                FETCH
CLOSE               OPEN
CONNECT             PREPARE
Compound SQL        RELEASE (Connection)
DESCRIBE            ROLLBACK
DISCONNECT          SET CONNECTION
EXECUTE IMMEDIATE   SET variable
```

Note: INSERT, UPDATE, and DELETE are not supported in compound SQL for use with nicknames.

If a COMMIT statement is included, it must be the last sub-statement. If COMMIT is in this position, it will be issued even if the STOP AFTER FIRST *host-variable* STATEMENTS clause indicates that not all of the sub-statements are to executed. For example, suppose COMMIT is the last sub-statement in a compound SQL block of 100 sub-statements. If the STOP AFTER FIRST STATEMENTS clause indicates that only 50 sub-statements are to be executed, then COMMIT will be the 51st sub-statement.

An error will be returned if COMMIT is included when using CONNECT TYPE 2 or running in an XA distributed transaction processing environment (SQLSTATE 25000).

Rules

- Db2 Connect does not support SELECT statements selecting LOB columns in a compound SQL block.
- No host language code is allowed within a compound SQL (embedded) statement; that is, no host language code is allowed between the sub-statements that make up the compound SQL (embedded) statement.

- Only NOT ATOMIC compound SQL (embedded) statements will be accepted by Db2 Connect.
- Compound SQL (embedded) statements cannot be nested.
- A prepared COMMIT statement is not allowed in an ATOMIC compound SQL (embedded) statement

Notes

- One SQLCA is returned for the entire compound SQL (embedded) statement. Most of the information in that SQLCA reflects the values set by the application server when it processed the last sub-statement. For instance:
  - The SQLCODE and SQLSTATE are normally those for the last sub-statement (the exception is described in the next point).
  - If a "no data found" warning (SQLSTATE 02000) is returned, that warning is given precedence over any other warning so that a WHENEVER NOT FOUND exception can be acted upon. (This means that the SQLCODE, SQLERRML, SQLERRMC, and SQLERRP fields in the SQLCA that is eventually returned to the application are those from the sub-statement that triggered the "no data found" warning. If there is more than one "no data found" warning within the compound SQL (embedded) statement, the fields for the last sub-statement will be the fields that are returned.)
  - The SQLWARN indicators are an accumulation of the indicators set for all sub-statements.
- If one or more errors occurred during NOT ATOMIC compound SQL execution and none of these are of a serious nature, the SQLERRMC will contain information about these errors, up to a maximum of seven errors. The first token of the SQLERRMC will indicate the total number of errors that occurred. The remaining tokens will each contain the ordinal position and the SQLSTATE of the failing sub-statement within the compound SQL (embedded) statement. The format is a character string of the form:

  ```
  nnnXsssccccc
  ```

  with the substring starting with X repeating up to six more times and the string elements defined as follows.

  nnn
  The total number of statements that produced errors. (If the number would exceed 999, counting restarts at zero.) This field is left-aligned and padded with blanks.

  X
  The token separator X'FF'.

  sss
  The ordinal position of the statement that caused the error. (If the number would exceed 999, counting restarts at zero.) For example, if the first statement failed, this field would contain the number one left-aligned ("1").

  ccccc
  The SQLSTATE of the error.
- The second SQLERRD field contains the number of statements that failed (returned negative SQLCODEs).
- The third SQLERRD field in the SQLCA is an accumulation of the number of rows affected by all sub-statements.
- The fourth SQLERRD field in the SQLCA is a count of the number of successful sub-statements. If, for example, the third sub-statement in a compound SQL (embedded) statement failed, the fourth SQLERRD field would be set to 2, indicating that 2 sub-statements were successfully processed before the error was encountered.
- The fifth SQLERRD field in the SQLCA is an accumulation of the number of rows updated or deleted due to the enforcement of referential integrity constraints for all sub-statements that triggered such constraint activity.

Examples

- *Example 1:* In a C program, issue a compound SQL (embedded) statement that updates both the ACCOUNTS and TELLERS tables. If there is an error in any of the statements, undo the effect of all statements (ATOMIC). If there are no errors, commit the current unit of work.

```
EXEC SQL BEGIN COMPOUND ATOMIC STATIC
  UPDATE ACCOUNTS SET ABALANCE = ABALANCE + :delta
    WHERE AID = :aid;
  UPDATE TELLERS SET TBALANCE = TBALANCE + :delta
    WHERE TID = :tid;
  INSERT INTO TELLERS (TID, BID, TBALANCE) VALUES (:i, :branch_id, 0);
  COMMIT;
END COMPOUND;
```

- *Example 2:* In a C program, insert 10 rows of data into the database. Assume the host variable :nbr contains the value 10 and S1 is a prepared INSERT statement. Further, assume that all the inserts should be attempted regardless of errors (NOT ATOMIC).

```
EXEC SQL BEGIN COMPOUND NOT ATOMIC STATIC STOP AFTER FIRST :nbr STATEMENTS
  EXECUTE S1 USING DESCRIPTOR :*sqlda0;
  EXECUTE S1 USING DESCRIPTOR :*sqlda1;
  EXECUTE S1 USING DESCRIPTOR :*sqlda2;
  EXECUTE S1 USING DESCRIPTOR :*sqlda3;
  EXECUTE S1 USING DESCRIPTOR :*sqlda4;
  EXECUTE S1 USING DESCRIPTOR :*sqlda5;
  EXECUTE S1 USING DESCRIPTOR :*sqlda6;
  EXECUTE S1 USING DESCRIPTOR :*sqlda7;
  EXECUTE S1 USING DESCRIPTOR :*sqlda8;
  EXECUTE S1 USING DESCRIPTOR :*sqlda9;
END COMPOUND;
```

Compound SQL (compiled)

A compound SQL (compiled) statement can contain SQL control statements and SQL statements. Compound SQL (compiled) statements can be used to implement procedural logic through a sequence of SQL statements with a local scope for variables, conditions, cursors, and handlers.

Invocation

This statement can be embedded in a trigger, SQL function, or SQL procedure; or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

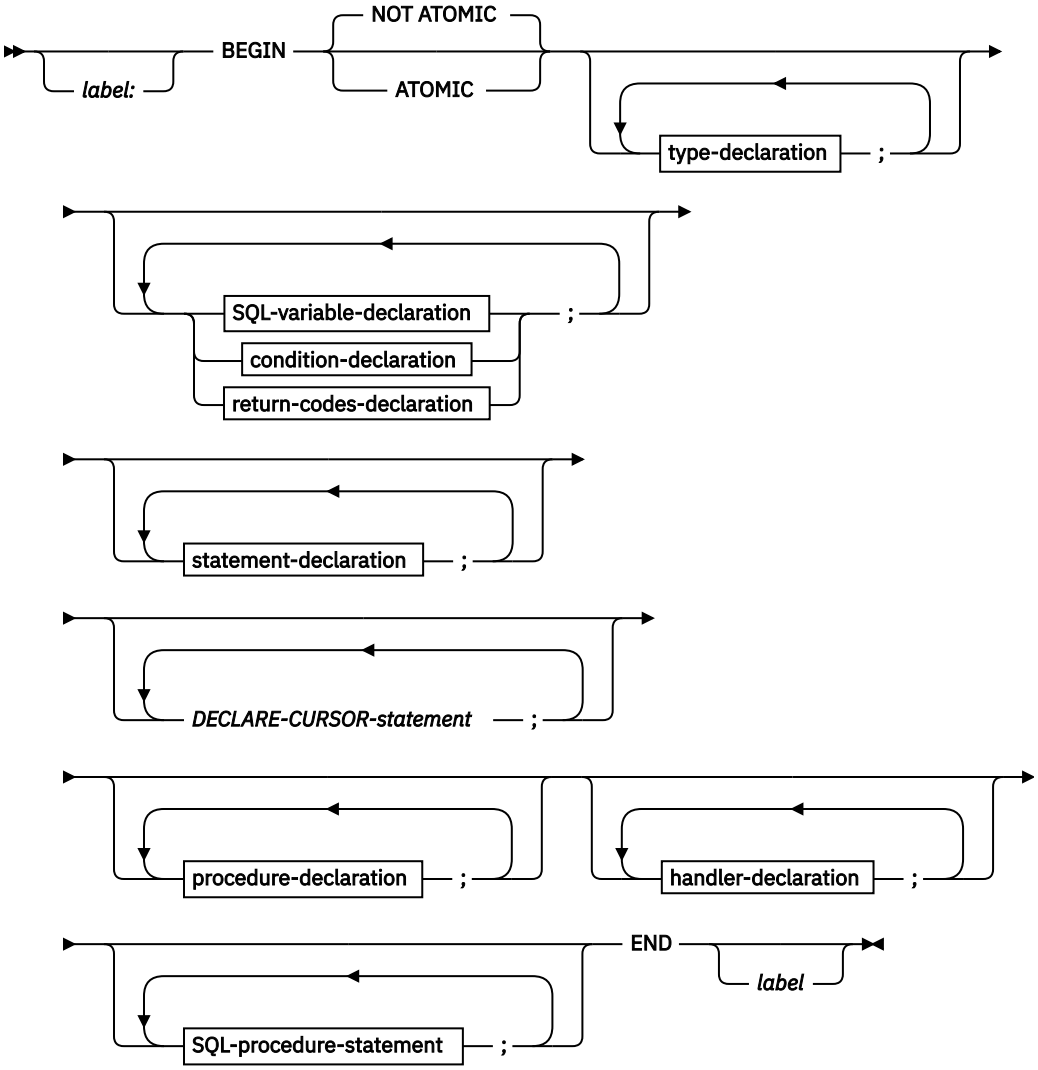
Authorization

For an *SQL-variable-declaration* that specifies a *cursor-value-constructor* that uses a *select-statement*, the privileges held by the authorization ID of the statement must include the privileges necessary to execute the *select-statement*. See the Authorization section in "SQL queries".

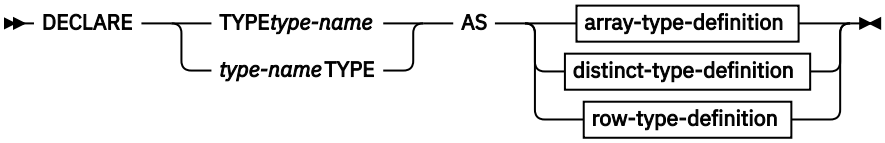
The privileges held by the authorization ID of the statement must also include all of the privileges necessary to invoke the SQL statements that are specified in the compound statement.

Only PUBLIC group privileges are considered for any SQL objects specified inside the body of compound statement.

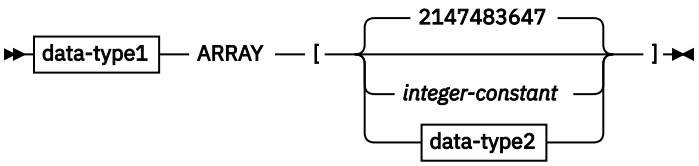
Syntax



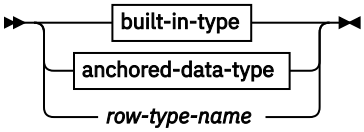
type-declaration



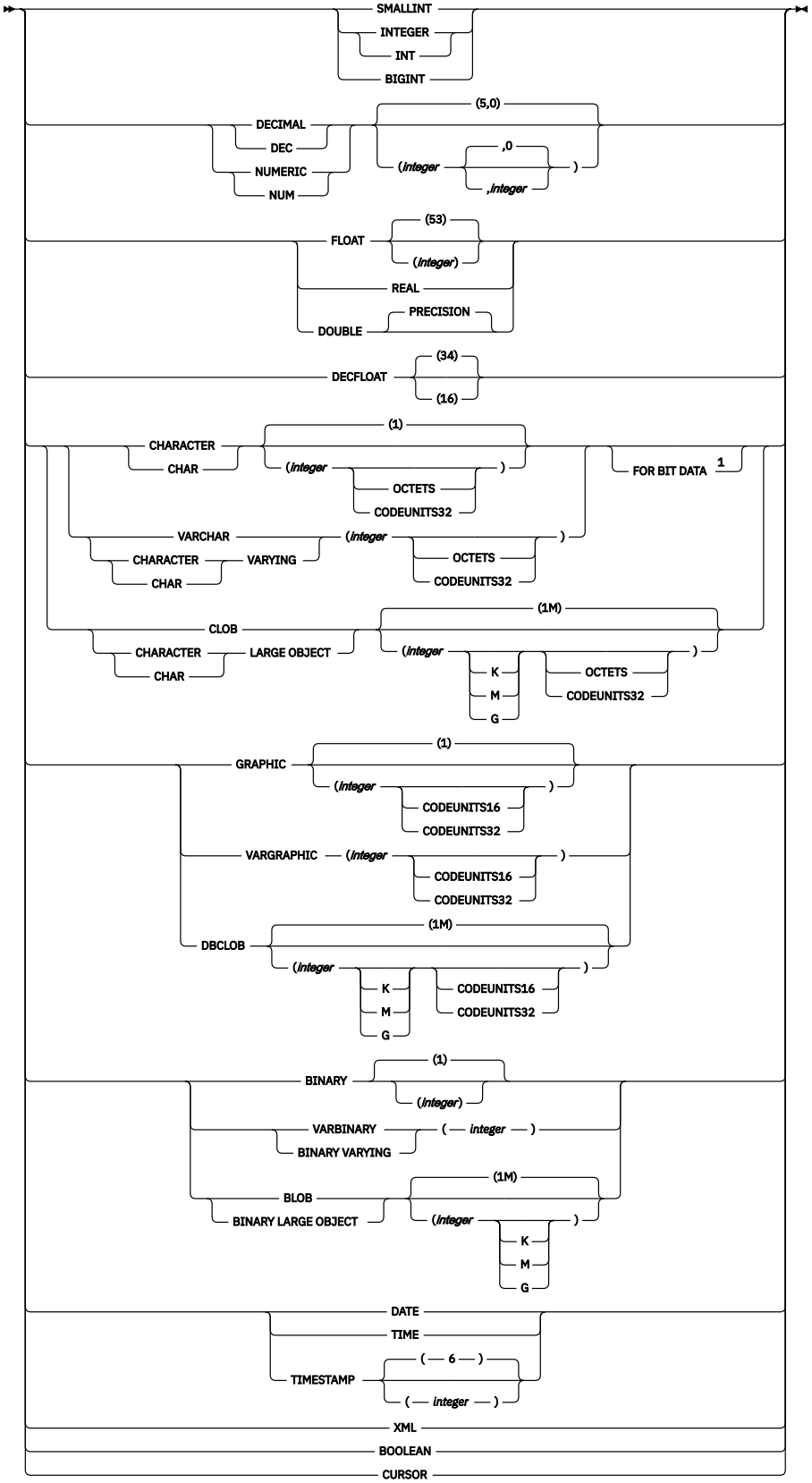
array-type-definition



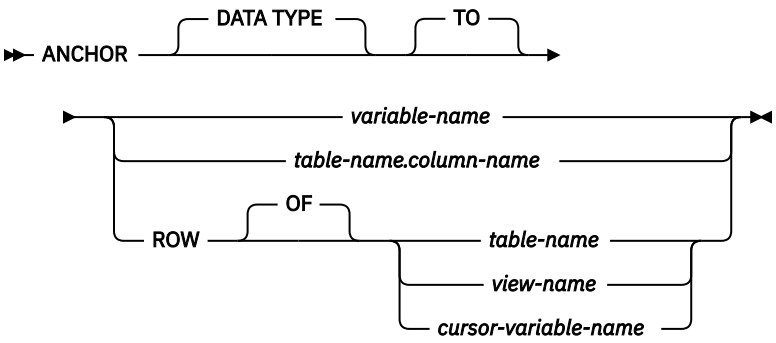
data-type1



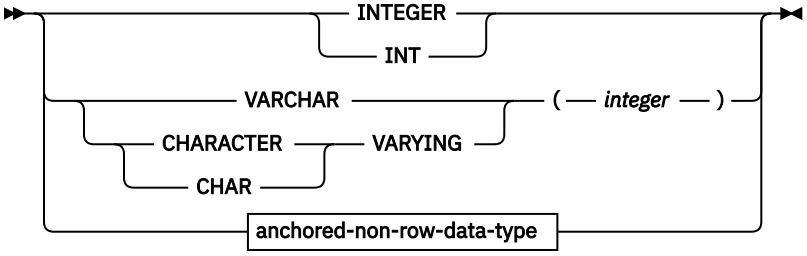
built-in-type



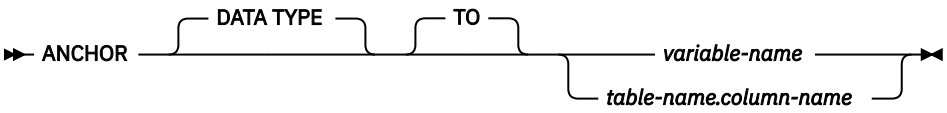
anchored-data-type



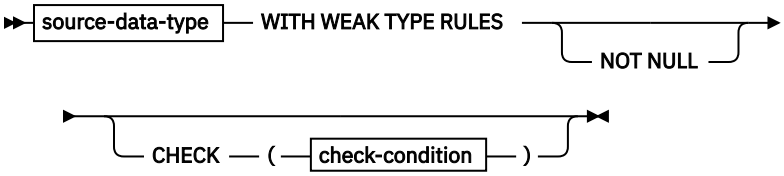
data-type2



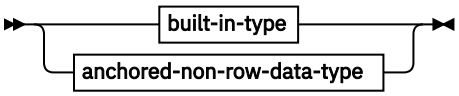
anchored-non-row-data-type



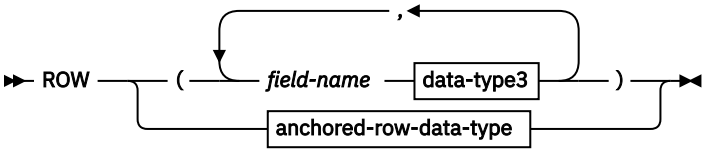
distinct-type-definition



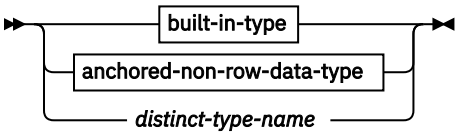
source-data-type



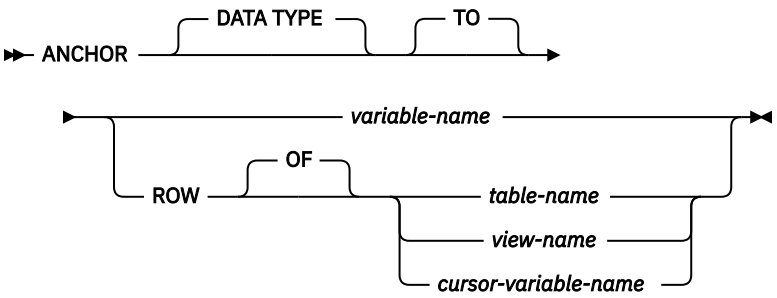
row-type-definition



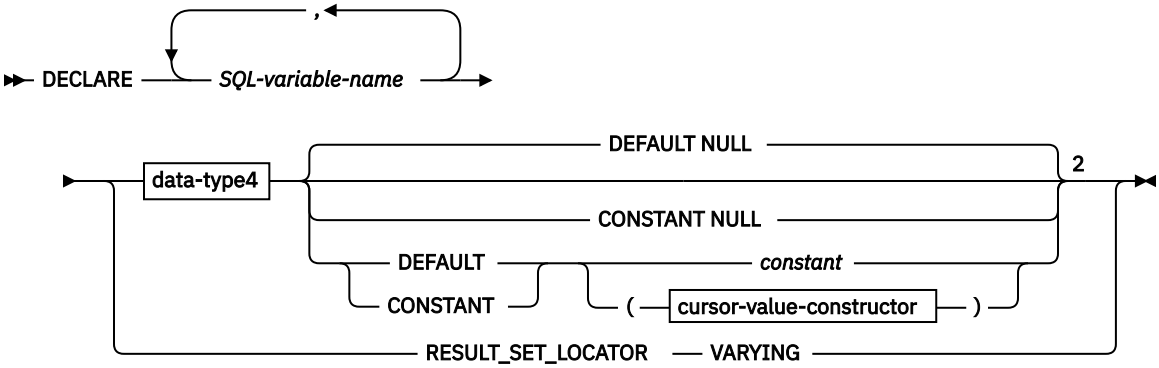
data-type3



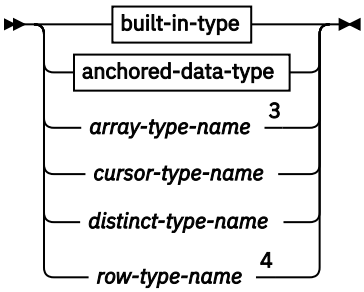
anchored-row-data-type



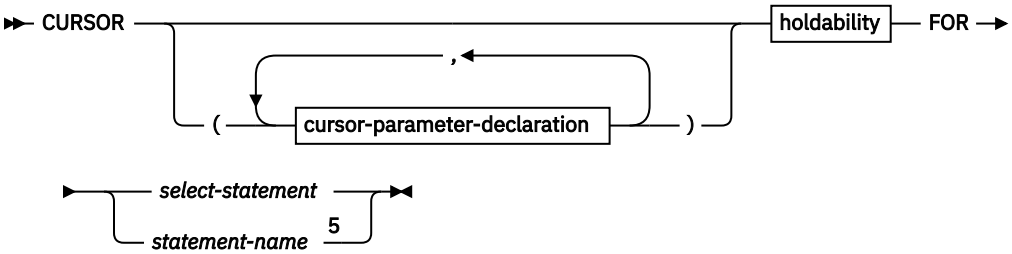
SQL-variable-declaration



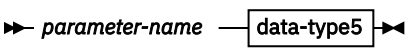
data-type4



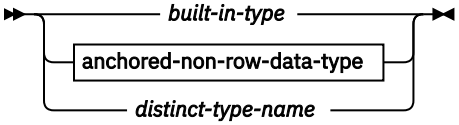
cursor-value-constructor



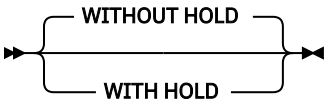
cursor-parameter-declaration



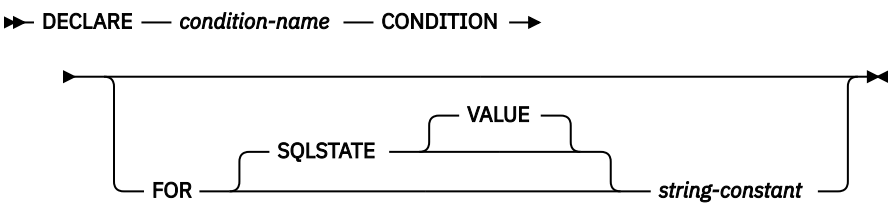
data-type5



holdability

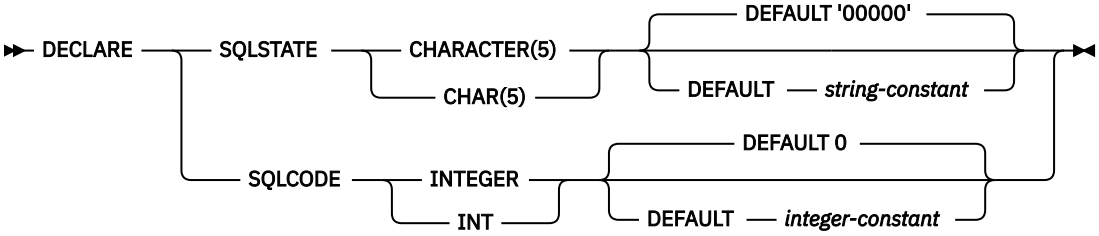


condition-declaration

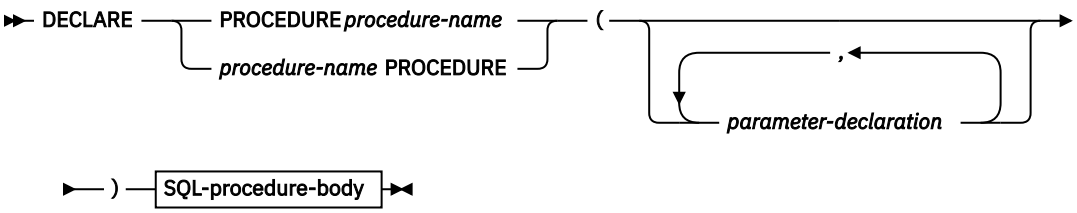


statement-declaration

return-codes-declaration

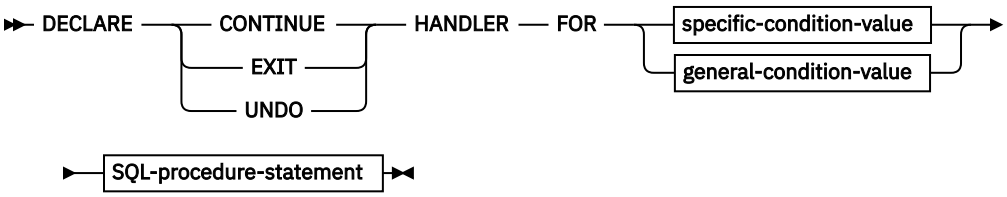


procedure-declaration

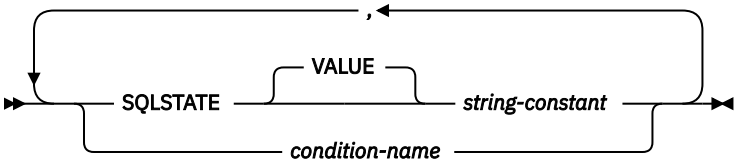


SQL-procedure-body

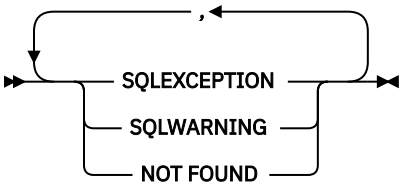
handler-declaration



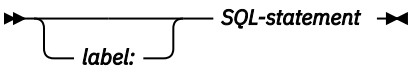
specific-condition-value



general-condition-value



SQL-procedure-statement



Notes:

[1] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).
[2] If *data-type4* specifies a CURSOR built-in type or *cursor-type-name*, only NULL or *cursor-value-constructor* can be specified. Only DEFAULT NULL can be explicitly specified for *array-type-name* or *row-type-name* .
[3] Only DEFAULT NULL can be explicitly specified for *array-type-name*.
[4] Only DEFAULT NULL can be explicitly specified for *row-type-name*.
[5] *statement-name* cannot be specified if *cursor-parameter-declaration* is specified.

Description

label
Defines the label for the code block. If the beginning label is specified, it can be used to qualify SQL variables declared in the compound statement and can also be specified on a LEAVE statement. If the ending label is specified, it must be the same as the beginning label.

ATOMIC or NOT ATOMIC
ATOMIC indicates that if an unhandled exception condition occurs in the compound statement, all SQL statements in the compound statement will be rolled back.

NOT ATOMIC indicates that an unhandled exception condition within the compound statement does not cause the compound statement to be rolled back.

If the ATOMIC keyword is specified in a dynamically prepared compound statement or an SQL function that is not within a module, the compound statement is processed as a compound SQL (inlined) statement.

A compound statement that is used in the function body of a module table function can only be defined as NOT ATOMIC.

type-declaration
Declares a user-defined data type that is local to the compound statement.

type-name
Specifies the name of a local user-defined data type. The name cannot be the same as any other type declared within the current compound statement (SQLSTATE 42734). The unqualified *type-name* has the same restrictions as described in any CREATE TYPE statement (SQLSTATE 42939).

array-type-definition
Specifies the attributes of an array data type to associate with the *type-name*. See "CREATE TYPE (array)" for a description of the syntax elements. The *row-type-name* can refer to a declared row type that is previously declared and in the scope of the current compound SQL (compiled) statement. The *variable-name* specified in an anchored-data-type clause can refer to a local variable in the scope of the current compound SQL (compiled) statement.

distinct-type-definition
Specifies the source type and optional data type constraints of a weakly typed distinct type to associate with the *type-name*. See "CREATE TYPE (distinct)" for a complete description of the

syntax elements. The *variable-name* specified in anchored-non-row-data-type clause can refer to a local variable in the scope of the current compound SQL (compiled) statement. The data type of the anchor *variable-name* or *column-name* must be a built-in data type.

row-type-definition
Specifies the fields of a row data type to associate with the *type-name*. See "CREATE TYPE (row)" for a complete description of the syntax elements. The *variable-name* specified in anchored-non-row-data-type or anchored-row-data-type clauses can refer to a local variable in the scope of the current compound SQL (compiled) statement.

SQL-variable-declaration
Declares a variable that is local to the compound statement.

SQL-variable-name
Defines the name of a local variable. All SQL variable names are converted to uppercase. The name cannot be the same as another SQL variable within the same compound statement and cannot be the same as a parameter name. An SQL variable name must not be the same as a column name. If an SQL statement contains an identifier with the same name as an SQL variable and a column reference, the identifier is interpreted as a column. If the compound statement in which the variable is declared has a label, then references to the variable can be qualified with the label. For example, variable V declared in a compound statement with a label C can be referred to as C.V.

data-type4
Specifies the data type of the variable. A structured type or reference type cannot be specified (SQLSTATE 429BB).

built-in-type
Specifies a built-in data type. For a more complete description of each built-in data type except BOOLEAN and CURSOR, which cannot be specified for a table, see "CREATE TABLE". The XML data type cannot be specified in a compound SQL (compiled) statement used in a trigger, in a function, or as a stand-alone statement (SQLSTATE 429BB). The XML data type can be specified when the compound SQL (compiled) statement is used in an SQL procedure body.

BOOLEAN
For a Boolean.

CURSOR
For a cursor.

anchored-data-type
Identifies another object used to determine the data type of the SQL variable. The data type of the anchor object has the same limitations that apply to specifying the data type directly, or in the case of a row, to creating a row type.

ANCHOR DATA TYPE TO
Indicates an anchored data type is used to specify the data type.

variable-name
Identifies an SQL variable, SQL parameter, or global variable. The data type of the referenced variable is used as the data type for *SQL-variable-name*.

table-name.column-name
Identifies a column name of an existing table or view. The data type of the column is used as the data type for *SQL-variable-name*.

ROW OF *table-name* or *view-name*
Specifies a row of fields with names and data types that are based on the column names and column data types of the table identified by *table-name* or the view identified by *view-name*. The data type of *SQL-variable-name* is an unnamed row type.

ROW OF *cursor-variable-name*
Specifies a row of fields with names and data types that are based on the field names and field data types of the cursor variable identified by *cursor-variable-name*. The specified cursor variable must be one of the following elements (SQLSTATE 428HS):

- An SQL variable or global variable with a strongly typed cursor data type
- An SQL variable or global variable with a weakly typed cursor data type that was created or declared with a CONSTANT clause specifying a *select-statement* where all the result columns are named.

If the cursor type of the cursor variable is not strongly rtyped using a named row type, the data type of *SQL-variable-name* is an unnamed row type.

array-type-name
Specifies the name of a user-defined array type. The array data type can be a locally declared data type, a schema data type, or a module data type.

cursor-type-name
Specifies the name of a cursor type. The cursor data type can be a schema data type or a module data type.

distinct-type-name
Specifies the name of a distinct type. The distinct data type can be a schema data type or a module data type. The length, precision, and scale of the declared variable are, respectively, the length, precision, and scale of the source type of the distinct type.

row-type-name
Specifies the name of a user-defined row type. The row data type can be a locally declared data type, a schema data type or a module data type. The fields of the variable are the fields of the row type.

DEFAULT or CONSTANT
Specifies a value for the SQL variable when the compound SQL (compiled) statement is referenced. If neither is specified, the default for the SQL variable is the null value. Only DEFAULT NULL can be explicitly specified if *array-type-name* or *row-type-name* is specified.

DEFAULT
Defines the default for the SQL variable. The variable is initialized when the compound SQL (compiled) statement is referenced. The default value must be assignment-compatible with the data type of the variable.

CONSTANT
Specifies that the SQL variable has a fixed value that cannot be changed. An SQL variable that is defined using CONSTANT cannot be used as the target of any assignment operation. The fixed value must be assignment-compatible with the data type of the variable.

NULL
Specifies NULL as the default for the SQL variable.

constant
Specifies a constant as the default for the SQL variable. If *data-type4* specifies a CURSOR built-in type or *cursor-type-name*, *constant* cannot be specified (SQLSTATE 42601).

cursor-value-constructor
A *cursor-value-constructor* specifies the *select-statement* that is associated with the SQL variable. The assignment of a *cursor-value-constructor* to a cursor variable defines the underlying cursor of that cursor variable.

(cursor-parameter-declaration, ...)
Specifies the input parameters of the cursor, including the name and the data type of each parameter. Named input parameters can be specified only if *select-statement* is also specified in *cursor-value-constructor* (SQLSTATE 428HU).

parameter-name
Names the cursor parameter for use as an SQL variable within *select-statement*. The name cannot be the same as any other parameter name for the cursor. Names should also be chosen to avoid any column names that could be used in *select-statement*, since column names are resolved before parameter names.

data-type5
Specifies the data type of the cursor parameter used within *select-statement*. Structured types, and reference types cannot be specified (SQLSTATE 429BB).

built-in-type
Specifies a built-in data type. For a more complete description of each built-in data type, see "CREATE TABLE". The BOOLEAN and CURSOR built-in types cannot be specified (SQLSTATE 429BB).

anchored-non-row-data-type
Identifies another object used to determine the data type of the cursor parameter. The data type of the anchor object has the same limitations that apply to specifying the data type directly.

ANCHOR DATA TYPE TO
Indicates an anchored data type is used to specify the data type.

variable-name
Identifies a local SQL variable, an SQL parameter, or a global variable. The data type of the referenced variable is used as the data type for the cursor parameter.

table-name.column-name
Identifies a column name of an existing table or view. The data type of the column is used as the data type for the cursor parameter.

distinct-type-name
Specifies the name of a distinct type. If *distinct-type-name* is specified without a schema name, the distinct type is resolved by searching the schemas in the SQL path.

holdability
Specifies whether the cursor is prevented from being closed as a consequence of a commit operation. See "DECLARE CURSOR" for more information. The default is WITHOUT HOLD.

WITHOUT HOLD
Does not prevent the cursor from being closed as a consequence of a commit operation.

WITH HOLD
Maintains resources across multiple units of work. Prevents the cursor from being closed as a consequence of a commit operation.

select-statement
Specifies the SELECT statement of the cursor. See "select-statement" for more information. If *cursor-parameter-declaration* is included in *cursor-value-constructor*, then *select-statement* must not include any local SQL variables or routine SQL parameters (SQLSTATE 42704).

statement-name
Specifies the prepared *select-statement* of the cursor. See "PREPARE" for an explanation of prepared statements. The target cursor variable must not have a data type that is a strongly rtyped user-defined cursor type (SQLSTATE 428HU). Named input parameters must not be specified in *cursor-value-constructor* if *statement-name* is specified (SQLSTATE 428HU).

RESULT_SET_LOCATOR VARYING
Specifies the data type for a result set locator variable.

condition-declaration
Declares a condition name with an optional associated SQLSTATE value.

condition-name
Specifies the name of the condition. The condition name must be unique within the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement (SQLSTATE 42734). A condition name can only be

referenced within the compound statement in which it is declared, including any compound statements that are nested within that compound statement (SQLSTATE 42737).

CONDITION FOR SQLSTATE VALUE*string-constant*
Specifies the SQLSTATE that is associated with the condition. The string constant must be specified as five characters enclosed in single quotation marks, and the SQLSTATE class (the first two characters) must not be '00'. If this clause is not specified, the condition has no associated SQLSTATE value.

statement-declaration
Declares a list of one or more names that are local to the compound statement. Each name in *statement-name* must not be the same as any other statement name declared in the same compound statement.

return-codes-declaration
Declares special variables called SQLSTATE and SQLCODE that are set automatically to the value returned after processing an SQL statement. Both the SQLSTATE and SQLCODE variables can only be declared in the outermost compound statement when there are nested compound SQL (compiled) statements; for example in an SQL procedure body. These variables may be declared only once per SQL procedure.

declare-cursor-statement
Declares a built-in cursor in the procedure body. Variables of user-defined cursor data types are declared using *SQL-variable-declaration* statements.

Each declared cursor must have a unique name within the compound statement in which it is declared, excluding any declarations in compound statements that are nested within that compound statement (SQLSTATE 42734). The cursor can be referenced only from within the compound statement in which it is declared, including any compound statements that are nested within that compound statement (SQLSTATE 34000).

Use an OPEN statement to open the cursor, and a FETCH statement to read rows using the cursor. To return result sets from the SQL procedure to the client application, the cursor must be declared using the WITH RETURN clause. The following example returns one result set to the client application:

```
CREATE PROCEDURE RESULT_SET()
  LANGUAGE SQL
  RESULT SETS 1
  BEGIN
    DECLARE C1 CURSOR WITH RETURN FOR
      SELECT id, name, dept, job
        FROM staff;
    OPEN C1;
END
```

Note: To process result sets, you must write your client application using one of the Db2 Call Level Interface (Db2 Call Level Interface), Open Database Connectivity (ODBC), Java Database Connectivity (JDBC), or embedded SQL for Java (SQLJ) application programming interfaces.

For more information about declaring a cursor, see "DECLARE CURSOR".

procedure-declaration
Declares a procedure that is local to the compound statement. The definition of a local procedure does not include the specification of any of the options possible in a "CREATE PROCEDURE (SQL)" statement. The options default as they would for a "CREATE PROCEDURE (SQL)" statement with the exception of MODIFIES SQL DATA. The data access level for the procedure is automatically determined to be the minimum level required to process the SQL procedure body.

procedure-name
Defines the names of a local procedure. The name must be specified without any qualification (SQLSTATE 42601). The procedure signature, consisting of the *procedure-name* and the number of declared parameters, must be unique within the current compound statement. Outer compound statements within which the current compound statement is nested cannot contain a procedure with the same name.

parameter-declaration
Specifies the parameters of the local procedure. See "CREATE PROCEDURE (SQL)" for a description of the syntax elements. The parameter data type can be a locally declared data type in the scope of the current compound statement.

SQL-procedure-body
Specifies the SQL statement that is the body of the SQL procedure. Names referenced in the *SQL-procedure-body* can refer to declared objects (such as declared variables, data types, and procedures) that are previously declared and in the scope of the compound statement in which the local procedure is declared.

handler-declaration
Specifies a *handler*, and a set of one or more *SQL-procedure-statements* to execute when an exception or completion condition occurs in the compound statement.*SQL-procedure-statement* is a statement that executes when the handler receives control.

A handler is said to be active for the duration of the execution of the set of *SQL-procedure-statements* that follow the set of *handler-declarations* within the compound statement in which the handler is declared, including any nested compound statements.

There are three types of condition handlers:

CONTINUE
After the handler is invoked successfully, control is returned to the SQL statement that follows the statement that raised the exception. If the error that raised the exception is a FOR, IF, CASE, WHILE, or REPEAT statement (but not an SQL-procedure-statement within one of these), then control returns to the statement that follows END FOR, END IF, END CASE, END WHILE, or END REPEAT.

EXIT
After the handler is invoked successfully, control is returned to the end of the compound statement that declared the handler.

UNDO
Before the handler is invoked, any SQL changes that were made in the compound statement are rolled back. After the handler is invoked successfully, control is returned to the end of the compound statement that declared the handler. If UNDO is specified, the compound statement where the handler is declared must be ATOMIC.

The conditions that cause the handler to be activated are defined in the handler-declaration as follows:

specific-condition-value
Specifies that the handler is a *specific condition handler*.

SQLSTATE VALUE*string-constant*
Specifies an SQLSTATE for which the handler is invoked. The first two characters of the SQLSTATE value must not be '00'.

condition-name
Specifies a condition name for which the handler is invoked. The condition name must be previously defined in a condition declaration or it must identify a condition that exists at the current server.

general-condition-value
Specifies that the handler is a *general condition handler*.

SQLEXCEPTION
Specifies that the handler is invoked when an exception condition occurs. An exception condition is represented by an SQLSTATE value whose first two characters are not '00', '01', or '02'.

SQLWARNING
Specifies that the handler is invoked when a warning condition occurs. A warning condition is represented by an SQLSTATE value whose first two characters are '01'.

NOT FOUND
Specifies that the handler is invoked when a NOT FOUND condition occurs. A NOT FOUND condition is represented by an SQLSTATE value whose first two characters are '02'.

SQL-procedure-statement
Specifies the SQL procedure statement.

label
Specifies a label for the SQL procedure statement. The label must be unique within a list of SQL procedure statements, including any compound statements nested within the list. Note that compound statements that are not nested can use the same label. A list of SQL procedure statements is possible in a number of SQL control statements.

SQL-statement
All executable SQL statements except for:

- ALTER
- CONNECT
- CREATE
- DESCRIBE
- DISCONNECT
- DROP
- FLUSH EVENT MONITOR
- FREE LOCATOR
- GRANT
- REFRESH TABLE
- RELEASE (connection only)
- RENAME TABLE
- RENAME TABLESPACE
- REVOKE
- SET CONNECTION
- SET INTEGRITY
- SET PASSTHRU
- SET SERVER OPTION
- TRANSFER OWNERSHIP

The following executable statements are not supported in stand-alone compound SQL (compiled) statements, but are supported in compound SQL (compiled) statements used within an SQL function, SQL procedure, or trigger:

- CREATE of an index, table, or view
- DECLARE GLOBAL TEMPORARY TABLE
- DROP of an index, table, or view
- GRANT
- ROLLBACK

The ROLLBACK statement is also not supported in any nested statement invoked within the stand-alone compound SQL (compiled) statement.

The following statements, which are not executable statements, are supported in compound SQL (compiled) statements:

- ALLOCATE CURSOR
- ASSOCIATE LOCATORS

Rules

- ATOMIC compound statements cannot be nested.
- The following rules apply to handler declarations:
  - A handler declaration cannot contain the same *condition-name* or SQLSTATE value more than once, and cannot contain an SQLSTATE value and a *condition-name* that represent the same SQLSTATE value.
  - Where two or more condition handlers are declared in a compound statement:
    - No two handler declarations may specify the same general condition category (SQLEXCEPTION, SQLWARNING, NOT FOUND).
    - No two handler declarations may specify the same specific condition, either as an SQLSTATE value or as a *condition-name* that represents the same value.
  - A handler is activated when it is the most appropriate handler for an exception or completion condition. The most appropriate handler is determined based on the following considerations:
    - The scope of a handler declaration *H* is the list of *SQL-procedure-statement* that follows the handler declarations contained within the compound statement in which *H* appears. This means that the scope of *H* does not include the statements contained in the body of the condition handler *H*, implying that a condition handler cannot handle conditions that arise inside its own body. Similarly, for any two handlers *H1* and *H2* declared in the same compound statement, *H1* will not handle conditions arising in the body of *H2*, and *H2* will not handle conditions arising in the body of *H1*.
    - A handler for a *specific-condition-value* or a *general-condition-value* C declared in an inner scope takes precedence over another handler for C declared in an enclosing scope.
    - When a specific handler for condition *C* and a general handler which would also handle *C* are declared in the same scope, the specific handler takes precedence over the general handler.
    - When a handler for a module condition that has no associated SQLSTATE value and a handler for SQLSTATE 45000 are declared in the same scope, the handler for the module condition takes precedence over the handler for SQLSTATE 45000.

    If an exception condition occurs for which there is no appropriate handler, the SQL procedure containing the failing statement is terminated with an unhandled exception condition. If a completion condition occurs for which there is no appropriate handler, execution continues with the next SQL statement.
- Referencing variables or parameters of data type XML in SQL procedures after a commit or rollback operation occurs, without first assigning new values to these variables, is not supported (SQLSTATE 560CE).
- ***Use of anchored data types:*** An anchored data type cannot refer to the following objects (SQLSTATE 428HS): a nickname, typed table, typed view, statistical view that is associated with an expression-based index, declared temporary table, row definition that is associated with a weakly typed cursor, object with a code page or collation that is different from the database code page or database collation.
- If named parameter markers are used in a compound SQL (compiled) statement that is dynamically prepared or executed, every parameter marker name must be unique (SQLSTATE 42997).

Notes

- ***XML assignments:*** Assignment to parameters and variables of data type XML is done by reference.

  Passing parameters of data type XML in a CALL statement to an SQL procedure is done by reference. When XML values are passed by reference, any input node trees are used directly from the XML argument. This direct usage preserves all properties, including document order, the original node identities, and all parent properties.

Examples

- *Example 1*: A simple stand-alone compound statement that outputs the word 'Hello':

```
SET SERVEROUTPUT ON;
BEGIN
    CALL DBMS_OUTPUT.PUT_LINE ( 'Hello' );
END
```

- *Example 2*: A simple stand-alone compound statement that counts the number of records in `staff` and outputs the result:

```
SET SERVEROUTPUT ON;
BEGIN
    DECLARE v_numRecords INTEGER DEFAULT 1;
 SELECT COUNT(*) INTO v_numRecords FROM staff;

 CALL DBMS_OUTPUT.PUT_LINE (v_numRecords);
END
```

- *Example 3*: Create a procedure with a compound SQL (compiled) statement that performs the following actions:

  1. Declares SQL variables
  2. Declares a cursor to return the salary of employees in a department determined by an IN parameter. In the SELECT statement, casts the data type of the *salary* column from a DECIMAL into a DOUBLE.
  3. Declares an EXIT handler for the condition NOT FOUND (end of file) which assigns the value '6666' to the OUT parameter `medianSalary`
  4. Select the number of employees in the given department into the SQL variable `numRecords`
  5. Fetch rows from the cursor in a WHILE loop until 50% + 1 of the employees have been retrieved
  6. Return the median salary

```
CREATE PROCEDURE DEPT_MEDIAN
  (IN deptNumber SMALLINT, OUT medianSalary DOUBLE)
  LANGUAGE SQL
  BEGIN
    DECLARE v_numRecords INTEGER DEFAULT 1;
    DECLARE v_counter INTEGER DEFAULT 0;
    DECLARE c1 CURSOR FOR
      SELECT CAST(salary AS DOUBLE) FROM staff
        WHERE DEPT = deptNumber
        ORDER BY salary;
    DECLARE EXIT HANDLER FOR NOT FOUND
      SET medianSalary = 6666;
-- initialize OUT parameter
      SET medianSalary = 0;
      SELECT COUNT(*) INTO v_numRecords FROM staff
        WHERE DEPT = deptNumber;
      OPEN c1;
      WHILE v_counter < (v_numRecords / 2 + 1) DO
        FETCH c1 INTO medianSalary;
        SET v_counter = v_counter + 1;
      END WHILE;
      CLOSE c1;
  END
```

- *Example 4*: The following example illustrates the flow of execution in a hypothetical case where an UNDO handler is activated from another condition as the result of RESIGNAL:

```
CREATE PROCEDURE A()
LANGUAGE SQL
CS1: BEGIN ATOMIC
  DECLARE C CONDITION FOR SQLSTATE '12345';
  DECLARE D CONDITION FOR SQLSTATE '23456';

  DECLARE UNDO HANDLER FOR C
  H1: BEGIN
    -- Perform rollback after error, perform final cleanup, and exit
    -- procedure A.

    -- ...
```

```
      -- When this handler completes, execution continues after
      -- compound statement CS1; procedure A will terminate.
    END;

    -- Perform some work here ...
    CS2: BEGIN
      DECLARE CONTINUE HANDLER FOR D
      H2: BEGIN
        -- Perform local recovery, then forward the error
        -- condition to the outer handler for additional
        -- processing.

        -- ...

        RESIGNAL C; -- will activate UNDO handler H1; execution
                    -- WILL NOT return here.  Any local cursors
                    -- declared in H2 and CS2 will be closed.
      END;

      -- Perform some more work here ...

      -- Simulate raising of condition D by some SQL statement
      -- in compound statement CS2:
      SIGNAL D; -- will activate H2
    END;
  END
```

CONNECT (type 1)

The CONNECT (Type 1) statement connects an application process to the identified application server according to the rules for remote unit of work.

An application process can only be connected to one application server at a time. This is called the *current server*. A default application server may be established when the application requester is initialized. If implicit connect is available and an application process is started, it is implicitly connected to the default application server. The application process can explicitly connect to a different application server by issuing a CONNECT statement. A connection lasts until a CONNECT RESET statement or a DISCONNECT statement is issued or until another CONNECT statement changes the application server.

Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared. When invoked using the command line processor, additional options can be specified.

For more information, refer to "Using command line SQL statements and XQuery statements" in *Command Reference*.

Authorization

CONNECT processing goes through two levels of access control. Both levels must be satisfied for the connection to be successful.

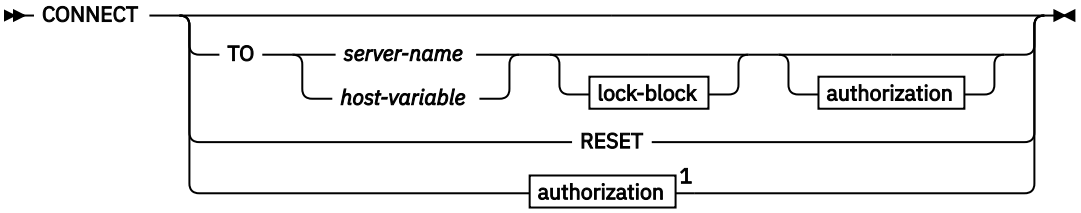
The first level of access control is authentication, where the user ID associated with the connection must be successfully authenticated according to the authentication method set up for the server. At successful authentication, a database authorization ID is derived from the connection user ID according to the authentication plug-in in effect for the server. This database authorization ID must then pass the second level of access control for the connection, that is, authorization. To do so, this authorization ID must hold at least one of the following authorities:

- CONNECT authority
- SECADM authority
- DBADM authority
- SYSADM authority

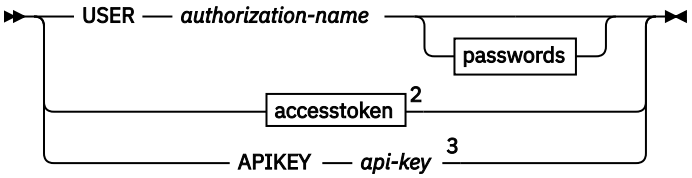
- SYSCTRL authority
- SYSMAINT authority
- SYSMON authority

Note: For a partitioned database, the user and group definitions must be identical across all database partitions.

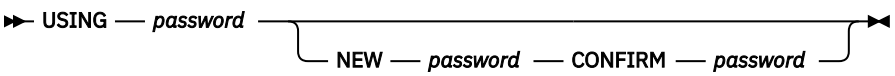
Syntax



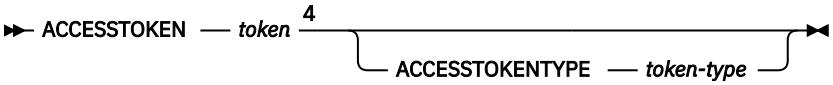
authorization



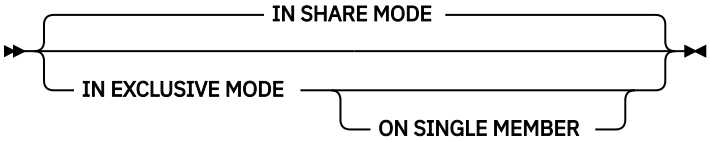
passwords



accesstoken



lock-block



Notes:

[1] This form is only valid if implicit connect is enabled.
[2] This feature is available starting from Db2 Version 11.5 Mod Pack 4.
[3] This feature is available starting from Db2 Version 11.5 Mod Pack 4.
[4] This feature is available starting from Db2 Version 11.5 Mod Pack 4.

Description

CONNECT (with no operand)

Returns information about the current server. The information is returned in the SQLERRP field of the SQLCA as described in "Successful Connection".

If a connection state exists, the authorization ID and database alias are placed in the SQLERRMC field of the SQLCA. If the authorization ID is longer than 8 bytes, it will be truncated to 8 bytes, and the truncation will be flagged in the SQLWARN0 and SQLWARN1 fields of the SQLCA, with 'W' and 'A', respectively.

If no connection exists and implicit connect is possible, then an attempt to make an implicit connection is made. If implicit connect is not available, this attempt results in an error (no existing connection). If no connection, then the SQLERRMC field is blank.

The territory code and code page of the application server are placed in the SQLERRMC field (as they are with a successful CONNECT statement).

This form of CONNECT:

- Does not require the application process to be in the connectable state.
- If connected, does not change the connection state.
- If unconnected and implicit connect is available, a connection to the default application server is made. In this case, the country or region code and code page of the application server are placed in the SQLERRMC field, like a successful CONNECT statement.
- If unconnected and implicit connect is not available, the application process remains unconnected.
- Does not close cursors.

TO *server-name* or *host-variable*

Identifies the application server by the specified *server-name* or a *host-variable* which contains the server-name.

If a *host-variable* is specified, it must be a character string variable with a length attribute that is not greater than 8, and it must not include an indicator variable. The *server-name* that is contained within the *host-variable* must be left-aligned and must not be delimited by quotation marks.

Note that the *server-name* is a database alias identifying the application server. It must be listed in the application requester's local directory.

When the CONNECT statement is executed, the application process must be in the connectable state.

Successful Connection

If the CONNECT statement is successful:

- All open cursors are closed, all prepared statements are destroyed, and all locks are released from the previous application server.
- The application process is disconnected from its previous application server, if any, and connected to the identified application server.
- The actual name of the application server (not an alias) is placed in the CURRENT SERVER special register.
- Information about the application server is placed in the SQLERRP field of the SQLCA. If the application server is an IBM product, the information has the form *pppvvrrm*, where:
  - *ppp* represents one of the following product identifiers:
    - DSN for Db2 for z/OS
    - ARI for Db2 Server for VSE & VM
    - QSQ for Db2 for IBM i
    - SQL for Db2
  - *vv* is a two-digit version identifier, such as '08'
  - *rr* is a two-digit release identifier, such as '01'
  - *m* is a one-character modification level identifier, such as '0'.

  For example, Version 9.5 of Db2 is identified as 'SQL09050'.
- The SQLERRMC field of the SQLCA is set to contain the following values (separated by X'FF')
  1. The country or region code of the application server (or blanks if using Db2 Connect),
  2. The code page of the application server (or CCSID if using Db2 Connect),
  3. The authorization ID (up to first 8 bytes only),

4. The database alias,
5. The platform type of the application server. Currently identified values are:

   Token
   Server

   QAS
   Db2 for IBM i

   QDB2
   Db2 for z/OS

   QDB2/6000
   Db2 Database for AIX

   QDB2/LINUX
   Db2 Database for Linux

   QDB2/NT
   Db2 Database for Windows

   QSQLDS/VM
   Db2 Server for VM

   QSQLDS/VSE
   Db2 Server for VSE
6. The agent ID. It identifies the agent executing within the database manager on behalf of the application. This field is the same as the **agent_id** element returned by the database monitor.
7. The agent index. It identifies the index of the agent and is used for service.
8. If the server instance operates in a Db2 pureScale environment, as indicated by SQLWARN0 and SQLWARN4 being set to 'W' and 'S' respectively, this value represents the member number. If, as indicated by token 10, the server instance operates in a partitioned environment, this token represents the member number. If the server instance operates in a non-partitioned environment and outside of a Db2 pureScale environment, this value is not applicable and is always 0.
9. The code page of the application client.
10. If this value is zero, the server instance operates in a non-partitioned environment and outside of a Db2 pureScale environment. Otherwise, this non-zero value represents the number of members in a Db2 pureScale instance, if SQLWARN0 and SQLWARN4 are set to 'W' and 'S' respectively. If this value is non-zero but neither SQLWARN0 nor SQLWARN4 is set, it represents the number of members in a partitioned environment.

- The SQLERRD(1) field of the SQLCA indicates the maximum expected difference in length of mixed character data (CHAR data types) when converted to the database code page from the application code page. A value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction.
- The SQLERRD(2) field of the SQLCA indicates the maximum expected difference in length of mixed character data (CHAR data types) when converted to the application code page from the database code page. A value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction.
- The SQLERRD(3) field of the SQLCA indicates whether or not the database on the connection is updatable. A database is initially updatable, but is changed to read-only if a unit of work determines the authorization ID cannot perform updates. The value is one of:
  - 1 - updatable
  - 2 - read-only
- The SQLERRD(4) field of the SQLCA returns certain characteristics of the connection. The value is one of:

0
N/A (only possible if running from a client which is not at the latest level, is one-phase commit, and is an updater).

1
one-phase commit.

2
one-phase commit; read-only (only applicable to connections to DRDA1 databases in a TP Monitor environment).

3
two-phase commit.

- The SQLERRD(5) field of the SQLCA returns the authentication type for the connection. The value is one of:

  0
  Authenticated on the server.

  1
  Authenticated on the client.

  2
  Authenticated using Db2 Connect.

  4
  Authenticated on the server with encryption.

  5
  Authenticated using Db2 Connect with encryption.

  7
  Authenticated using an external Kerberos security mechanism.

  9
  Authenticated using an external GSS API plug-in security mechanism.

  11
  Authenticated on the server, which accepts encrypted data.

  255
  Authentication not specified.
- The SQLERRD(6) field of the SQLCA returns the database partition number of the database partition to which the connection was made if in a partitioned database environment. Otherwise, a value of 0 is returned.
- The SQLWARN1 field in the SQLCA will be set to 'A' if the authorization ID of the successful connection is longer than 8 bytes. This indicates that truncation has occurred. The SQLWARN0 field in the SQLCA will be set to 'W' to indicate this warning.

Unsuccessful Connection

If the CONNECT statement is unsuccessful:

- The SQLERRP field of the SQLCA is set to the name of the module at the application requester that detected the error. The first three characters of the module name identify the product.
- If the CONNECT statement is unsuccessful because the application process is not in the connectable state, the connection state of the application process is unchanged.
- If the CONNECT statement is unsuccessful because the *server-name* is not listed in the local directory, an error message (SQLSTATE 08001) is issued and the connection state of the application process remains unchanged:
  - If the application requester was not connected to an application server then the application process remains unconnected.
  - If the application requester was already connected to an application server, the application process remains connected to that application server. Any further statements are executed at that application server.

- If the CONNECT statement is unsuccessful for any other reason, the application process is placed into the unconnected state.

IN SHARE MODE
Allows other concurrent connections to the database and prevents other users from connecting to the database in exclusive mode.

IN EXCLUSIVE MODE
Prevents concurrent application processes from executing any operations at the application server, unless they have the same authorization ID as the user holding the exclusive lock. This option is not supported by Db2 Connect.

ON SINGLE MEMBER
Specifies that the coordinator database member is connected in exclusive mode and all other members are connected in share mode.

If the database is neither in a partitioned environment nor a Db2 pureScale environment, this option can be specified, but it has no effect.

RESET
Disconnects the application process from the current server. A commit operation is performed. If implicit connect is available, the application process remains unconnected until an SQL statement is issued.

USER *authorization-name/host-variable*
Identifies the user ID trying to connect to the application server. If a host-variable is specified, it must be a character string variable that does not include an indicator variable. The user ID that is contained within the *host-variable* must be left-aligned and must not be delimited by quotation marks.

USING *password/host-variable*
Identifies the password of the user ID trying to connect to the application server. The maximum length of the password is determined by the data server you are connecting to. If a host variable is specified, it must be a character string variable and it must not include an indicator variable.

NEW *password/host-variable* CONFIRM *password*
Identifies the new password that should be assigned to the user ID identified by the USER option. The maximum length of the password is determined by the data server you are connecting to. If a host variable is specified, it must be a character string variable and it must not include an indicator variable. The system on which the password will be changed depends on how the user authentication has been set up. To support the changing passwords on Linux, the database instance must be configured to use the security plug-ins IBMOSchgpwdclient and IBMOSchgpwdserver.

Notes

- It is good practice for the first SQL statement executed by an application process to be the CONNECT statement.
- If a CONNECT statement is issued to the current application server with a different user ID and password then the conversation is deallocated and reallocated. All cursors are closed by the database manager (with the loss of the cursor position if the WITH HOLD option was used).
- If a CONNECT statement is issued to the current application server with the same user ID and password then the conversation is not deallocated and reallocated. Cursors, in this case, are not closed.
- To use a multiple-partition partitioned database environment, the user or application must connect to one of the database partitions listed in the `db2nodes.cfg` file. You should try to ensure that not all users use the same database partition as the coordinator partition.
- The *authorization-name* SYSTEM cannot be explicitly specified in the CONNECT statement. However, on Windows operating systems, local applications running under the Local System Account can implicitly connect to the database, such that the user ID is SYSTEM.
- When connecting to Windows Server explicitly, the *authorization-name* or user *host-variable* can be specified using the Microsoft Windows Security Account Manager (SAM)-compatible name.

- The database can be inaccessible if the database was not explicitly activated, a client application performs frequent reconnections, or the time interval between issuing the **DEACTIVATE DATABASE** and **ACTIVATE DATABASE** commands is very short. Activate the database by issuing the **ACTIVATE DATABASE** command and then attempt to connect to the database.
- ***Syntax alternatives***: The following are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - DBPARTITIONNUM or NODE can be specified in place of MEMBER, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.

Examples

- *Example 1:* In a C program, connect to the application server TOROLAB, using database alias TOROLAB, user ID FERMAT, and password THEOREM.

```
   EXEC SQL  CONNECT TO TOROLAB USER FERMAT USING THEOREM;
```

- *Example 2:* In a C program, connect to an application server whose database alias is stored in the host variable APP_SERVER (varchar(8)). Following a successful connection, copy the 3-character product identifier of the application server to the variable PRODUCT (char(3)).

```
   EXEC SQL  CONNECT TO :APP_SERVER;
   if (strncmp(SQLSTATE,'00000',5))
     strncpy(PRODUCT,sqlca.sqlerrp,3);
```

- *Example 3:* Connect to the SAMPLE database using a JWT access token.

```
connect to sample accesstoken <access_token> accesstokentype jwt

   Database Connection Information

 Database server        = DB2/LINUXX8664 11.5.4.0
 SQL authorization ID   = NEWTON
 Local database alias   = SAMPLE
```

CONNECT (type 2)

The CONNECT (Type 2) statement connects an application process to the identified application server and establishes the rules for application-directed distributed unit of work. This server is then the current server for the process.

Most aspects of a CONNECT (Type 1) statement also apply to a CONNECT (Type 2) statement. Rather than repeating that material here, this section describes only those elements of Type 2 that differ from Type 1.

Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared. When invoked using the command line processor, additional options can be specified.

For more information, refer to "Using command line SQL statements and XQuery statements" in *Command Reference*.

Authorization

CONNECT processing goes through two levels of access control. Both levels must be satisfied for the connection to be successful.

The first level of access control is authentication, where the user ID associated with the connection must be successfully authenticated according to the authentication method set up for the server. At successful authentication, a database authorization ID is derived from the connection user ID according to the authentication plug-in in effect for the server. This database authorization ID must then pass the second

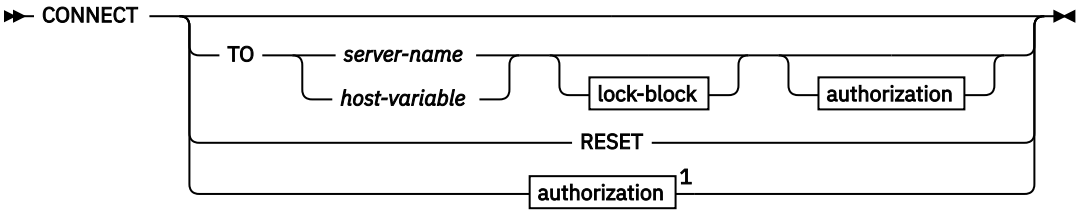
level of access control for the connection, that is, authorization. To do so, this authorization ID must hold at least one of the following authorities:

- CONNECT authority
- SECADM authority
- DBADM authority
- SYSADM authority
- SYSCTRL authority
- SYSMAINT authority
- SYSMON authority

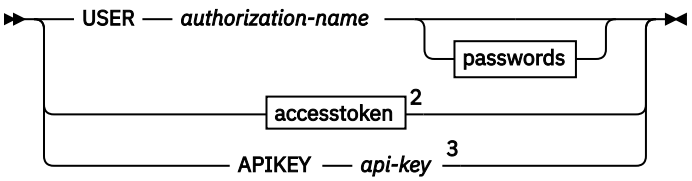
Note: For a partitioned database, the user and group definitions must be identical across all database partitions.

Syntax

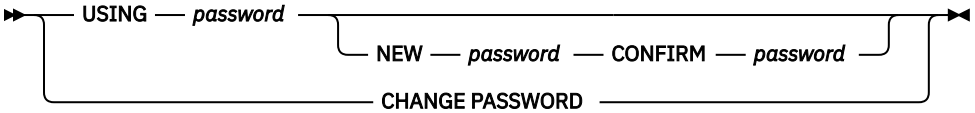
The selection between Type 1 and Type 2 is determined by precompiler options. For an overview of these options, see "Connecting to distributed relational databases".



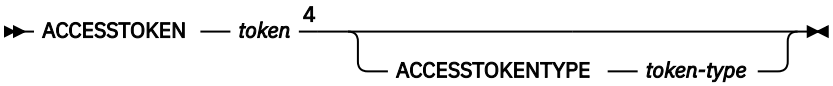
authorization



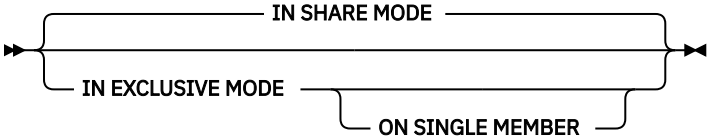
passwords



accesstoken



lock-block



Notes:

[1] This form is only valid if implicit connect is enabled.
[2] This feature is available starting from Db2 Version 11.5 Mod Pack 4.
[3] This feature is available starting from Db2 Version 11.5 Mod Pack 4.
[4] This feature is available starting from Db2 Version 11.5 Mod Pack 4.

Description

TO *server-name/host-variable*

The rules for coding the name of the server are the same as for Type 1.

If the SQLRULES(STD) option is in effect, the *server-name* must not identify an existing connection of the application process, otherwise an error (SQLSTATE 08002) is raised.

If the SQLRULES(DB2) option is in effect and the *server-name* identifies an existing connection of the application process, that connection is made current and the old connection is placed into the dormant state. That is, the effect of the CONNECT statement in this situation is the same as that of a SET CONNECTION statement.

For information about the specification of SQLRULES, see "Options that Govern Distributed Unit of Work Semantics".

Successful Connection

If the CONNECT statement is successful:

- A connection to the application server is either created (or made non-dormant) and placed into the current and held states.
- If the CONNECT TO is directed to a different server than the current server, then the current connection is placed into the dormant state.
- The CURRENT SERVER special register and the SQLCA are updated in the same way as for CONNECT (Type 1).

Unsuccessful Connection

If the CONNECT statement is unsuccessful:

- No matter what the reason for failure, the connection state of the application process and the states of its connections are unchanged.
- As with an unsuccessful Type 1 CONNECT, the SQLERRP field of the SQLCA is set to the name of the module at the application requester or server that detected the error.

CONNECT (with no operand), IN SHARE/EXCLUSIVE MODE, USER, and USING

If a connection exists, Type 2 behaves like a Type 1. The authorization ID and database alias are placed in the SQLERRMC field of the SQLCA. If a connection does not exist, no attempt to make an implicit connection is made and the SQLERRP and SQLERRMC fields return a blank. (Applications can check if a current connection exists by checking these fields.)

A CONNECT with no operand that includes USER and USING can still connect an application process to a database using the DB2DBDFT environment variable. This method is equivalent to a Type 2 CONNECT RESET, but permits the use of a user ID and password.

RESET

Equivalent to an explicit connect to the default database if it is available. If a default database is not available, the connection state of the application process and the states of its connections are unchanged.

Availability of a default database is determined by installation options, environment variables, and authentication settings.

Rules

- As outlined in "Options that Govern Distributed Unit of Work Semantics", a set of connection options governs the semantics of connection management. Default values are assigned to every preprocessed source file. An application can consist of multiple source files precompiled with different connection options.

  Unless a SET CLIENT command or API has been executed first, the connection options used when preprocessing the source file containing the first SQL statement executed at run time become the effective connection options.

If a CONNECT statement from a source file preprocessed with different connection options is subsequently executed without the execution of any intervening SET CLIENT command or API, an error (SQLSTATE 08001) is returned. Note that once a SET CLIENT command or API has been executed, the connection options used when preprocessing all source files in the application are ignored.

Example 1 in the "Examples" section of this statement illustrates these rules.

- Although the CONNECT statement can be used to establish or switch connections, CONNECT with the USER/USING clause will only be accepted when there is no current or dormant connection to the named server. The connection must be released before issuing a connection to the same server with the USER/USING clause, otherwise it will be rejected (SQLSTATE 51022). Release the connection by issuing a DISCONNECT statement or a RELEASE statement followed by a COMMIT statement.

Comparing Type 1 and Type 2 CONNECT Statements

The semantics of the CONNECT statement are determined by the CONNECT precompiler option or the SET CLIENT API (see "Options that Govern Distributed Unit of Work Semantics"). CONNECT Type 1 or CONNECT Type 2 can be specified and the CONNECT statements in those programs are known as Type 1 and Type 2 CONNECT statements, respectively. Their semantics are described in the following tables:

Use of **CONNECT**:

Type 1	Type 2
Each unit of work can only establish connection to one application server.	Each unit of work can establish connection to multiple application servers.
The current unit of work must be committed or rolled back before allowing a connection to another application server.	The current unit of work need not be committed or rolled back before connecting to another application server.
The CONNECT statement establishes the current connection. Subsequent SQL requests are forwarded to this connection until changed by another CONNECT.	Same as Type 1 CONNECT if establishing the first connection. If switching to a dormant connection and SQLRULES is set to STD, then the SET CONNECTION statement must be used instead.
Connecting to the current connection is valid and does not change the current connection.	Same as Type 1 CONNECT if the SQLRULES precompiler option is set to 'DB2'. If SQLRULES is set to STD, then the SET CONNECTION statement must be used instead.
Connecting to another application server disconnects the current connection. The new connection becomes the current connection. Only one connection is maintained in a unit of work.	Connecting to another application server puts the current connection into the *dormant state*. The new connection becomes the current connection. Multiple connections can be maintained in a unit of work. If the CONNECT is for an application server on a dormant connection, it becomes the current connection. Connecting to a dormant connection using CONNECT is only allowed if SQLRULES(DB2) was specified. If SQLRULES(STD) was specified, then the SET CONNECTION statement must be used instead.
SET CONNECTION statement is supported for Type 1 connections, but the only valid target is the current connection.	SET CONNECTION statement is supported for Type 2 connections to change the state of a connection from dormant to current.

Use of **CONNECT...USER...USING**:

Type 1	Type 2
Connecting with the USER...USING clauses disconnects the current connection and establishes a new connection with the given authorization name and password.	Connecting with the USER/USING clause will only be accepted when there is no current or dormant connection to the same named server.

Use of **Implicit CONNECT, CONNECT RESET**, and **Disconnecting**:

Type 1	Type 2
CONNECT RESET can be used to disconnect the current connection.	CONNECT RESET is equivalent to connecting to the default application server explicitly if one has been defined in the system. Connections can be disconnected by the application at a successful COMMIT. Prior to the commit, use the RELEASE statement to mark a connection as release-pending. All such connections will be disconnected at the next COMMIT. An alternative is to use the precompiler options DISCONNECT(EXPLICIT), DISCONNECT(CONDITIONAL), DISCONNECT(AUTOMATIC), or the DISCONNECT statement instead of the RELEASE statement.
After using CONNECT RESET to disconnect the current connection, if the next SQL statement is not a CONNECT statement, then it will perform an implicit connect to the default application server if one has been defined in the system.	CONNECT RESET is equivalent to an explicit connect to the default application server if one has been defined in the system.
It is an error to issue consecutive CONNECT RESETs.	It is an error to issue consecutive CONNECT RESETs ONLY if SQLRULES(STD) was specified because this option disallows the use of CONNECT to existing connection.
CONNECT RESET implicitly commits the current unit of work.	CONNECT RESET implicitly rolls back the current unit of work.
If an existing connection is disconnected by the system for whatever reasons, then subsequent non-CONNECT SQL statements to this database will receive an SQLSTATE of 08003.	If an existing connection is disconnected by the system, COMMIT, ROLLBACK, and SET CONNECTION statements are still permitted.
The unit of work will be implicitly committed when the application process terminates successfully.	Same as Type 1.
All connections (only one) are disconnected when the application process terminates.	All connections (current, dormant, and those marked for release pending) are disconnected when the application process terminates.

CONNECT Failures:

Type 1	Type 2
Regardless of whether there is a current connection when a CONNECT fails (with an error other than server-name not defined in the local directory), the application process is placed in the unconnected state. Subsequent non-CONNECT statements receive an SQLSTATE of 08003.	If there is a current connection when a CONNECT fails, the current connection is unaffected. If there was no current connection when the CONNECT fails, then the program is then in an unconnected state. Subsequent non-CONNECT statements receive an SQLSTATE of 08003.

Notes

- Implicit connect is supported for the first SQL statement in an application with Type 2 connections. In order to execute SQL statements on the default database, first the CONNECT RESET or the CONNECT USER/USING statement must be used to establish the connection. The CONNECT statement with no operands will display information about the current connection if there is one, but will not connect to the default database if there is no current connection.
- The *authorization-name* SYSTEM cannot be explicitly specified in the CONNECT statement. However, on Windows operating systems, local applications running under the Local System Account can implicitly connect to the database, such that the user ID is SYSTEM.
- When connecting to Windows Server explicitly, the *authorization-name* or user *host-variable* can be specified using the Microsoft Windows Security Account Manager (SAM)-compatible name.
- ***Termination of a connection***: When a connection is terminated and a transaction has not yet been committed or rolled back, see "Use of Implicit CONNECT, CONNECT RESET, and Disconnecting" section for details on what happens to such transactions. To ensure consistent behavior, code an explicit COMMIT statement or ROLLBACK statement instead of depending on the behavior of the CONNECT statement.
- ***Syntax alternatives***: The following are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - DBPARTITIONNUM or NODE can be specified in place of MEMBER, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.

Examples

- *Example 1:* This example illustrates the use of multiple source programs (shown in the boxes), some preprocessed with different connection options (shown in the statement preceding the code), and one of which contains a SET CLIENT API call.

  PGM1: CONNECT(2) SQLRULES(DB2) DISCONNECT(CONDITIONAL)

```
...
exec sql CONNECT TO OTTAWA;
exec sql SELECT col1 INTO :hv1
FROM tbl1;
...
```

  PGM2: CONNECT(2) SQLRULES(STD) DISCONNECT(AUTOMATIC)

```
...
exec sql CONNECT TO QUEBEC;
exec sql SELECT col1 INTO :hv1
FROM tbl2;
...
```

  PGM3: CONNECT(2) SQLRULES(STD) DISCONNECT(EXPLICIT)

```
...
SET CLIENT CONNECT 2  SQLRULES DB2  DISCONNECT EXPLICIT [1]
exec sql CONNECT TO LONDON;
exec sql SELECT col1 INTO :hv1
```

```
    FROM tbl3;
    ...
```

Note:

1. Not the actual syntax of the SET CLIENT API

PGM4: CONNECT(2) SQLRULES(DB2) DISCONNECT(CONDITIONAL)

```
    ...
    exec sql CONNECT TO REGINA;
    exec sql SELECT col1 INTO :hv1
    FROM tbl4;
    ...
```

If the application executes PGM1 then PGM2:

- connect to OTTAWA runs: connect=2, sqlrules=DB2, disconnect=CONDITIONAL
- connect to QUEBEC fails with SQLSTATE 08001 because both SQLRULES and DISCONNECT are different.

If the application executes PGM1 then PGM3:

- connect to OTTAWA runs: connect=2, sqlrules=DB2, disconnect=CONDITIONAL
- connect to LONDON runs: connect=2, sqlrules=DB2, disconnect=EXPLICIT

This is OK because the SET CLIENT API is run before the second CONNECT statement.

If the application executes PGM1 then PGM4:

- connect to OTTAWA runs: connect=2, sqlrules=DB2, disconnect=CONDITIONAL
- connect to REGINA runs: connect=2, sqlrules=DB2, disconnect=CONDITIONAL

This is OK because the preprocessor options for PGM1 are the same as those for PGM4.

- *Example 2:* This example shows the interrelationships of the CONNECT (Type 2), SET CONNECTION, RELEASE, and DISCONNECT statements. S0, S1, S2, and S3 represent four servers.

Sequence	Statement	Current Server	Dormant Connections	Release Pending
0	– No statement	– None	– None	– None
1	– SELECT * FROM TBLA	– S0 (default)	– None	– None
2	– CONNECT TO S1 – SELECT * FROM TBLB	– S1 – S1	– S0 – S0	– None – None
3	– CONNECT TO S2 – UPDATE TBLC SET ...	– S2 – S2	– S0, S1 – S0, S1	– None – None
4	– CONNECT TO S3 – SELECT * FROM TBLD	– S3 – S3	– S0, S1, S2 – S0, S1, S2	– None – None
5	– SET CONNECTION S2	– S2	– S0, S1, S3	– None
6	– RELEASE S3	– S2	– S0, S1	– S3
7	– COMMIT	– S2	– S0, S1	– None
8	– SELECT * FROM TBLE	– S2	– S0, S1	– None

Sequence	Statement	Current Server	Dormant Connections	Release Pending
9	– DISCONNECT S1	– S2	– S0	– None
	– SELECT * FROM TBLF	– S2	– S0	– None

- *Example 3:* Connect to the SAMPLE database using a JWT access token.

```
connect to sample accesstoken <access_token> accesstokentype jwt

   Database Connection Information

 Database server        = DB2/LINUXX8664 11.5.4.0
 SQL authorization ID   = NEWTON
 Local database alias   = SAMPLE
```

CREATE ALIAS

The CREATE ALIAS statement defines an alias for a module, nickname, sequence, table, view, or another alias. Aliases are also known as synonyms.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the alias does not exist
- SCHEMAADM authority on the schema if the schema name of the alias refers to an existing schema
- CREATEIN privilege on the schema, if the schema name of the alias refers to an existing schema, or CREATEIN privilege on SYSPUBLIC, if a public alias is being created
- DBADM authority

Privileges required to use the referenced object through its alias are identical to the privileges required to use the object directly.

To replace an existing alias, the authorization ID of the statement must be the owner of the existing alias (SQLSTATE 42501).

Syntax

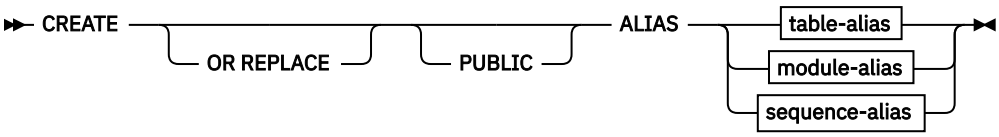
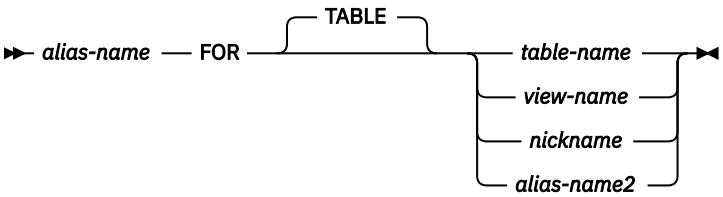
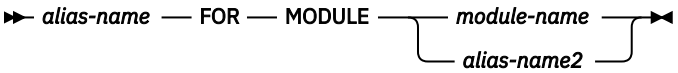


table-alias



module-alias



sequence-alias

Description

OR REPLACE
Specifies to replace the definition for the alias if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog. This option is ignored if a definition for the alias does not exist at the current server. This option can be specified only by the owner of the object.

PUBLIC
Specifies that the alias is an object in the system schema SYSPUBLIC.

alias-name
Names the alias. For a table alias, the name must not identify a nickname, table, view, or table alias that exists at the current server. For a module alias, the name must not identify a module or module alias that exists at the current server. For a sequence alias, the name must not identify a sequence or sequence alias that exists at the current server.

If a two-part name is specified, the schema name cannot begin with 'SYS' (SQLSTATE 42939) except if PUBLIC is specified, then the schema name must be SYSPUBLIC (SQLSTATE 428EK).

FOR TABLE *table-name, view-name, nickname,* or *alias-name2*
Identifies the table, view, nickname, or table alias for which *alias-name* is defined. If another alias name is supplied (*alias-name2*), then it must not be the same as the new *alias-name* being defined (in its fully-qualified form). The *table-name* cannot be a declared temporary table (SQLSTATE 42995).

FOR MODULE *module-name*, or *alias-name2*
Identifies the module or module alias for which *alias-name* is defined. If another alias name is supplied (*alias-name2*), then it must not be the same as the new alias-name being defined (in its fully-qualified form).

FOR SEQUENCE *sequence-name*, or *alias-name2*
Identifies the sequence or sequence alias for which *alias-name* is defined. If another alias name is supplied (*alias-name2*), then it must not be the same as the new *alias-name* being defined (in its fully-qualified form). The *sequence-name* must not be a sequence generated by the system for an identity column (SQLSTATE 428FB).

Notes

- The keyword PUBLIC is used to create a public alias (also known as a public synonym). If the keyword PUBLIC is not used, the type of alias is a private alias (also known as a private synonym).
- Public aliases can be used only in SQL statements and with the LOAD utility.
- The definition of the newly created table alias is stored in SYSCAT.TABLES. The definition of the newly created module alias is stored in SYSCAT.MODULES. The definition of the newly created sequence alias is stored in SYSCAT.SEQUENCES.
- An alias can be defined for an object that does not exist at the time of the definition. If it does not exist, a warning is issued (SQLSTATE 01522). However, the referenced object must exist when a SQL statement containing the alias is compiled, otherwise an error is issued (SQLSTATE 52004).
- An alias can be defined to refer to another alias as part of an alias chain but this chain is subject to the same restrictions as a single alias when used in an SQL statement. An alias chain is resolved in the same way as a single alias. If an alias used in a statement in a package, an SQL routine, a trigger, the default

expression for a global variable, or a view definition points to an alias chain, then a dependency is recorded for the package, SQL routine, trigger, global variable, or view on each alias in the chain. An alias cannot refer to itself in an alias chain and such a cycle is detected at alias definition time (SQLSTATE 42916).

- **Resolving an unqualified alias name:** When resolving an unqualified name, private aliases are considered before public aliases.
- **Conservative binding for public aliases:** If a public alias is used in a statement in a package, an SQL routine, a trigger, the default expression for a global variable, or a view definition, the public alias will continue to be used by these objects regardless of what other object with the same name is created subsequently.
- Creating an alias with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products.
  - SYNONYM can be specified in place of ALIAS

Examples

- *Example 1:* HEDGES attempts to create an alias for a table T1 (both unqualified).

```
CREATE ALIAS A1 FOR T1
```

  The alias HEDGES.A1 is created for HEDGES.T1.

- *Example 2:* HEDGES attempts to create an alias for a table (both qualified).

```
CREATE ALIAS HEDGES.A1 FOR MCKNIGHT.T1
```

  The alias HEDGES.A1 is created for MCKNIGHT.T1.

- *Example 3:* HEDGES attempts to create an alias for a table (alias in a different schema; HEDGES is not a DBADM; HEDGES does not have CREATEIN on schema MCKNIGHT).

```
CREATE ALIAS  MCKNIGHT.A1 FOR MCKNIGHT.T1
```

  This example fails (SQLSTATE 42501).

- *Example 4:* HEDGES attempts to create an alias for an undefined table (both qualified; FUZZY.WUZZY does not exist).

```
CREATE ALIAS HEDGES.A1 FOR FUZZY.WUZZY
```

  This statement succeeds but with a warning (SQLSTATE 01522).

- *Example 5:* HEDGES attempts to create an alias for an alias (both qualified).

```
CREATE ALIAS HEDGES.A1 FOR MCKNIGHT.T1
CREATE ALIAS HEDGES.A2 FOR HEDGES.A1
```

  The first statement succeeds (as per example 2).

  The second statement succeeds and an alias chain is created, consisting of HEDGES.A2 which refers to HEDGES.A1 which refers to MCKNIGHT.T1. Note that it does not matter whether or not HEDGES has any privileges on MCKNIGHT.T1. The alias is created regardless of the table privileges.

- *Example 6:* Designate A1 as an alias for the nickname FUZZYBEAR.

```
CREATE ALIAS A1 FOR FUZZYBEAR
```

- *Example 7:* A large organization has a finance department numbered D108 and a personnel department numbered D577. D108 keeps certain information in a table that resides at a Db2 RDBMS. D577 keeps

certain records in a table that resides at an Oracle RDBMS. A DBA defines the two RDBMSs as data sources within a federated system, and gives the tables the nicknames of DEPTD108 and DEPTD577, respectively. A federated system user needs to create joins between these tables, but would like to reference them by names that are more meaningful than their alphanumeric nicknames. So the user defines FINANCE as an alias for DEPTD108 and PERSONNEL as an alias for DEPTD577.

```
CREATE ALIAS FINANCE FOR DEPTD108
CREATE ALIAS PERSONNEL FOR DEPTD577
```

- *Example 8:* Create a public alias called TABS for the catalog view SYSCAT.TABLES.

```
CREATE PUBLIC ALIAS TABS FOR SYSCAT.TABLES
```

CREATE AUDIT POLICY

The CREATE AUDIT POLICY statement defines an auditing policy at the current server. The policy determines what categories are to be audited; it can then be applied to other database objects to determine how the use of those objects is to be audited.

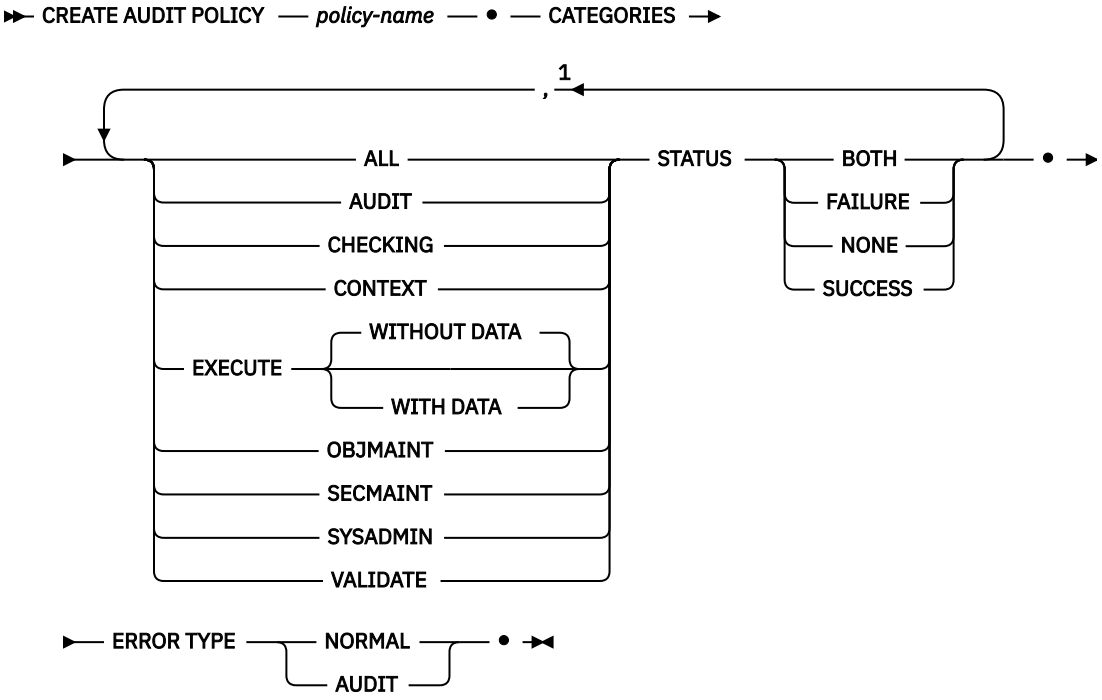
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include SECADM authority.

Syntax



Notes:

[1] Each category can be specified at most once (SQLSTATE 42614), and no other category can be specified if ALL is specified (SQLSTATE 42601).

Description

policy-name
Names the audit policy. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *policy-name* must not identify an audit policy already described in the catalog (SQLSTATE 42710). The name must not begin with the characters 'SYS' (SQLSTATE 42939).

CATEGORIES
A list of one or more audit categories for which a status is specified. If ALL is not specified, the STATUS of any category that is not explicitly specified is set to NONE.

ALL
Sets all categories to the same status. The EXECUTE category is WITHOUT DATA.

AUDIT
Generates records when audit settings are changed or when the audit log is accessed.

CHECKING
Generates records during authorization checking of attempts to access or manipulate database objects or functions.

CONTEXT
Generates records to show the operation context when a database operation is performed.

EXECUTE
Generates records to show the execution of SQL statements.

WITHOUT DATA or WITH DATA
Specifies whether or not input data values provided for any host variables and parameter markers should be logged as part of the EXECUTE category.

WITHOUT DATA
Input data values provided for any host variables and parameter markers are not logged as part of the EXECUTE category. WITHOUT DATA is the default.

WITH DATA
Input data values provided for any host variables and parameter markers are logged as part of the EXECUTE category. Not all input values are logged; specifically, LOB, LONG, XML, and structured type parameters appear as the null value. Date, time, and timestamp fields are logged in ISO format. The input data values are converted to the database code page before being logged. If code page conversion fails, no errors are returned and the unconverted data is logged.

OBJMAINT
Generates records when data objects are created or dropped.

SECMAINT
Generates records when object privileges, database privileges, or DBADM authority is granted or revoked. Records are also generated when the database manager security configuration parameters **sysadm_group**, **sysctrl_group**, or **sysmaint_group** are modified.

SYSADMIN
Generates records when operations requiring SYSADM, SYSMAINT, or SYSCTRL authority are performed.

VALIDATE
Generates records when users are authenticated or when system security information related to a user is retrieved.

STATUS
Specifies a status for the specified category.

BOTH
Successful and failing events will be audited.

FAILURE
Only failing events will be audited.

SUCCESS
Only successful events will be audited.

NONE
No events in this category will be audited.

ERROR TYPE
Specifies whether audit errors are to be returned or ignored.

NORMAL
Any errors generated by the audit are ignored and only the SQLCODEs for errors associated with the operation being performed are returned to the application.

AUDIT
All errors, including errors occurring within the audit facility itself, are returned to the application.

Rules

- An AUDIT-exclusive SQL statement must be followed by a COMMIT or ROLLBACK statement (SQLSTATE 5U021). AUDIT-exclusive SQL statements are:
  - AUDIT
  - CREATE AUDIT POLICY, ALTER AUDIT POLICY, or DROP (AUDIT POLICY)
  - DROP (ROLE or TRUSTED CONTEXT if it is associated with an audit policy)
- An AUDIT-exclusive SQL statement cannot be issued within a global transaction (SQLSTATE 51041) such as, for example, an XA transaction.

Notes

- Only one uncommitted AUDIT-exclusive SQL statement is allowed at a time across all database partitions. If an uncommitted AUDIT-exclusive SQL statement is executing, subsequent AUDIT-exclusive SQL statements wait until the current AUDIT-exclusive SQL statement commits or rolls back.
- Changes are written to the system catalog, but do not take effect until they are committed, even for the connection that issues the statement.

Example

Create an audit policy to audit successes and failures for the AUDIT and OBJMAINT categories; only failures for the SECMAINT, CHECKING, and VALIDATE categories, and no events for the other categories.

```
CREATE AUDIT POLICY DBAUDPRF
  CATEGORIES AUDIT    STATUS BOTH,
             SECMAINT STATUS FAILURE,
             OBJMAINT STATUS BOTH,
             CHECKING STATUS FAILURE,
             VALIDATE STATUS FAILURE
  ERROR TYPE NORMAL
```

CREATE BUFFERPOOL

The CREATE BUFFERPOOL statement defines a buffer pool at the current server. Buffer pools are defined on members which can access data partitions.

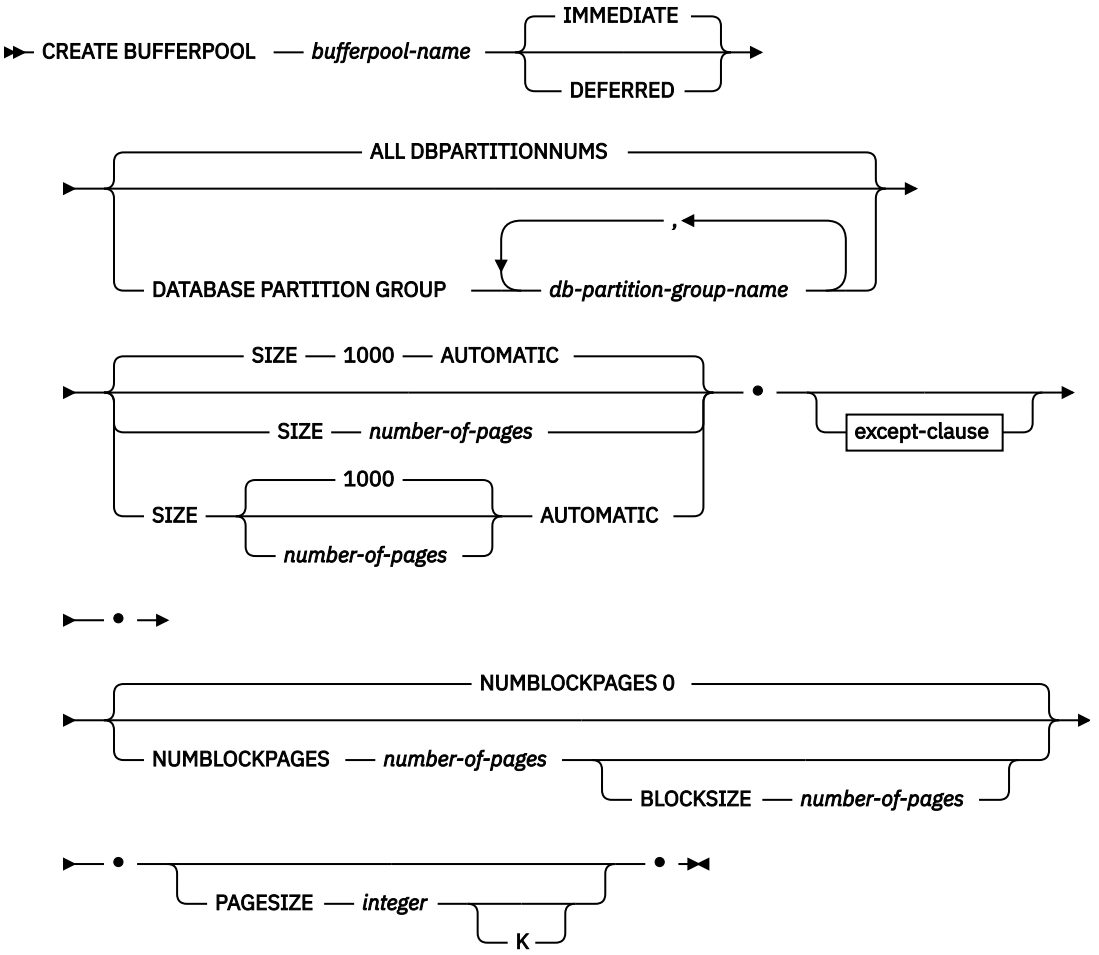
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

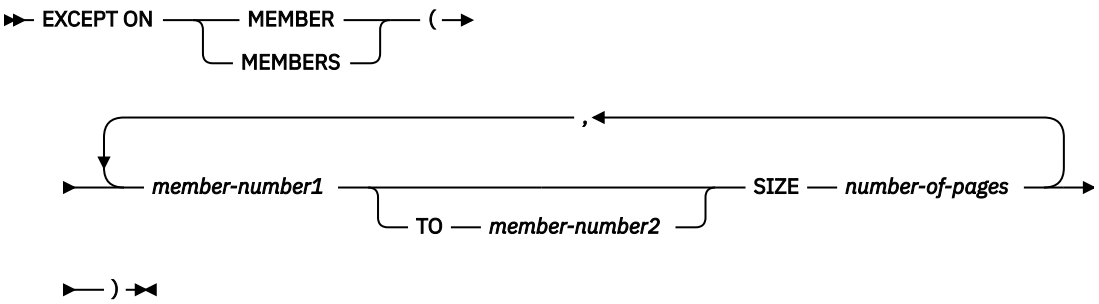
Authorization

The privileges held by the authorization ID of the statement must include SYSCTRL or SYSADM authority.

Syntax



except-clause



Description

bufferpool-name

Names the buffer pool. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *bufferpool-name* must not identify a buffer pool that already exists in the catalog (SQLSTATE 42710). The *bufferpool-name* must not begin with the characters 'SYS' (SQLSTATE 42939).

IMMEDIATE or DEFERRED

Indicates whether or not the buffer pool will be created immediately.

IMMEDIATE
The buffer pool will be created immediately after the CREATE BUFFERPOOL statement is committed. If there is not enough reserved space in the database shared memory to allocate the new buffer pool (SQLSTATE 01657) the statement is executed as DEFERRED.

DEFERRED
The buffer pool will be created when the database is deactivated (all applications need to be disconnected from the database). Reserved memory space is not needed; required memory will be allocated from the system.

ALL DBPARTITIONNUMS or DATABASE PARTITION GROUP
Identifies the members on which the buffer pool is to be defined. The default is ALL DBPARTITIONNUMS.

ALL DBPARTITIONNUMS
This buffer pool will be created on all members which can access all data partitions in the database.

DATABASE PARTITION GROUP ***db-partition-group-name, ...***
Identifies the database partition group or groups to which the buffer pool definition applies. The buffer pool will be created only on members in the specified database partition groups. Each database partition group must exist in the database (SQLSTATE 42704).

SIZE
Specifies the size of the buffer pool. This size will be the default size for all members on which the buffer pool exists. The default is 1000 pages.

number-of-pages
The number of pages for the new buffer pool. The minimum number of pages is 2 and the maximum is architecture-dependent (SQLSTATE 42615).

AUTOMATIC
Enables self tuning for this buffer pool. The database manager adjusts the size of the buffer pool in response to workload requirements. The implicit or explicit number of pages that are specified is used as the initial size of the buffer pool. On subsequent database activations, the buffer pool size is based on the last tuning value that is determined by the self-tuning memory manager (STMM). The STMM enforces a minimum size for automatic buffer pools, which is the minimum of the current size and 5000 pages. To determine the current size of buffer pools that are enabled for self tuning, use the MON_GET_BUFFERPOOL routine and examine the current size of the buffer pools. The size of the buffer pool is found in the **bp_cur_buffsz** monitor element.

NUMBLOCKPAGES ***number-of-pages***
Specifies the number of pages that should exist in the block-based area. The number of pages must not be greater than 98 percent of the number of pages for the buffer pool (SQLSTATE 54052). Specifying the value 0 disables block I/O. The actual value of NUMBLOCKPAGES used will be a multiple of BLOCKSIZE.

NUMBLOCKPAGES is not supported in a Db2 pureScale environment (SQLSTATE 56038).

BLOCKSIZE ***number-of-pages***
Specifies the number of pages in a block. The block size must be a value between 2 and 256 (SQLSTATE 54053). The default value is 32.

BLOCKSIZE is not supported in a Db2 pureScale environment (SQLSTATE 56038).

EXCEPT ON MEMBER or EXCEPT ON MEMBERS
Specifies the member or members for which the size of the buffer pool will be different than the default specified for the database partition group to which the member has access. If this clause is not specified, all members that can access the data partitions in the specified database partition group will have the same size as specified for this buffer pool.

member-number1
Specifies a member number for a member that has access to a data partition for which the buffer pool is created (SQLSTATE 42729).

TO *member-number2*
Specifies a range of member numbers. The value of *member-number2* must be greater than or equal to the value of *member-number1* (SQLSTATE 428A9). Each member identified by the member number range inclusive must have access to the data partition for which the buffer pool is created (SQLSTATE 428A9).

SIZE *number-of-pages*
The size of the buffer pool specified as the number of pages. The minimum number of pages is 2 and the maximum is architecture-dependent (SQLSTATE 42615).

PAGESIZE *integer* [K]
Defines the size of pages used for the buffer pool. The valid values for *integer* without the suffix K are 4096, 8192, 16384, or 32768. The valid values for *integer* with the suffix K are 4, 8, 16, or 32. Any number of spaces is allowed between *integer* and K, including no space. If the page size is not one of these values, an error is returned (SQLSTATE 428DE).

The default value is provided by the **pagesize** database configuration parameter, which is set when the database is created.

Notes

- If the buffer pool is created using the DEFERRED option, any table space created in this buffer pool will use a small system buffer pool of the same page size, until next database activation. The database has to be restarted for the buffer pool to become active and for table space assignments to the new buffer pool to take effect. The default option is IMMEDIATE.
- There should be enough real memory on the machine for the total of all the buffer pools, as well as for the rest of the database manager and application requirements. If the database is unable to obtain memory for the regular buffer pools, it will attempt to start with small system buffer pools, one for each page size (4K, 8K, 16K and 32K). In this situation, a warning will be returned to the user (SQLSTATE 01626), and the pages from all table spaces will use the system buffer pools.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - NODEGROUP can be specified in place of DATABASE PARTITION GROUP
  - DBPARTITIONNUM or NODE can be specified in place of MEMBER, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON
  - DBPARTITIONNUMS or NODES can be specified in place of MEMBERS, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON

CREATE DATABASE PARTITION GROUP

The CREATE DATABASE PARTITION GROUP statement defines a new database partition group within the database, assigns database partitions to the database partition group, and records the database partition group definition in the system catalog.

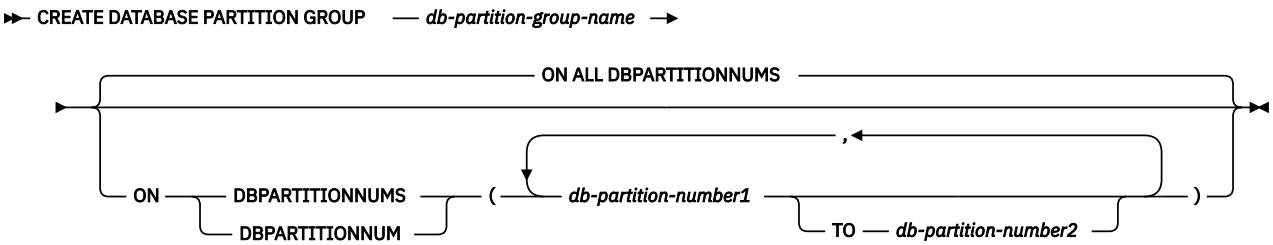
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include SYSCTRL or SYSADM authority.

Syntax



Description

db-partition-group-name
Names the database partition group. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *db-partition-group-name* must not identify a database partition group that already exists in the catalog (SQLSTATE 42710). The *db-partition-group-name* must not begin with the characters 'SYS' or 'IBM' (SQLSTATE 42939).

ON ALL DBPARTITIONNUMS
Specifies that the database partition group is defined over all database partitions defined to the database (db2nodes.cfg file) at the time the database partition group is created.

If a database partition is added to the database system, the ALTER DATABASE PARTITION GROUP statement should be issued to include this new database partition in a database partition group (including IBMDEFAULTGROUP). Furthermore, the REDISTRIBUTE DATABASE PARTITION GROUP command must be issued to move data to the database partition.

ON DBPARTITIONNUMS
Specifies the database partitions that are in the database partition group. DBPARTITIONNUM is a synonym for DBPARTITIONNUMS.

db-partition-number1
Specify a database partition number. (A *node-name* of the form NODE*nnnnn* can be specified for compatibility with the previous version.)

TO *db-partition-number2*
Specify a range of database partition numbers. The value of *db-partition-number2* must be greater than or equal to the value of *db-partition-number1* (SQLSTATE 428A9). All database partitions between and including the specified database partition numbers are included in the database partition group.

Rules

- Each database partition specified by number must be defined in the db2nodes.cfg file (SQLSTATE 42729).
- Each *db-partition-number* listed in the ON DBPARTITIONNUMS clause can appear only once (SQLSTATE 42728).
- A valid *db-partition-number* is between 0 and 999 inclusive (SQLSTATE 42729).
- The CREATE DATABASE PARTITION GROUP statement might fail (SQLSTATE 55071) if an add database partition server request is either pending or in progress. This statement might also fail (SQLSTATE 55077) if a new database partition server is added online to the instance and not all applications are aware of the new database partition server.

Notes

- This statement creates a distribution map for the database partition group. A distribution map identifier (PMAP_ID) is generated for each distribution map. This information is recorded in the catalog and can be retrieved from SYSCAT.DBPARTITIONGROUPS and SYSCAT.PARTITIONMAPS. Each entry in the distribution map specifies the target database partition on which all rows that are hashed reside. For

a single-partition database partition group, the corresponding distribution map has only one entry. For a multiple partition database partition group, the corresponding distribution map has 32768 entries, where the database partition numbers are assigned to the map entries in a round-robin fashion, by default.

- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - NODE can be specified in place of DBPARTITIONNUM
  - NODES can be specified in place of DBPARTITIONNUMS
  - NODEGROUP can be specified in place of DATABASE PARTITION GROUP

Examples

The following examples are based on a partitioned database with six database partitions defined as 0, 1, 2, 5, 7, and 8.

- *Example 1*: Assume that you want to create a database partition group called MAXGROUP on all six database partitions. The statement is as follows:

```
CREATE DATABASE PARTITION GROUP MAXGROUP ON ALL DBPARTITIONNUMS
```

- *Example 2*: Assume that you want to create a database partition group called MEDGROUP on database partitions 0, 1, 2, 5, and 8. The statement is as follows:

```
CREATE DATABASE PARTITION GROUP MEDGROUP
  ON DBPARTITIONNUMS( 0 TO 2, 5, 8)
```

- *Example 3*: Assume that you want to create a single-partition database partition group MINGROUP on database partition 7. The statement is as follows:

```
CREATE DATABASE PARTITION GROUP MINGROUP
  ON DBPARTITIONNUM (7)
```

CREATE EVENT MONITOR

The CREATE EVENT MONITOR statement defines a monitor that records certain events that occur when you use the database. The definition of each event monitor also specifies where the database records the events.

Several different types of event monitors can be created by using this statement. Some of these types are described here, while the remaining types are described separately (see Related links). The types of event monitors that are described separately are as follows:

- Activities. The event monitor records activity events that occur by using the database. The definition of the activities event monitor also specifies where the database records the events.
- Change history. The event monitor records events for changes to configuration parameters, registry variables, and the execution of DDL statements and utilities. The event monitor also records initial configuration and registry values at event monitor startup time.
- Locking. The event monitor records lock-related events that occur by using the database. All records are collected in the unformatted event table.
- Package cache. The event monitor records events that are related to the package cache statement.
- Statistics. The event monitor records statistics events that occur by using the database. The definition of the statistics event monitor also specifies where the database records the events.
- Threshold violations. The event monitor records threshold violation events that occur by using the database. The definition of the threshold violations event monitor also specifies where the database records the events.

- Unit of work. The event monitor records events when a unit of work completes. All records are collected in the unformatted event table.

Invocation

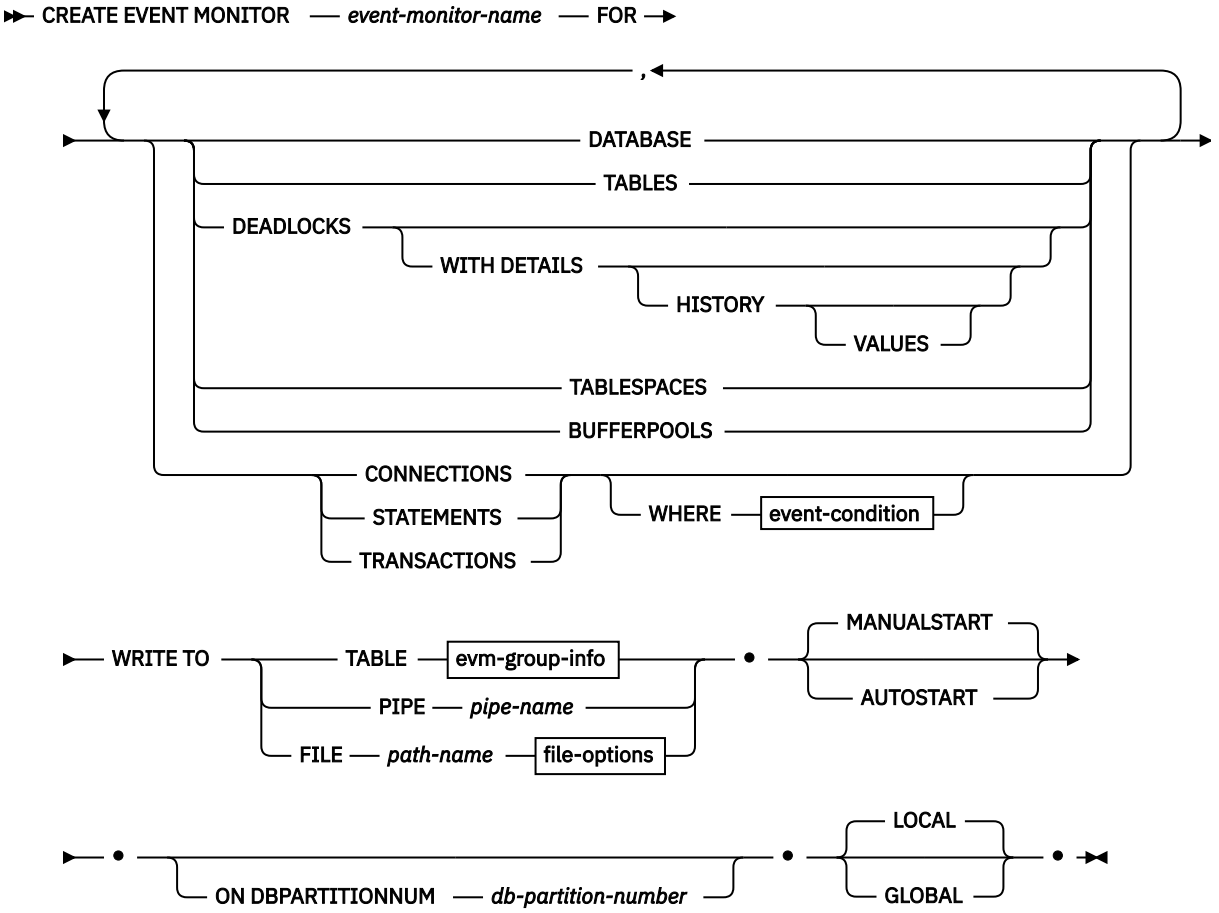
This statement can be embedded in an application program or entered interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

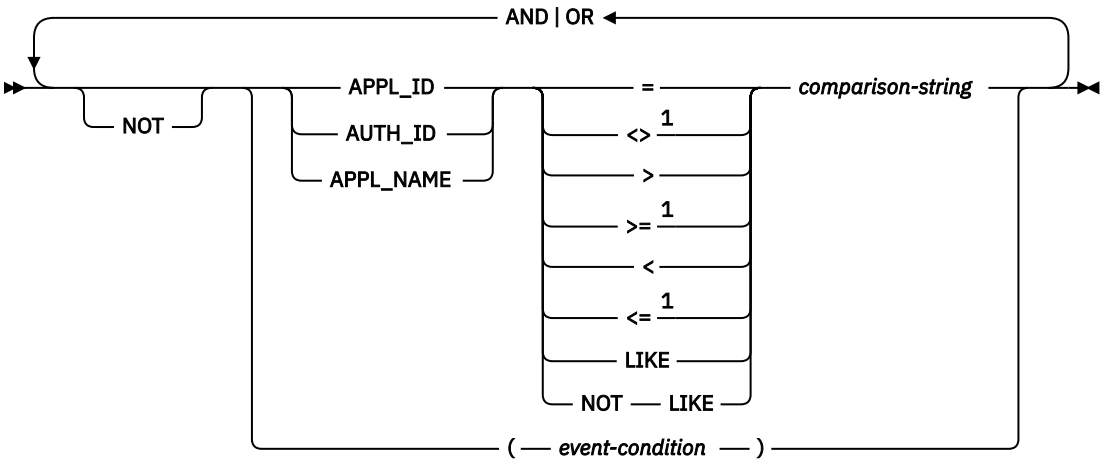
The privileges that are held by the authorization ID of the statement must include one of the following authorities:

- DBADM authority
- SQLADM authority

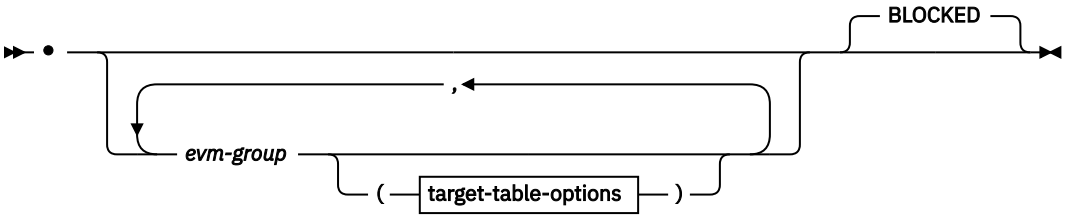
Syntax



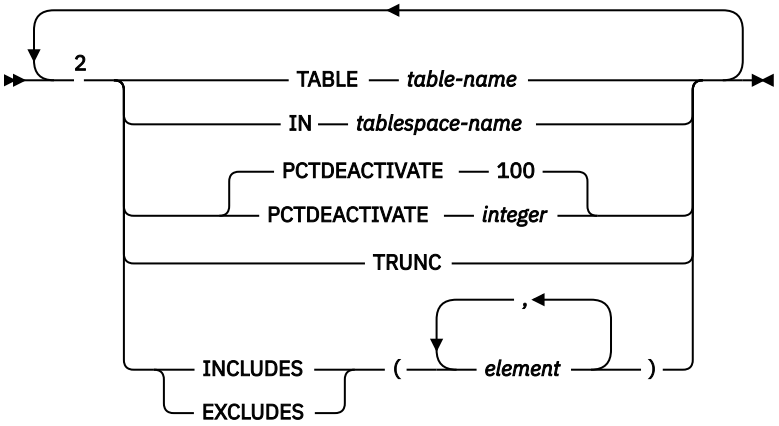
event-condition



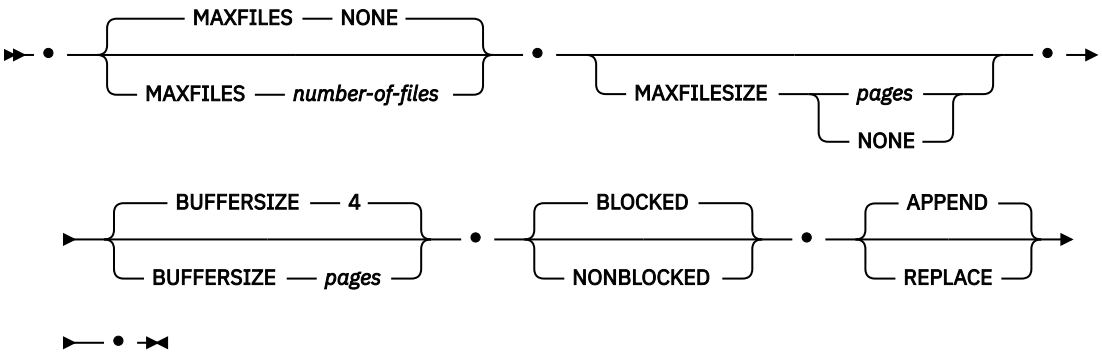
evm-group-info



target-table-options



file-options



Notes:

[1] Other forms of these operators are also supported.
[2] Each clause can be specified once.
[3] Clauses can be separated with a space or a comma.

Description

event-monitor-name
Name of the event monitor is a one-part name. It is an SQL identifier (either ordinary or delimited). The *event-monitor-name* must not identify an event monitor that exists in the catalog (SQLSTATE 42710).

FOR
Introduces the type of event to record.

DATABASE
Specifies that the event monitor records a database event when the last application disconnects from the database.

TABLES
Specifies that the event monitor records a table event for each active table when the last application disconnects from the database. For partitioned tables, a table event is recorded for each data partition of each active table. An active table is a table that changed since the first connection to the database.

DEADLOCKS

Note: This option was deprecated. Its use is no longer recommended and might be removed in a future release. Use the CREATE EVENT MONITOR FOR LOCKING statement to monitor lock-related events, such as lock timeouts, lock waits, and deadlocks.

Specifies that the event monitor records a deadlock event whenever a deadlock occurs.

WITH DETAILS
Specifies that the event monitor is to generate a more detailed deadlock connection event for each application that is involved in a deadlock. This additional detail includes:

- Information about the statement that the application was running when the deadlock occurred, such as the statement text.
- The locks that are held by the application when the deadlock occurred. In a partitioned database environment, includes only those locks that are held on the database partition on which the application was waiting for its lock when the deadlock occurred. For partitioned tables, includes the data partition identifier.

HISTORY
Specifies that the event monitor data also includes:

- The history of all statements in the current unit of work at the participating node (including WITH HOLD cursors that are opened in previous units of work). SELECT statements that are entered at the uncommitted read (UR) isolation level are not included in the statement history.
- The statement compilation environment for each SQL statement in binary format (if available)

VALUES
Specifies that the event monitor data also includes:

- The data values used as input variables for each SQL statement. These data values do not include LOB data, long data, structured type data, or XML data.

Only one of the following commands: DEADLOCKS, DEADLOCKS WITH DETAILS, DEADLOCKS WITH DETAILS HISTORY, or DEADLOCKS WITH DETAILS HISTORY VALUES can be specified in a single CREATE EVENT MONITOR statement (SQLSTATE 42613).

TABLESPACES
Specifies that the event monitor records a table space event for each table space when the last application disconnects from the database.

BUFFERPOOLS
Specifies that the event monitor records a buffer pool event when the last application disconnects from the database.

CONNECTIONS
Specifies that the event monitor records a connection event when an application disconnects from the database.

STATEMENTS
Specifies that the event monitor records a statement event whenever an SQL statement finishes running.

TRANSACTIONS

Note: This option was deprecated. Its use is no longer recommended and might be removed in a future release. Use the CREATE EVENT MONITOR FOR UNIT OF WORK statement to monitor transaction events.

Specifies that the event monitor records a transaction event whenever a transaction completes (that is, whenever there is a commit or rollback operation).

WHERE event-condition
Defines a filter that determines which connections cause a CONNECTION, STATEMENT, or TRANSACTION event to occur. If the result of the event condition is TRUE for a particular connection, then that connection generates the requested events.

This clause is a special form of the WHERE clause that should not be confused with a standard search condition.

To determine whether an application generates events for a particular event monitor, the WHERE clause is evaluated:

- For each active connection when an event monitor is first turned on,
- Then, for each new connection to the database at connect time

The WHERE clause is not evaluated for each event.

If no WHERE clause is specified, all events of the specified event type are monitored.

The event-condition must not exceed 32 678 bytes in the database code page (SQLSTATE 22001).

APPL_ID
Specifies that the application ID of each connection is compared with the *comparison-string* to determine whether the connection generates CONNECTION, STATEMENT, or TRANSACTION events (whichever was specified).

AUTH_ID
Specifies that the authorization ID of each connection is compared with the *comparison-string* to determine whether the connection generates CONNECTION, STATEMENT, or TRANSACTION events (whichever was specified).

APPL_NAME
Specifies that the application program name of each connection is compared with the *comparison-string* to determine whether the connection generates CONNECTION, STATEMENT, or TRANSACTION events (whichever was specified).

The application program name is the first 20 bytes of the application program file name after the last path separator.

comparison-string
A string to be compared with the APPL_ID, AUTH_ID, or APPL_NAME of each application that connects to the database. *comparison-string* must be a string constant (that is, you cannot use host variables and other string expressions).

WRITE TO
Introduces the target for the data.

TABLE
Indicates that the target for the event monitor data is a set of database tables. The event monitor separates the data stream into one or more logical data groups and inserts each group into a separate table. Data for groups with a target table is kept, whereas data for groups not having

a target table is discarded. Each monitor element that is contained within a group is mapped to a table column with the same name. Only elements that have a corresponding table column are inserted into the table. Other elements are discarded.

evm-group-info
Defines the target table for a logical data group. This clause must be specified for each grouping that is to be recorded. However, if no evm-group-info clauses are specified, all groups for the event monitor type are recorded.

For more information about logical data groups, see "Logical data groups and event monitor output tables" in *Database Monitoring Guide and Reference*.

evm-group
Identifies the logical data group for which a target table is being defined. The value depends upon the type of event monitor, as shown in the following table:

Table 130. Values for evm-group based on the type of event monitor

Type of Event Monitor	evm-group value
Database	• DB • CONTROL[1] • DBMEMUSE
Tables	• TABLE • CONTROL[1]
Deadlocks	• CONNHEADER • DEADLOCK • DLCONN • CONTROL[1]
Deadlocks with details	• CONNHEADER • DEADLOCK • DLCONN[2] • DLLOCK[3] • CONTROL[1]
Deadlocks with details history	• CONNHEADER • DEADLOCK • DLCONN[2] • DLLOCK[3] • STMTHIST • CONTROL[1]
Deadlocks with details history values	• CONNHEADER • DEADLOCK • DLCONN[2] • DLLOCK[3] • STMTHIST • STMTVALS • CONTROL[1]

Table 130. Values for evm-group based on the type of event monitor (continued)

Type of Event Monitor	evm-group value
Table spaces	• TABLESPACE • CONTROL[1]
Buffer pools	• BUFFERPOOL • CONTROL[1]
Connections	• CONNHEADER • CONN • CONTROL[1] • CONNMEMUSE
Statements	• CONNHEADER • STMT • SUBSECTION[4] • CONTROL[1]
Transactions	• CONNHEADER • XACT • CONTROL[1]
Activities	• ACTIVITY • ACTIVITYMETRICS • ACTIVITYSTMT • ACTIVITYVALS • CONTROL[1]
Statistics	• QSTATS • SCSTATS • SCMETRICS • WCSTATS • WLSTATS • WLMETRICS • HISTOGRAMBIN • CONTROL[1]
Threshold Violations	• THRESHOLDVIOLATIONS • CONTROL[1]
Locking[5]	• LOCK • LOCK_PARTICIPANTS • LOCK_PARTICIPANT_ACTIVITIES • LOCK_ACTIVITY_VALUES • CONTROL[1]

Table 130. Values for evm-group based on the type of event monitor (continued)

Type of Event Monitor	evm-group value
Package Cache[5]	• PKGCACHE • PKGCACHE_METRICS • CONTROL[1]
Unit of Work[5]	• UOW • UOW_METRICS • UOW_PACKGE_LIST • UOW_EXECUTABLE_LIST • CONTROL[1]
Change History	• CHANGESUMMARY • EVMONSTART • TXNCOMPLETION • DDLSTMTEXEC • DBDBMCFG • REGVAR • UTILSTART • UTILSTOP • UTILPHASE • UTILLOCATION • CONTROL[1]

[1] Logical data groups dbheader (conn_time element only), start, and overflow, are all written to the CONTROL group. The overflow group is written if the event monitor is non-blocked and events were discarded.

[2] Corresponds to the DETAILED_DLCONN event.

[3] Corresponds to the LOCK logical data groups that occur within each DETAILED_DLCONN event.

[4] Created only for partitioned database environments.

[5] Refers to the Formatted Event Table version of this event monitor type.

target-table-options

Identifies the target table for the group. If a value for *target-table-options* is not specified, CREATE EVENT MONITOR processing proceeds as follows:

- A derived table name is used (see description for TABLE *table-name*).
- A default table space is chosen (see description for IN *tablespace-name*).
- All elements are included.
- PCTDEACTIVATE and TRUNC are not specified.

TABLE *table-name*

Specifies the name of the target table. The target table must be a nonpartitioned row-organized table. If the name is unqualified, the table schema defaults to

the value in the CURRENT SCHEMA special register. If no name is provided, the unqualified name is derived from *evm-group* and *event-monitor-name* as follows:

```
substring(evm-group CONCAT '_'
  CONCAT event-monitor-name,1,128)
```

IN *tablespace-name*
Defines the table space in which the table is to be created. If no table space name is provided, the table space is chosen by using the same process as when a table is created without a table space name that uses CREATE TABLE.

PCTDEACTIVATE *integer*
If a table for the event monitor is being created in the automatic storage (non-temporary) or DMS table space, the PCTDEACTIVATE parameter specifies how full the table space must be before the event monitor automatically deactivates. The specified value, which represents a percentage, can range 0 - 100, where 100 means that the event monitor deactivates when the table space becomes full. The default value is 100 if PCTDEACTIVATE is not specified. This option is ignored for SMS table spaces.

Important: If the target table space has auto-resize enabled, set PCTDEACTIVATE to 100. Alternatively, omit this clause entirely to have the default of 100 apply. Otherwise, the event monitor might deactivate unexpectedly if the table space reaches the threshold that is specified by PCTDEACTIVTATE before the table space is automatically resized.

It is recommended that, when a target table space has auto-resize enabled, the PCTDEACTIVATE parameter be set to 100.

TRUNC
Specifies that the STMT_TEXT and STMT_VALUE_DATA columns are defined as VARCHAR(*n*), where *n* is the largest size that can fit into the table row. In this case, any data that is longer than *n* bytes is truncated. The following example illustrates how the value of *n* is calculated. Assume that:

- The table is created in a table space that uses 32K pages.
- The total length of all the other columns in the table equals 357 bytes.

In this case, the maximum row size for a table is 32677 bytes. Therefore, the element would be defined as VARCHAR(32316); that is, 32677 - 357 - 4. If TRUNC is not specified, the column is defined as CLOB(2M). STMT_TEXT is found in the STMT event group, the STMT_HISTORY event group, and the DLCONN event group (for deadlocks with details event monitors). STMT_VALUE_DATA is found in the DATA_VALUE event group.

INCLUDES
Specifies that the following elements are to be included in the table.

EXCLUDES
Specifies that the following elements are not to be included in the table.

element
Identifies a monitor element. Element information can be provided in one of the following forms:

- Specify no element information. In this case, all elements are included in the CREATE TABLE statement.
- Specify the elements to include in the form: INCLUDES (element1, element2, ..., element*n*). Only table columns are created for these elements.
- Specify the elements to exclude in the form: EXCLUDES (element1, element2, ..., element*n*). Only table columns are created for all elements except these.

Use the db2evtbl command to build a CREATE EVENT MONITOR statement that includes a complete list of elements for a group.

BLOCKED
Specifies that each agent that generates an event must wait for an event buffer to be written out to disk if the agent determines that both event buffers are full. Select BLOCKED to prevent event data loss. It is the default option.

PIPE
Specifies that the target for the event monitor data is a named pipe. The event monitor writes the data to the pipe in a single stream (that is, as if it were a single, infinitely long file). Writing the data to a pipe, an event monitor does not perform blocked writes. If the pipe buffer has no room, then the event monitor discards the data. It is the monitoring application's responsibility to read the data promptly if it wants to ensure no data loss.

pipe-name
The name of the pipe (FIFO on AIX) to which the event monitor writes the data.

The naming rules for pipes are platform-specific.

Operating system	Naming rules
AIX	Pipe names are treated like file names. As a result, relative pipe names are permitted, and are treated like relative path-names (see description for *path-name*).
Linux	Pipe names are treated like file names. As a result, relative pipe names are permitted, and are treated like relative path-names (see description for *path-name*).
Windows	There is special syntax for a pipe name and as a result, absolute pipe names are required.

The existence of the pipe is not checked at event monitor creation time. It is the responsibility of the monitoring application to create and open the pipe for reading at the time that the event monitor is activated. If the pipe is not available now, then the event monitor turns itself off and logs an error. (That is, if the event monitor was activated at database start time as a result of the AUTOSTART option, then the event monitor logs an error in the system error log). If the event monitor is activated by the SET EVENT MONITOR STATE SQL statement, then that statement fails (SQLSTATE 58030).

In a Db2 pureScale environment, the *pipe-name* must be on a shared file system whether the event monitor is LOCAL or GLOBAL. This requirement is to allow this event monitors to operate correctly if a member failover occurs. Failure to use a *pipe-name* on a shared file system results in an error (SQLSTATE 428A3) if the event monitor activates during a member failover.

FILE
Indicates that the target for the event monitor data is a file (or set of files). The event monitor writes out the stream of data as a series of eight character numbered files, with the extension "evt". (For example, `00000000.evt`, `00000001.evt`, and `00000002.evt`). The data should be considered to be one logical file even though the data is broken up into smaller pieces (that is, the start of the data stream is the first byte in the file`00000000.evt`; the end of the data stream is the last byte in the file `nnnnnnnn.evt`).

The maximum size of each file and the maximum number of files can be defined. An event monitor does not split a single event record across two files. However, an event monitor might write related records in two different files. It is the responsibility of the application that uses this data to track related information when it is processing the event files.

path-name
The name of the directory in which the event monitor writes the event files data. The path must be known at the server; however, the path itself might reside on another database partition (for example, an NFS-mounted file). A string constant must be used to specify the *path-name*.

The directory does not have to exist at CREATE EVENT MONITOR time. However, a check is made for the existence of the target path when the event monitor is activated. Then, if the target path does not exist, an error (SQLSTATE 428A3) is raised.

If an absolute path is specified, the specified path is the one used.

In environments other than Db2 pureScale, if a relative path (a path that does not start with the root) is specified, then the path relative to the DB2EVENT directory in the database directory is used. In a Db2 pureScale environment, if a relative path is specified, then the path relative to the directory owning the database in the database directory is used.

It is possible to specify two or more event monitors that have the same target path. However, when one of the event monitors is activated for the first time, and if the target directory is not empty, it is impossible to activate any of the other event monitors.

In a Db2 pureScale environment, the *path-name* must be on a shared file system whether this event monitor is LOCAL or GLOBAL. This requirement is to allow event monitors to operate correctly if a member failover occurs. Failure to use a *path-name* on a shared file system results in an error (SQLSTATE 428A3) if the event monitor activates during a member failover.

file-options
Specifies the options for the file format.

MAXFILES NONE
Specifies that the event monitor can create any number of event files. This is the default.

MAXFILES *number-of-files*
Specifies that the number of event monitor files that exist for a particular event monitor at any time is limited. Whenever an event monitor must create another file, it checks to make sure that the number of .evt files in the directory is less than *number-of-files*. If this limit is already reached, then the event monitor turns itself off.

If an application removes the event files from the directory after they are written, then the total number of files that an event monitor can produce can exceed *number-of-files*. This option is provided so that the event data does not use more than a specified amount of disk space.

MAXFILESIZE *pages*
Specifies that the size of each event monitor file has a limit. Whenever an event monitor writes a new event record to a file. It checks that the file does not grow to be greater than *pages* (in units of 4K pages). If the resulting file would be too large, then the event monitor switches to the next file. The default for this option is:

- Linux - 1000 4K pages
- UNIX - 1000 4K pages
- Windows - 200 4K pages

The number of pages must be greater than at least the size of the event buffer in pages. If this requirement is not met, then an error (SQLSTATE 428A4) is raised.

MAXFILESIZE NONE
Specifies that a file's size has no set limit. If MAXFILESIZE NONE is specified, then MAXFILES 1 must also be specified. This option means that one file contains all of the event data for a particular event monitor. In this case, the only event file is `00000000.evt`.

BUFFERSIZE *pages*
Specifies the size of the event monitor buffers (in units of 4K pages). All event monitor file I/O is buffered to improve the performance of the event monitors. The larger the buffers, the less I/O is performed by the event monitor. Highly active event monitors have larger buffers than relatively inactive event monitors. When the monitor is started, two buffers of the specified size are allocated. Event monitors use double buffering to permit asynchronous I/O.

The default size of each buffer is four pages (two 16K buffers are allocated). The minimum size is one page. The maximum size of the buffers is limited by the value of the MAXFILESIZE parameter and the size of the monitor heap because the buffers are allocated from that heap. If many event monitors are being used at the same time, increase the size of the **mon_heap_sz** database manager configuration parameter.

Event monitors that write their data to a pipe also have two internal (non-configurable) buffers that are each one page in size. These buffers are also allocated from the monitor heap (MON_HEAP). For each active event monitor that has a pipe target, increase the size of the database heap by two pages.

BLOCKED
Specifies that each agent that generates an event must wait for an event buffer to be written out to disk if the agent determines that both event buffers are full. BLOCKED must be selected to prevent event data loss. This is the default option.

NONBLOCKED
Specifies that each agent that generates an event must not wait for the event buffer to be written out to disk if the agent determines that both event buffers are full. NONBLOCKED event monitors do not slow down database operations to the extent of BLOCKED event monitors. However, NONBLOCKED event monitors are subject to data loss on highly active systems.

APPEND
Specifies that if event data files exist when the event monitor is turned on, then the event monitor appends the new event data to the existing stream of data files. When the event monitor is reactivated, it resumes writing to the event files from when it was turned off. APPEND is the default option.

The APPEND option does not apply at CREATE EVENT MONITOR time, if there exists event data in the directory where the newly created event monitor is to write its event data.

REPLACE
Specifies that if event data files exist when the event monitor is turned on, then the event monitor erases all of the event files and start writing data to file `00000000.evt`.

MANUALSTART
Specifies that the event monitor must be activated manually by using the SET EVENT MONITOR STATE statement. After a MANUALSTART event monitor is activated, it can be deactivated only by using the SET EVENT MONITOR STATE statement or by stopping the instance. This is the default.

AUTOSTART
Specifies that the event monitor is to be automatically activated whenever the database partition on which the event monitor runs is activated.

ON DBPARTITIONNUM *db-partition-number*
Specifies the database partition (in a partitioned database environment) or member (in a Db2 pureScale environment) on which a file or pipe event monitor is to run. When the monitoring scope is defined as LOCAL, data is collected only on the specified partition or member. When the monitoring scope is defined as GLOBAL, all database partitions or members collect data and report to the database partition or member with the specified number. The I/O component physically runs on the specified database partition or member, write records to the specified file or pipe.

When Db2 pureScale is enabled, -1 can be specified, which allows the I/O component to run from any active member. Additionally, if the I/O component is no longer able to run on member, the event monitor is restarted with the I/O component that is activated on another available active member.

This clause is not valid for table event monitors. In a partitioned database environment, write-to-table event monitors runs and writes events on all database partitions where table spaces for target tables are defined.

In a Db2 pureScale environment, write-to-table event monitors record events on all active members.

If this clause is not specified and Db2 pureScale is not enabled, the currently connected database partition number (for the application) is used.

If this clause is not specified and Db2 pureScale is enabled, the I/O component is able to run on any currently connected database partition number.

LOCAL
The event monitor reports only on the database partition that is running. It gives a partial trace of the database activity. This is the default.

This clause is valid for file or pipe monitors. It is not valid for table event monitors.

GLOBAL
The event monitor reports on all database partitions. For a partitioned database, only DEADLOCKS event monitors can be defined as GLOBAL.

This clause is valid for file or pipe monitors. It is not valid for table event monitors.

Rules

- Each of the event types (DATABASE, TABLES, DEADLOCKs, ...) can be specified only in a particular event monitor definition.

Notes

- Event monitor definitions are recorded in the SYSCAT.EVENTMONITORS catalog view. The events themselves are recorded in the SYSCAT.EVENTS catalog view. The names of target tables are recorded in the SYSCAT.EVENTTABLES catalog view.
- A performance impact occurs by using DEADLOCKS WITH DETAILS rather than DEADLOCKS. When a deadlock occurs, the database manager requires extra time to record the extra deadlock information.
- A CONNHEADER event is normally written whenever a connection is established. However, if an event monitor is created only for DEADLOCKS WITH DETAILS, a CONNHEADER event is written the first time that the connection participates in a deadlock.
- In a database with multiple database partitions, the ON DBPARTITIONNUM clause can be used with FILE and PIPE event monitors having a DEADLOCKS event type to indicate where the event monitor itself resides. Information from other database partitions, if relevant, is sent to that location for processing.
- In a database with multiple database partitions, a deadlock event monitor receives information about applications that have locks in the deadlock from all the database partitions on which those locks existed. If the database partition to which the application is connected (the application coordinator partition) is not one of the participating database partitions, no information about a deadlock event is received from that database partition.
- The BUFFERSIZE parameter restricts the size of STMT, STMT_HISTORY, DATA_VALUE, and DETAILED_DLCONN events. If an STMT or a STMT_HISTORY event cannot fit within a buffer, it is truncated by truncating statement text. If a DETAILED_DLCONN event cannot fit within a buffer, it is truncated by removing locks. If it still cannot fit, statement text is truncated. If a DATA_VAL event cannot fit within a buffer, the data value is truncated.

  Event monitors WITH DETAILS HISTORY VALUES (and to a lesser extent, WITH DETAILS HISTORY) use a significant amount of monitor heap space to track statements and their data values, as described in the **mon_heap_sz** configuration parameter.
- If the database partition on which the event monitor is to run is not active, event monitor activation occurs when that database partition next activates.
- After an event monitor is activated, it behaves like an autostart event monitor until that event monitor is explicitly deactivated or the instance is recycled. That is, if an event monitor is active when a database partition is deactivated, and that database partition is later reactivated, the event monitor is also explicitly reactivated.
- **Write to table event monitors** General notes:
  - All target tables are created when the CREATE EVENT MONITOR statement runs.

– If the creation of a table fails for any reason, an error is passed back to the application program, and the CREATE EVENT MONITOR statement fails.
– A target table can be used by one event monitor only. During CREATE EVENT MONITOR processing, if a target table is found to be defined for use by another event monitor, the CREATE EVENT MONITOR statement fails and an error is passed back to the application program. A table is defined for use by another event monitor if the table name matches a value found in the SYSCAT.EVENTTABLES catalog view.
– During CREATE EVENT MONITOR processing, if a table exists, but is not defined for use by another event monitor, no table is created, and processing continues. A warning is passed back to the application program.
– Any table spaces must exist before the CREATE EVENT MONITOR statement runs. The CREATE EVENT MONITOR statement does not create table spaces.
– If specified, the LOCAL and GLOBAL keywords are ignored. With WRITE TO TABLE event monitors, an event monitor output process or thread is started on each database partition in the instance, and each of these processes reports data only for the database partition on which it is running.
– The following event types from the flat monitor log file or pipe format are not recorded by writing to table event monitors:
  - LOG_STREAM_HEADER
  - LOG_HEADER
  - DB_HEADER (Elements db_name and db_path are not recorded. The element conn_time is recorded in CONTROL).
– In a partitioned database environment, data is only written to target tables on the database partitions where their table spaces exist. If a table space for a target table does not exist on some database partition, data for that target table is ignored. This behavior allows users to choose a subset of database partitions for monitoring, by creating a table space that exists only on certain database partitions.

  In a Db2 pureScale environment, data is written from every member.

  In a partitioned database environment, if some target tables do not reside on a database partition, but other target tables do reside on that database partition, only the data for the target tables that do reside on that database partition is recorded.
– Users must manually prune all target tables.

Table Columns:

– Column names in a table match an event monitor element identifier. Any event monitor element that does not have a corresponding target table column is ignored.
– Use the db2evtbl command to build a CREATE EVENT MONITOR command that includes a complete list of elements for a group.
– The types of columns that are used for monitor elements correlate to the following mapping:

```
SQLM_TYPE_STRING                    CHAR[n], VARCHAR[n] or CLOB(n)
                                    (If the data in the event monitor
                                    record exceeds n bytes,
                                    it is truncated.)
SQLM_TYPE_U8BIT and SQLM_TYPE_8BIT  SMALLINT, INTEGER or BIGINT
SQLM_TYPE_16BIT and SQLM_TYPE_U16BIT SMALLINT, INTEGER or BIGINT
SQLM_TYPE_32BIT and SQLM_TYPE_U32BIT INTEGER or BIGINT
SQLM_TYPE_U64BIT and SQLM_TYPE_64BIT BIGINT
sqlm_timestamp                      TIMESTAMP
sqlm_time(elapsed time)             BIGINT
sqlca:
  sqlerrmc                          VARCHAR[72]
  sqlstate                          CHAR[5]
  sqlwarn                           CHAR[11]
  other fields                      INTEGER or BIGINT
```

– Columns are defined to be NOT NULL.

– Because the performance of tables with CLOB columns is inferior to tables that have VARCHAR columns, consider using the TRUNC keyword to specify the STMT *evm-group* value (or the DLCONN *evm-group* value when using the DEADLOCKS WITH DETAILS event type).
– Unlike other target tables, the columns in the CONTROL table do not match monitor element identifiers. Columns are defined as follows:

Column Name	Data type	Nullable	Description
PARTITION_KEY	INTEGER	N	Distribution key (partitioned database only)
PARTITION_NUMBER	INTEGER	N	Database partition number (partitioned database only)
EVMONNAME	VARCHAR(128)	N	Name of the event monitor
MESSAGE	VARCHAR(128)	N	Describes the nature of the MESSAGE_TIME column. For more information, see "message - Control Table Message monitor element" in the *Database Monitoring Guide and Reference*.
MESSAGE_TIME	TIMESTAMP	N	Timestamp

– In a partitioned database environment, the first column of each table is named PARTITION_KEY, is NOT NULL, and is of type INTEGER. This column is used as the distribution key for the table. The value of this column is chosen so that each event monitor process inserts data into the database partition on which the process is running. That is, insert operations run locally on the database partition where the event monitor process is running. On any database partition, the **PARTITION_KEY** field contains the same value. This means that if a database partition is dropped and data is redistributed, all data on the dropped database partition goes to one other database partition instead of being evenly distributed. Therefore, before you remove a database partition, consider deleting all table rows on that database partition.
– In a partitioned database environment, a column that is named PARTITION_NUMBER can be defined for each table. This column is NOT NULL and is of type INTEGER. It contains the number of the database partition on which the data was inserted. Unlike the PARTITION_KEY column, the PARTITION_NUMBER column is not mandatory. The PARTITION_NUMBER column is not allowed in a nonpartitioned database environment.

Table Attributes:

– Default table attributes are used. Besides, distribution key (partitioned databases only), no extra options are specified when you create tables.
– Indexes on the table can be created.
– Extra table attributes (such as volatile, RI, triggers, constraints) can be added, but the event monitor process (or thread) ignores them.
– If "not logged initially" is added as a table attribute, it is turned off at the first COMMIT, and is not set back on.

Event Monitor Activation:

– When an event monitor activates, all target table names are retrieved from the SYSCAT.EVENTTABLES catalog view.

- In a partitioned database environment, activation processing occurs on every database partition of the instance. On a particular database partition, activation processing determines the table spaces and database partition groups for each target table. The event monitor activates on a database partition if at least one target table exists on that database partition. Moreover, if some target table is not found on a database partition, that target table is flagged so that data that is destined for that table is dropped during runtime processing.
- If a target table does not exist when the event monitor activates (or, in a partitioned database environment, if the table space does not reside on a database partition), activation continues, and data that would otherwise be inserted into this table is ignored.
- Activation processing validates each target table. If validation fails, activation of the event monitor fails, and messages are written to the administration log.
- During activation in a partitioned database environment, the CONTROL table rows for FIRST_CONNECT and EVMON_START are only inserted on the catalog database partition. The table space for the control table must exist on the catalog database partition. If it does not exist on the catalog database partition, these inserts are not inserted.
- In a partitioned database environment, if a partition is not yet active when a write to table event monitor is activated, the event monitor is activated the next time that partition is activated.

Run Time:

- An event monitor runs with DATAACCESS authority.
- While an event monitor is active, an insert operation into a target table fails:
  - Uncommitted changes are rolled back.
  - A message is written to the administration log.
  - The event monitor is deactivated.
- If an event monitor is active, a local COMMIT runs when it finishes processing an event monitor buffer.
- In a partitioned database environment, the actual statement text, which can be up to 2 MB, is only stored (in the STMT or DLCONN table) by the event monitor process on the application coordinator database partition. On other database partitions, this value has zero length.
- In an environment other than a partitioned database or a Db2 pureScale database, all write to table event monitors are deactivated when the last application exits (and the database was not activated explicitly). In a Db2 pureScale environment, write to table event monitors are deactivated on a given member when the database deactivates on that member and reactivates when the database activates on that member again. In a partitioned database environment, write to table event monitors are deactivated when the catalog partition deactivates.
- The DROP EVENT MONITOR statement does not drop target tables.
- Whenever a write-to-table event monitor activates, it acquires IN table locks on each target table to prevent them from being modified while the event monitor is active. Table locks are maintained on all tables while the event monitor is active. If exclusive access is required for any of the target tables (for example, when a utility is to be run), first deactivate the event monitor to release the table locks, and then attempting such access.

- **Syntax alternatives**: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - NODE can be specified in place of DBPARTITIONNUM
  - Commas can be used to separate multiple options in the *target-table-options* clause.

Examples

- *Example 1:* The following example creates an event monitor called SMITHPAY. This event monitor collects event data for the database and for the SQL statements that are run by the PAYROLL application that is owned by the JSMITH authorization ID. The data is appended to the absolute path /home/

jsmith/event/smithpay/. A maximum of 25 files are created. Each file is a maximum of 1 024 4K pages long. The file I/O is non-blocked.

```
CREATE EVENT MONITOR SMITHPAY
  FOR DATABASE, STATEMENTS
  WHERE APPL_NAME = 'PAYROLL' AND AUTH_ID = 'JSMITH'
  WRITE TO FILE '/home/jsmith/event/smithpay'
  MAXFILES 25
  MAXFILESIZE 1024
  NONBLOCKED
  APPEND
```

- *Example 2:* The following example creates an event monitor called DEADLOCKS_EVTS. This event monitor collects deadlock events and writes them to the relative path DLOCKS. One file is written, and the file size has no limit. Each time the event monitor is activated, it appends the event data to the file 00000000.evt if it exists. The event monitor is started each time that the database is started. The I/O is blocked by default.

```
CREATE EVENT MONITOR DEADLOCK_EVTS
  FOR DEADLOCKS
  WRITE TO FILE 'DLOCKS'
  MAXFILES 1
  MAXFILESIZE NONE
  AUTOSTART
```

- *Example 3:* This example creates an event monitor called DB_APPLS. This event monitor collects connection events, and writes the data to the named pipe /home/jsmith/applpipe.

```
CREATE EVENT MONITOR DB_APPLS
  FOR CONNECTIONS
  WRITE TO PIPE '/home/jsmith/applpipe'
```

- *Example 4:* This example, which assumes a partitioned database environment, creates an event monitor called FOO. This event monitor collects SQL statement events and writes them to SQL tables with the following derived names:
  - CONNHEADER_FOO
  - STMT_FOO
  - SUBSECTION_FOO
  - CONTROL_FOO

  Because no table space information is supplied, all tables are created in a table space selected by the system, based on the rules described under the IN *tablespace-name* clause. All tables include all elements for their group (that is, columns are defined whose names are equivalent to the element names.)

```
CREATE EVENT MONITOR FOO
  FOR STATEMENTS
  WRITE TO TABLE
```

- *Example 5:* This example, which assumes a partitioned database environment, creates an event monitor called BAR. This event monitor collects SQL statement and transaction events and writes them to tables as follows:
  - Any data from the STMT group is written to table MYDEPT.MYSTMTINFO. The table is created in table space MYTABLESPACE. Create columns only for the following elements: ROWS_READ, ROWS_WRITTEN, and STMT_TEXT. Any other elements of the group is discarded.
  - Any data from the SUBSECTION group is written to table MYDEPT.MYSUBSECTIONINFO. The table is created in table space MYTABLESPACE. The table includes all columns, except START_TIME, STOP_TIME, and PARTIAL_RECORD.
  - Any data from the XACT group is written to table XACT_BAR. Because no table space information is supplied, the table is created in a table space selected by the system, based on the rules described under the IN *tablespace-name* clause. This table includes all elements that are contained in the XACT group.

– No tables are created for connheader or control, and all data for these groups is discarded.

```
CREATE EVENT MONITOR BAR
  FOR STATEMENTS, TRANSACTIONS
  WRITE TO TABLE
  STMT(TABLE MYDEPT.MYSTMTINFO IN MYTABLESPACE
    INCLUDES(ROWS_READ, ROWS_WRITTEN, STMT_TEXT)),
  STMT(TABLE MYDEPT.MYSTMTINFO IN MYTABLESPACE
    EXCLUDES(START_TIME, STOP_TIME, PARTIAL_RECORD)),
  XACT
```

CREATE EVENT MONITOR (activities)

The CREATE EVENT MONITOR (activities) statement defines a monitor that will record activity events that occur when using the database. The definition of the activity event monitor also specifies where the database should record the events.

Invocation

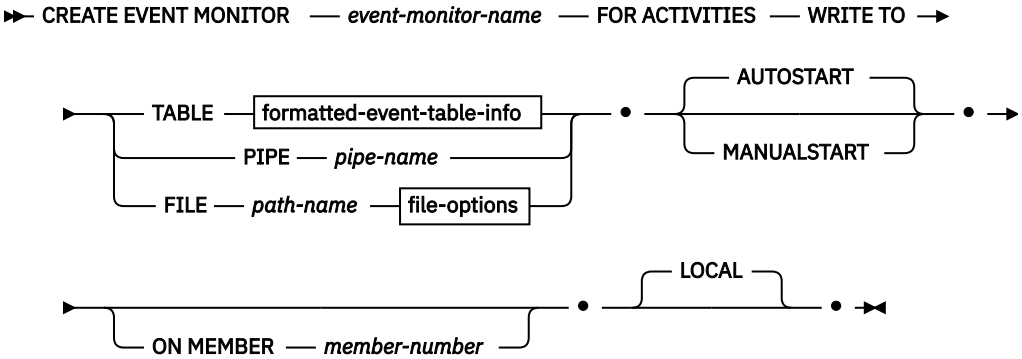
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

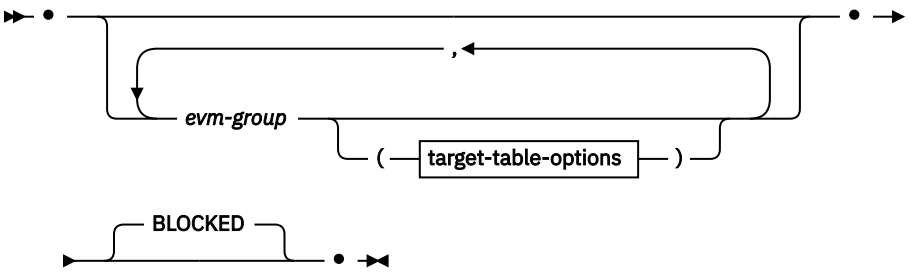
The privileges held by the authorization ID of the statement must include one of the following authorities:

- DBADM authority
- SQLADM authority

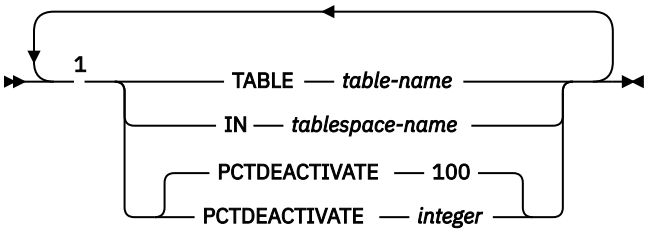
Syntax



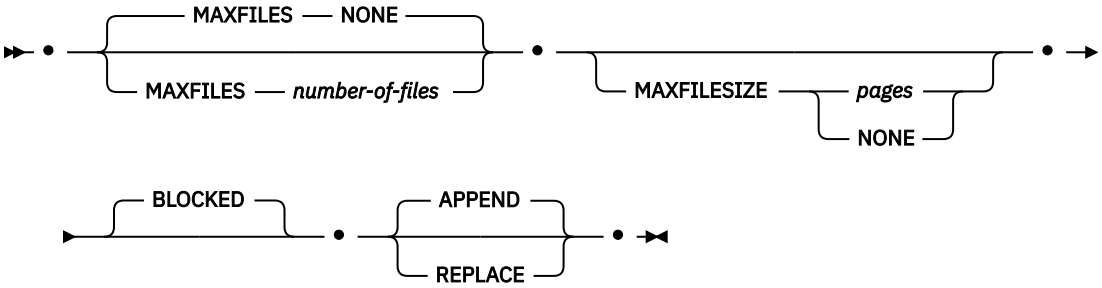
formatted-event-table-info



target-table-options



file-options



Notes:

[1] Each clause can be specified only once.
[2] Clauses can be separated with a space or a comma.

Description

event-monitor-name
Name of the event monitor. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *event-monitor-name* must not identify an event monitor that already exists in the catalog (SQLSTATE 42710).

FOR
Introduces the type of event to record.

ACTIVITIES
Specifies that the event monitor records an activity event when an activity finishes executing, or before the completion of execution if the event is triggered by the WLM_CAPTURE_ACTIVITY_IN_PROGRESS procedure. The activity must either:

- Belong to a service class or workload that has COLLECT ACTIVITY DATA set
- Belong to a work class whose associated work action is COLLECT ACTIVITY DATA
- Be identified as the activity that violated a threshold whose COLLECT ACTIVITY DATA clause was specified
- Have been identified in a call to the WLM_CAPTURE_ACTIVITY_IN_PROGRESS procedure before completing

WRITE TO
Introduces the target for the data.

TABLE
Indicates that the target for the event monitor data is a set of database tables. The event monitor separates the data stream into one or more logical data groups and inserts each group into a separate table. Data for groups having a target table is kept, whereas data for groups not having a target table is discarded. Each monitor element contained within a group is mapped to a table column with the same name. Only elements that have a corresponding table column are inserted into the table. Other elements are discarded.

formatted-event-table-info

Defines the target tables for an event monitor. This clause should be specified for each grouping that is to be recorded. However, if no evm-group-info clauses are specified, all groups for the event monitor type are recorded.

For more information about logical data groups, see "Logical data groups and event monitor output tables" in *Database Monitoring Guide and Reference*.

evm-group

Identifies the logical data group for which a target table is being defined. The value depends upon the type of event monitor, as shown in the following table:

Type of Event Monitor	evm-group Value
Activities	• ACTIVITY • ACTIVITYMETRICS • ACTIVITYSTMT • ACTIVITYVALS • CONTROL

target-table-options

Identifies the target table for the group.

TABLE *table-name*

Specifies the name of the target table. The target table must be a non-partitioned table. If the name is unqualified, the table schema defaults to the value in the CURRENT SCHEMA special register. If no name is provided, the unqualified name is derived from *evm-group* and *event-monitor-name* as follows:

```
substring(evm-group CONCAT '_'
  CONCAT event-monitor-name,1,128)
```

IN *tablespace-name*

Defines the table space in which the table is to be created. If no table space name is provided, the table space is chosen using the same process as when a table is created without a table space name using CREATE TABLE.

Since the page size affects the INLINE LOB lengths used, consider specifying a table space with as large a page size as possible in order to improve the INSERT performance of the event monitor.

PCTDEACTIVATE *integer*

If a table for the event monitor is being created in an automatic storage (non-temporary) or DMS table space, the PCTDEACTIVATE parameter specifies how full the table space must be before the event monitor automatically deactivates. The specified value, which represents a percentage, can range from 0 to 100, where 100 means that the event monitor deactivates when the table space becomes completely full. The default value assumed is 100 if PCTDEACTIVATE is not specified. This option is ignored for SMS table spaces.

Important: If the target table space has auto-resize enabled, set PCTDEACTIVATE parameter to 100. Alternatively, omit this clause entirely to have the default of 100 apply. Otherwise, the event monitor might deactivate unexpectedly if the table space reaches the threshold specified by PCTDEACTIVTATE before the table space is automatically resized.

If a value for *target-table-options* is not specified, CREATE EVENT MONITOR processing proceeds as follows:

- A derived table name is used.
- A default table space is chosen.

- The PCTDEACTIVATE parameter defaults to 100.

BLOCKED
Specifies that each agent that generates an event should wait for an event buffer to be written out to disk if the agent determines that both event buffers are full. BLOCKED should be selected to guarantee no event data loss. This is the default option.

PIPE
Specifies that the target for the event monitor data is a named pipe. The event monitor writes the data to the pipe in a single stream (that is, as if it were a single, infinitely long file). When writing the data to a pipe, an event monitor does not perform blocked writes. If there is no room in the pipe buffer, then the event monitor will discard the data. It is the monitoring application's responsibility to read the data promptly if it wishes to ensure no data loss.

pipe-name
The name of the pipe (FIFO on AIX) to which the event monitor will write the data.

The naming rules for pipes are platform specific.

Operating system	Naming rules
AIX	Pipe names are treated like file names. As a result, relative pipe names are permitted, and are treated like relative path-names (see description for *path-name*).
Linux	Pipe names are treated like file names. As a result, relative pipe names are permitted, and are treated like relative path-names (see description for *path-name*).
Windows	There is a special syntax for a pipe name and, as a result, absolute pipe names are required.

The existence of the pipe will not be checked at event monitor creation time. It is the responsibility of the monitoring application to have created and opened the pipe for reading at the time that the event monitor is activated. If the pipe is not available at this time, then the event monitor will turn itself off, and will log an error. (That is, if the event monitor was activated at database start time as a result of the AUTOSTART option, then the event monitor will log an error in the system error log.) If the event monitor is activated via the SET EVENT MONITOR STATE SQL statement, then that statement will fail (SQLSTATE 58030).

FILE
Indicates that the target for the event monitor data is a file (or set of files). The event monitor writes out the stream of data as a series of 8 character numbered files, with the extension "evt". (for example, `00000000.evt`, `00000001.evt`, and `00000002.evt`). The data should be considered to be one logical file even though the data is broken up into smaller pieces (that is, the start of the data stream is the first byte in the file `00000000.evt`; the end of the data stream is the last byte in the file *nnnnnnnn*`.evt`).

The maximum size of each file can be defined as well as the maximum number of files. An event monitor will never split a single event record across two files. However, an event monitor may write related records in two different files. It is the responsibility of the application that uses this data to keep track of such related information when processing the event files.

path-name
The name of the directory in which the event monitor should write the event files data. The path must be known at the server; however, the path itself could reside on another database partition (for example, an NFS mounted file). A string constant must be used when specifying the *path-name*.

The directory does not have to exist at CREATE EVENT MONITOR time. However, a check is made for the existence of the target path when the event monitor is activated. At that time, if the target path does not exist, an error (SQLSTATE 428A3) is raised.

If an absolute path is specified, the specified path will be the one used. In environments other than Db2 pureScale, if a relative path (a path that does not start with the root) is specified, then the path relative to the DB2EVENT directory in the database directory will be used. In a Db2 pureScale environment, if a relative path is specified, then the path relative to the database owning directory in the database directory will be used.

It is possible to specify two or more event monitors that have the same target path. However, once one of the event monitors has been activated for the first time, and as long as the target directory is not empty, it will be impossible to activate any of the other event monitors.

file-options
Specifies the options for the file format.

MAXFILES NONE
Specifies that there is no limit to the number of event files that the event monitor will create. This is the default.

MAXFILES *number-of-files*
Specifies that there is a limit on the number of event monitor files that will exist for a particular event monitor at any time. Whenever an event monitor has to create another file, it will check to make sure that the number of .evt files in the directory is less than *number-of-files*. If this limit has already been reached, then the event monitor will turn itself off.

If an application removes the event files from the directory after they have been written, then the total number of files that an event monitor can produce can exceed *number-of-files*. This option has been provided to allow a user to guarantee that the event data will not consume more than a specified amount of disk space.

MAXFILESIZE *pages*
Specifies that there is a limit to the size of each event monitor file. Whenever an event monitor writes a new event record to a file, it checks that the file will not grow to be greater than *pages* (in units of 4K pages). If the resulting file would be too large, then the event monitor switches to the next file. The default for this option is:

- Linux - 1000 4K pages
- UNIX - 1000 4K pages
- Windows - 200 4K pages

The number of pages must be greater than at least the size of the event buffer in pages. If this requirement is not met, then an error (SQLSTATE 428A4) is raised.

MAXFILESIZE NONE
Specifies that there is no set limit on a file's size. If MAXFILESIZE NONE is specified, then MAXFILES 1 must also be specified. This option means that one file will contain all of the event data for a particular event monitor. In this case the only event file will be 00000000.evt.

BLOCKED
Specifies that each agent that generates an event should wait for an event buffer to be written out to disk if the agent determines that both event buffers are full. BLOCKED should be selected to guarantee no event data loss. This is the default option.

APPEND
Specifies that if event data files already exist when the event monitor is turned on, then the event monitor will append the new event data to the existing stream of data files. When the event monitor is reactivated, it will resume writing to the event files as if it had never been turned off. APPEND is the default option.

The APPEND option does not apply at CREATE EVENT MONITOR time, if there is existing event data in the directory where the newly created event monitor is to write its event data.

REPLACE
Specifies that if event data files already exist when the event monitor is turned on, then the event monitor will erase all of the event files and start writing data to file `00000000.evt`.

MANUALSTART
Specifies that the event monitor must be activated manually using the SET EVENT MONITOR STATE statement. After a MANUALSTART event monitor has been activated, it can be deactivated only by using the SET EVENT MONITOR STATE statement or by stopping the instance.

AUTOSTART
Specifies that the event monitor is to be automatically activated whenever the database partition on which the event monitor runs is activated. This is the default behavior of the activities event monitor.

ON MEMBER *member-number*
Specifies the member on which a file or pipe event monitor is to run. When the monitoring scope is defined as LOCAL, data is collected only on the specified member. The I/O component will physically run on the specified member, writing records to the specified file or pipe.

When the Db2 pureScale feature is enabled, -1 is the default.

If a value of -1 is specified, it allows the I/O component to run from any active member. Additionally, in the event that the I/O component is no longer able to run on a given member, the event monitor will be restarted with the I/O component running on another available active member.

This clause is not valid for table event monitors. In a partitioned database environment, write-to-table event monitors will run and write events on all database partitions where table spaces for target tables are defined.

In a Db2 pureScale environment, write-to-table event monitors will record events on all active members.

If this clause is not specified and Db2 pureScale is not enabled, the currently connected member (for the application) is used.

If this clause is not specified and Db2 pureScale is enabled, the I/O component is able to run on any currently connected member.

LOCAL
The event monitor reports only on the member that is running. It gives a partial trace of the database activity. This is the default.

This clause is valid for file or pipe monitors. It is not valid for table event monitors.

GLOBAL is not a valid scope for this type of event monitor.

Rules

- The ACTIVITIES event type cannot be combined with any other event types in a particular event monitor definition.

Notes

- Event monitor definitions are recorded in the SYSCAT.EVENTMONITORS catalog view. The events themselves are recorded in the SYSCAT.EVENTS catalog view. The names of target tables are recorded in the SYSCAT.EVENTTABLES catalog view.
- If the member on which the event monitor is to run is not active, event monitor activation occurs when that member is reactivated.
- After an event monitor is activated, it behaves like an autostart event monitor until that event monitor is explicitly deactivated or the instance is recycled. That is, if an event monitor is active when a member

is deactivated, and that member is subsequently reactivated, the event monitor is also explicitly reactivated.

- The FLUSH EVENT MONITOR statement is not applicable to this event monitor and will have no effect when issued against it.
- ***Write to table event monitors***: General notes:
  - All target tables are created when the CREATE EVENT MONITOR statement executes.
  - If the creation of a table fails for any reason, an error is passed back to the application program, and the CREATE EVENT MONITOR statement fails.
  - A target table can only be used by one event monitor. During CREATE EVENT MONITOR processing, if a target table is found to have already been defined for use by another event monitor, the CREATE EVENT MONITOR statement fails, and an error is passed back to the application program. A table is defined for use by another event monitor if the table name matches a value found in the SYSCAT.EVENTTABLES catalog view.
  - During CREATE EVENT MONITOR processing, if a table already exists, but is *not* defined for use by another event monitor, no table is created, and processing continues. A warning is passed back to the application program.
  - Any table spaces must exist before the CREATE EVENT MONITOR statement is executed. The CREATE EVENT MONITOR statement does not create table spaces.
  - If specified, the LOCAL and GLOBAL keywords are ignored. With WRITE TO TABLE event monitors, an event monitor output process or thread is started on each member in the instance, and each of these processes reports data only for the member on which it is running.
  - The following event types from the flat monitor log file or pipe format are not recorded by write to table event monitors:
    - LOG_STREAM_HEADER
    - LOG_HEADER
    - DB_HEADER (Elements **db_name** and **db_path** are not recorded. The element **conn_time** is recorded in CONTROL.)
  - In a partitioned database environment, data is only written to target tables on the database partitions where their table spaces exist. If a table space for a target table does not exist on some database partition, data for that target table is ignored. This behavior allows users to choose a subset of database partitions for monitoring, by creating a table space that exists only on certain database partitions.

    In a Db2 pureScale environment, data will be written from every member.

    In a partitioned database environment, if some target tables do not reside on a database partition, but other target tables do reside on that same database partition, only the data for the target tables that do reside on that database partition is recorded.
  - Users must manually prune all target tables.

  Table Columns:

  - Column names in a table match an event monitor element identifier. Any event monitor element that does not have a corresponding target table column is ignored.
  - Use the **db2evtbl** command to build a CREATE EVENT MONITOR statement that includes a complete list of elements for a group.
  - The types of columns being used for monitor elements correlate to the following mapping:

```
SQLM_TYPE_STRING                     CHAR[n], VARCHAR[n] or CLOB(n)
                                     (If the data in the event monitor
                                     record exceeds n bytes,
                                     it is truncated.)
SQLM_TYPE_U8BIT and SQLM_TYPE_8BIT   SMALLINT, INTEGER or BIGINT
SQLM_TYPE_16BIT and SQLM_TYPE_U16BIT SMALLINT, INTEGER or BIGINT
SQLM_TYPE_32BIT and SQLM_TYPE_U32BIT INTEGER or BIGINT
SQLM_TYPE_U64BIT and SQLM_TYPE_64BIT BIGINT
sqlm_timestamp                       TIMESTAMP
```

```
sqlm_time(elapsed time)              BIGINT
sqlca:
  sqlerrmc                           VARCHAR[72]
  sqlstate                           CHAR[5]
  sqlwarn                            CHAR[11]
  other fields                       INTEGER or BIGINT
```

- Columns are defined to be NOT NULL.
- Unlike other target tables, the columns in the CONTROL table do not match monitor element identifiers. Columns are defined as follows:

Column Name	Data Type	Nullable	Description
PARTITION_KEY	INTEGER	N	Distribution key (partitioned database only)
PARTITION_NUMBER	INTEGER	N	Database partition number (partitioned database only)
EVMONNAME	VARCHAR(128)	N	Name of the event monitor
MESSAGE	VARCHAR(128)	N	Describes the nature of the MESSAGE_TIME column. For more details see, "message - Control Table Message monitor element" in the *Database Monitoring Guide and Reference*
MESSAGE_TIME	TIMESTAMP	N	Timestamp

- In a partitioned database environment, the first column of each table is named PARTITION_KEY, is NOT NULL, and is of type INTEGER. This column is used as the distribution key for the table. The value of this column is chosen so that each event monitor process inserts data into the member on which the process is running; that is, insert operations are performed locally on the member where the event monitor process is running. On any database partition, the PARTITION_KEY field will contain the same value. This means that if a database partition is dropped and data redistribution is performed, all data on the dropped database partition will go to one other database partition instead of being evenly distributed. Therefore, before removing a database partition, consider deleting all table rows on that database partition.
- In a partitioned database environment, a column named PARTITION_NUMBER can be defined for each table. This column is NOT NULL and is of type INTEGER. It contains the number of the database partition on which the data was inserted. Unlike the PARTITION_KEY column, the PARTITION_NUMBER column is not mandatory. The PARTITION_NUMBER column is not allowed in a non-partitioned database environment.

Table Attributes:

- Default table attributes are used. Besides distribution key (partitioned databases only), no extra options are specified when creating tables.
- Indexes on the table can be created.
- Extra table attributes (such as volatile, RI, triggers, constraints, and so on) can be added, but the event monitor process (or thread) will ignore them.
- If "not logged initially" is added as a table attribute, it is turned off at the first COMMIT, and is not set back on.

Event Monitor Activation:

- When an event monitor activates, all target table names are retrieved from the SYSCAT.EVENTTABLES catalog view.
- In a partitioned database environment, activation processing occurs on every member of the instance. On a particular member, activation processing determines the table spaces and database partition groups for each target table. The event monitor only activates on a database partition if at least one target table exists on that database partition. Moreover, if some target table is not found on a database partition, that target table is flagged so that data destined for that table is dropped during runtime processing.
- If a target table does not exist when the event monitor activates (or, in a partitioned database environment, if the table space does not reside on a database partition), activation continues, and data that would otherwise be inserted into this table is ignored.
- Activation processing validates each target table. If validation fails, activation of the event monitor fails, and messages are written to the administration log.
- During activation in a partitioned database environment, the CONTROL table rows for FIRST_CONNECT and EVMON_START are only inserted on the catalog database partition. This requires that the table space for the control table exist on the catalog database partition. If it does not exist on the catalog database partition, these inserts are not performed.
- In a partitioned database environment, if a member is not yet active when a write to table event monitor is activated, the event monitor will be activated the next time that member is activated.

Run Time:

- An event monitor runs with DATAACCESS authority.
- If, while an event monitor is active, an insert operation into a target table fails:
  - Uncommitted changes are rolled back.
  - A message is written to the administration log.
  - The event monitor is deactivated.
- If an event monitor is active, it performs a local COMMIT when it has finished processing an event monitor buffer.
- In an environment other than a partitioned database or a Db2 pureScale environment, all write to table event monitors are deactivated when the last application terminates (and the database has not been explicitly activated).

  In a Db2 pureScale environment, write to table event monitors are deactivated on a given member when the member stops and is reactivated when the member restarts.

  In a partitioned database environment, write to table event monitors are deactivated when the catalog partition deactivates.
- The DROP EVENT MONITOR statement does not drop target tables.
- Whenever a write-to-table event monitor activates, it will acquire IN table locks on each target table in order to prevent them from being modified while the event monitor is active. Table locks are maintained on all tables while the event monitor is active. If exclusive access is required on any of the target tables (for example, when a utility is to be run), first deactivate the event monitor to release the table locks before attempting such access.

- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - DBPARTITIONNUM or NODE can be specified in place of MEMBER, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
  - Commas can be used to separate multiple options in the *target-table-options* clause

Example

Define an activity event monitor named DBACTIVITIES

```
CREATE EVENT MONITOR DBACTIVITIES
    FOR ACTIVITIES
    WRITE TO TABLE
    ACTIVITY (TABLE ACTIVITY_DBACTIVITIES
              IN USERSPACE1
              PCTDEACTIVATE 100),
```

```
    ACTIVITYMETRICS (TABLE ACTIVITYMETRICS_DBACTIVITIES
                     IN USERSPACE1
                     PCTDEACTIVATE 100),
```

```
    ACTIVITYSTMT (TABLE ACTIVITYSTMT_DBACTIVITIES
                  IN USERSPACE1
                  PCTDEACTIVATE 100),
    ACTIVITYVALS (TABLE ACTIVITYVALS_DBACTIVITIES
                  IN USERSPACE1
                  PCTDEACTIVATE 100),
    CONTROL (TABLE CONTROL_DBACTIVITIES
             IN USERSPACE1
             PCTDEACTIVATE 100)
    AUTOSTART;
```

CREATE EVENT MONITOR (change history)

The CREATE EVENT MONITOR (change history) statement creates an event monitor that can record events for changes to configuration parameters, registry variables, and the execution of DDL statements and utilities.

The event monitor created by the CREATE EVENT MONITOR (change history) statement can also record initial configuration and registry values at event monitor startup time. The set of events recorded depends on the event controls specified in the CREATE EVENT MONITOR statement.

Invocation

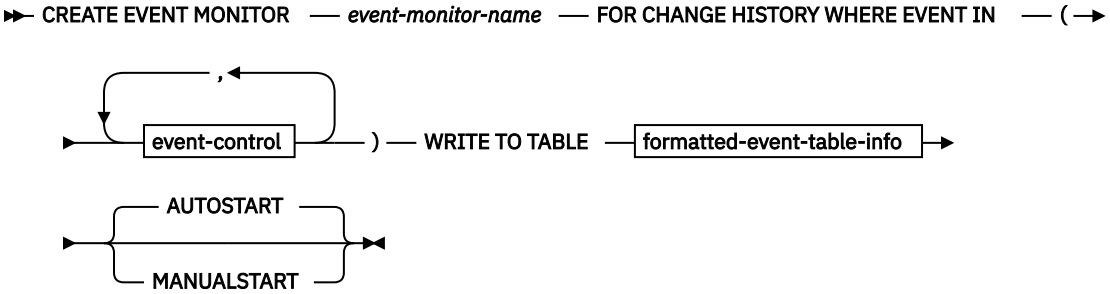
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

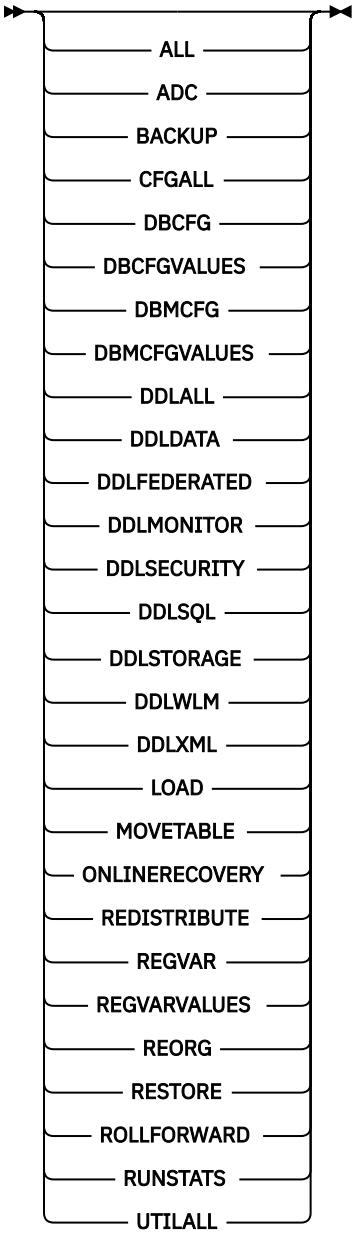
The privileges held by the authorization ID of the statement must include one of the following authorities:

- SQLADM authority
- DBADM authority

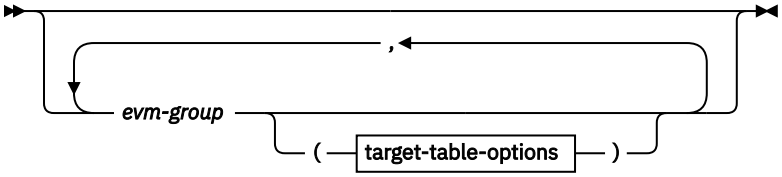
Syntax



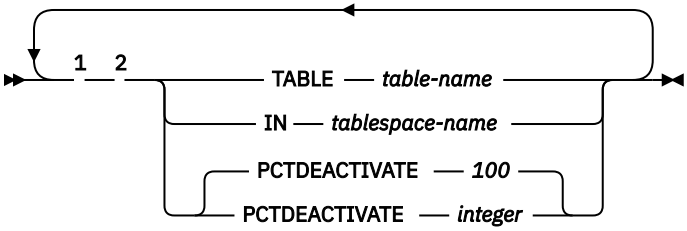
event-control



formatted-event-table-info



target-table-options



Notes:

[1] Each condition can be specified only once (SQLSTATE 42613).

[2] Clauses can be separated with a space or a comma.

Description

event-monitor-name

Name of the event monitor. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *event-monitor-name* must not identify an event monitor that exists in the catalog (SQLSTATE 42710).

FOR

Introduces the type of event to record.

CHANGE HISTORY

Specifies that this event monitor can record events for configuration changes, registry changes, and the execution of DDL statements and utilities. It can also record initial configuration and registry values at event monitor startup time. The set of events recorded depends on the event controls specified in the WHERE EVENT IN clause.

WHERE EVENT IN (event-control, ...)

Specifies one or more event controls used to identify which events are captured by the event monitor.

event-control

ALL

Capture all event types.

ADC

Capture execution of the automatic dictionary creation (ADC) utility.

BACKUP

Capture execution of the online backup utility.

CFGALL

Capture all configuration parameter and registry variable event types.

DBCFG

Capture database configuration parameter changes.

DBCFGVALUES

Record initial values for all database configuration parameters at event monitor startup time if any database configuration parameter update was not captured by the event monitor.

DBMCFG

Capture database manager configuration parameter changes.

DBMCFGVALUES

Record initial values for all database manager configuration parameters at event monitor startup time if any database manager configuration parameter update was not captured by the event monitor.

DDLALL

Capture execution for all types of DDL statements.

DDLDATA

Capture execution of index, sequence, table, and temporary table DDL.

DDLFEDERATED

Capture execution of nickname, server, type mapping, user mapping, and wrapper DDL.

DDLMONITOR

Capture execution of event monitor and usage list DDL.

DDLSECURITY

Capture execution of audit policy, grant, mask, permission role, revoke, security label, security label component, security policy, and trusted context DDL.

DDLSQL
Capture execution of alias, function, method, module, package, procedure, schema, synonym, transform, trigger, type, variable, and view DDL.

DDLSTORAGE
Capture execution of the ALTER DATABASE statement and buffer pool, partition group, storage group, and table space DDL.

DDLWLM
Capture execution of histogram, service class, threshold, work action set, work class set, and workload DDL.

DDLXML
Capture execution of XSROBJECT DDL.

LOAD
Capture execution of the load utility.

MOVETABLE
Capture execution of the table move utility (invocations of the ADMIN_MOVE_TABLE stored procedure).

ONLINERECOVERY
Capture execution of a crash recovery operation which utilized an asynchronous backward phase that allowed for database connectivity during the operation. (This includes the implicit crash recovery performed during an HADR TAKEOVER).

REDISTRIBUTE
Capture execution of the redistribute partition group utility.

REGVAR
Capture immediate registry variables changes.

REGVARVALUES
Record initial values for registry variables at event monitor startup time.

REORG
Capture execution of the reorg utility.

RESTORE
Capture execution of the online restore utility.

ROLLFORWARD
Capture execution of the online rollforward utility.

RUNSTATS
Capture execution of the runstats utility.

UTILALL
Capture execution of the load, move table, online backup, online restore, online rollforward, redistribute, reorg and runstats utilities.

WRITE TO
Introduces the target for the data.

TABLE
Indicates that the target for the event monitor data is a set of database tables. The event monitor separates the data stream into one or more logical data groups and inserts each group into a separate table. Each monitor element contained within a group is mapped to a table column with the same name. Only elements that have a corresponding table column are inserted into the table.

formatted-event-table-info
Defines the target table for a logical data group. Specify this clause for each grouping that is to be recorded. However, if no *evm-group* clauses are specified, the groups required for the event-control options specified are created along with the CONTROL, CHANGESUMMARY, and EVMONSTART logical groups.

evm-group
Identifies the logical data group for which a target table is being defined. The value depends upon the type of event monitor, as shown in the following table:

Type of event monitor	evm-group value
Change history	• CONTROL • CHANGESUMMARY • EVMONSTART • TXNCOMPLETION • DDLSTMTEXEC • DBDBMCFG • REGVAR • UTILSTART • UTILSTOP • UTILPHASE • UTILLOCATION

target-table-options
Identifies the target table for the group.

TABLE *table-name*
Specifies the name of the target table. The target table must be a non-partitioned table. If the name is unqualified, the table schema defaults to the value in the CURRENT SCHEMA special register. If no name is provided, the unqualified name is derived from *evm-group* and *event-monitor-name* as follows:

```
SUBSTRING(evm-group CONCAT '_'
   CONCAT event-monitor-name, 1, 128)
```

IN *tablespace-name*
Defines the table space in which the table is to be created. If no table space name is provided, the table space is chosen using the same process as when a table is created without a table space name using CREATE TABLE.

PCTDEACTIVATE *integer*
If a table for the event monitor is being created in an automatic storage (non-temporary) or DMS table space, the PCTDEACTIVATE parameter specifies how full the table space must be before the event monitor automatically deactivates. The specified value, which represents a percentage, can range from 0 to 100, where 100 means that the event monitor deactivates when the table space becomes full. The default value is 100 if PCTDEACTIVATE is not specified. This option is ignored for SMS table spaces.

Important: If the target table space has auto-resize enabled, set PCTDEACTIVATE parameter to 100. Alternatively, omit this clause entirely to have the default of 100 apply. Otherwise, the event monitor might deactivate unexpectedly if the table space reaches the threshold specified by PCTDEACTIVTATE before the table space is automatically resized.

If a value for target-table-info is not specified, CREATE EVENT MONITOR processing proceeds as follows:

- A derived table name is used.
- A default table space is chosen.
- The PCTDEACTIVATE parameter defaults to 100.

AUTOSTART
Specifies that the event monitor is to be automatically activated whenever the database partition on which the event monitor runs is activated. This is the default behavior.

MANUALSTART
Specifies that the event monitor must be activated manually using the SET EVENT MONITOR STATE statement. After a MANUALSTART event monitor is activated, it can be deactivated by using the SET EVENT MONITOR STATE statement or by stopping the instance.

Notes

- ***Creation of target event tables:*** The target event tables are created when the CREATE EVENT MONITOR FOR CHANGE HISTORY statement executes if the target tables do not exist.
- ***Previously created event tables:*** During CREATE EVENT MONITOR FOR CHANGE HISTORY processing, if an event table has already been defined for use by another event monitor, the CREATE EVENT MONITOR FOR CHANGE HISTORY statement fails, and an error is returned to the application program. An event table is defined for use by another event monitor if the event table name matches a value found in the SYSCAT.EVENTTABLES catalog view. If the event table exists and is not defined for use by another event monitor, then a table is not created, any other table options parameters are ignored, and processing continues. A warning is returned to the application program.
- ***Dropping event monitors:*** Dropping the event monitor does not drop the event tables. The associated event tables must be manually dropped after the event monitor is dropped.
- ***Pruning:*** The event tables must be manually pruned.
- ***Behavior in a partitioned environment:*** In a partitioned environment, if some target event tables do not exist on a partition, but other target event tables do exist on that same partition, only the data for the target event tables that do exist on that partition is recorded.
- ***FLUSH EVENT MONITOR:*** The FLUSH EVENT MONITOR statement is not applicable to this event monitor and has no effect when issued against it.
- ***Modifying event controls after monitor creation:*** After the change history event monitor is created, the event controls specified using the WHERE EVENT IN clause in the CREATE EVENT MONITOR statement cannot be changed or altered. To change the event controls, the event monitor must be deactivated, dropped, and then recreated specifying a new set of event controls using the WHERE EVENT IN clause.

Examples

- *Example 1*: This example creates a change history event monitor called CFG_WITH_OFFLINE that records configuration changes and initial values for configuration.

```
CREATE EVENT MONITOR CFG_WITH_OFFLINE
   FOR CHANGE HISTORY WHERE EVENT IN (CFGALL)
   WRITE TO TABLE
     CHANGESUMMARY (TABLE CHG_SUMMARY_HISTORY),
     DBDBMCFG (TABLE DB_DBM_HISTORY),
     REGVAR (TABLE REGVAR_HISTORY)
   AUTOSTART
```

  In this example the target tables are explicitly specified. The previous statement creates the following tables:

```
CHG_SUMMARY_HISTORY
DB_DBM_HISTORY
REGVAR_HISTORY
```

- *Example 2*: This example creates a change history event monitor called BKP_REST that collects events describing all online backup and restore utility executions.

```
CREATE EVENT MONITOR BKP_REST
   FOR CHANGE HISTORY WHERE EVENT IN (BACKUP, RESTORE)
   WRITE TO TABLE
```

  In this example the target tables are not explicitly specified. The CREATE EVENT MONITOR statement creates only the target tables that are needed based on the controls specified in the WHERE EVENT IN clause, along with tables for the CONTROL, CHANGESUMMARY, and EVMONSTART logical data groups. The BACKUP and RESTORE controls enable collection of utility events for online backup and

restore, and require the UTILSTART, UTILSTOP, UTILLOCATION, and UTILPHASE logical data groups. The previous statement creates the following tables:

```
CONTROL_BKP_REST
CHANGESUMMARY_BKP_REST
EVMONSTART_BKP_REST
UTILSTART_BKP_REST
UTILSTOP_BKP_REST
UTILLOCATION_BKP_REST
UTILPHASE_BKP_REST
```

CREATE EVENT MONITOR (locking)

The CREATE EVENT MONITOR (locking) statement creates an event monitor that will record lock-related events that occur when using the database.

Invocation

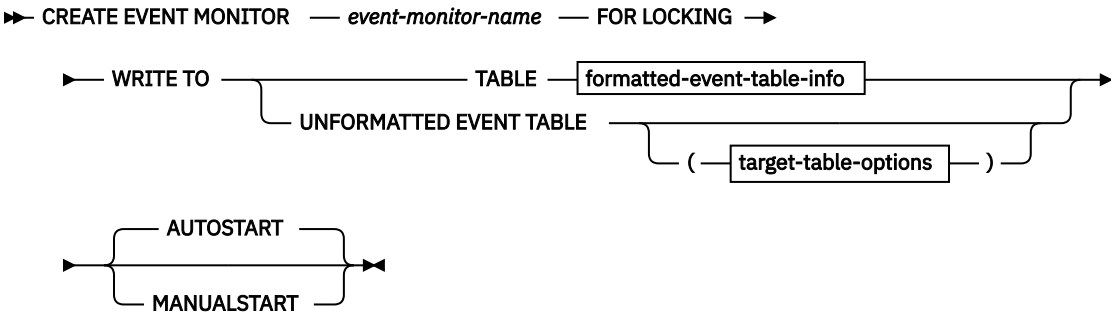
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

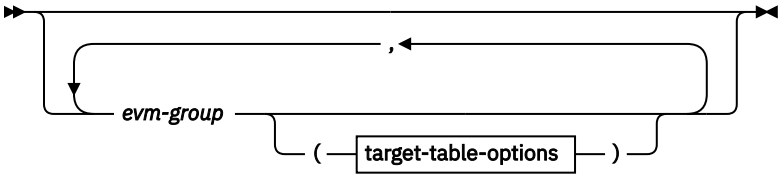
The privileges held by the authorization ID of the statement must include one of the following authorities:

- DBADM authority
- SQLADM authority

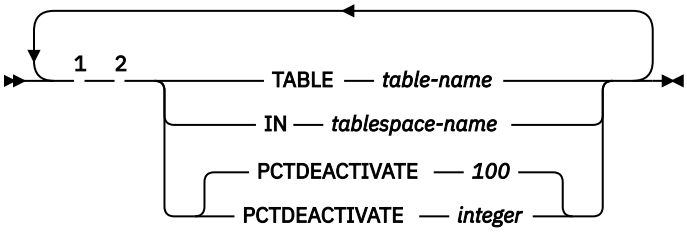
Syntax



formatted-event-table-info



target-table-options



Notes:

[1] Each table option can be specified a maximum of one time (SQLSTATE 42613).
[2] Clauses can be separated with a space or a comma.

Description

event-monitor-name
Name of the event monitor. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *event-monitor-name* must not identify an event monitor that already exists in the catalog (SQLSTATE 42710).

FOR
Introduces the type of event to record.

LOCKING
Specifies that this passive event monitor will record any lock event produced when the database manager encounters one or more of these conditions:

- LOCKTIMEOUT: the lock has timed-out.
- DEADLOCK: the lock was involved in a deadlock (victim and participant(s)).
- LOCKWAIT: locks that are not acquired in the specified duration.

The creation of the lock event monitor does not indicate that the locking data will be collected immediately. The actual locking event of interest is controlled at the workload level or database level.

WRITE TO
Specifies the target for the data.

TABLE
Indicates that the target for the event monitor data is a set of formatted event tables. The event monitor separates the data stream into one or more logical data groups and inserts each group into a separate table. Data for groups having a target table is kept, whereas data for groups not having a target table is discarded. Each monitor element contained within a group is mapped to a table column with the same name. Only elements that have a corresponding table column are inserted into the table. Other elements are discarded.

formatted-event-table-info
Defines the target formatted event tables for the event monitor. This clause should specify each grouping that is to be recorded. However, if no *evm-group* clauses are specified, all groups for the event monitor type are recorded.

For more information about logical data groups, see "Logical data groups and event monitor output tables" in *Database Monitoring Guide and Reference*.

evm-group
Identifies a logical data group for which a target table is being defined. The value depends upon the type of event monitor, as shown in the following table:

Type of Event Monitor	evm-group Value
Locking	• LOCK • LOCK_PARTICIPANTS • LOCK_PARTICIPANT_ACTIVITIES • LOCK_ACTIVITY_VALUES • CONTROL

UNFORMATTED EVENT TABLE
Specifies that the target for the event monitor is an unformatted event table. The unformatted event table is used to store collected locking event monitor data. Data is stored in an internal binary format within an inlined BLOB column. Each event can insert multiple records into this table and each inserted record can be of a different type with the associated BLOB content varying as

well. The data in the BLOB column is not in a readable format and requires conversion, through use of the **db2evmonfmt** Java-based tool, EVMON_FORMAT_UE_TO_XML table function, or EVMON_FORMAT_UE_TO_TABLES procedure, into a consumable format such as an XML document or a relational table.

target-table-options
Identifies options for the target table. If a value for **target-table-options** is not specified, CREATE EVENT MONITOR FOR LOCKING processing proceeds as follows:

- A derived table name is used (as explained in the description for TABLE *table-name*).
- A default table space is chosen using the same process as when a table is created without a table space name using CREATE TABLE.
- PCTDEACTIVATE is set to 100.

TABLE *table-name*
Specifies the name of the target table. The target table must be a non-partitioned table. If the name is unqualified, the table schema defaults to the value in the CURRENT SCHEMA special register. If a name is not provided for an unformatted event table, the unqualified name is equal to the *event-monitor-name*, that is, the unformatted event table will be named after the event monitor. If no name is provided for a formatted event table, the unqualified name is derived from *evm-group* and *event-monitorname* as follows:

```
substring(evm-group CONCAT '_'
  CONCAT event-monitor-name,1,128)
```

IN *tablespace-name*
Defines the table space in which the table is to be created. The CREATE EVENT MONITOR FOR LOCKING statement does not create table spaces.

If a table space name is not provided, the table space is chosen using the same process as when a table is created without a table space name using CREATE TABLE.

When specifying the table space name for a formatted event table, the table space's page size affects the INLINE LOB lengths used Consider specifying a table space with as large a page size as possible in order to improve the INSERT performance of the event monitor.

PCTDEACTIVATE *integer*
If a table for the event monitor is being created in an automatic storage (non-temporary) or DMS table space, the PCTDEACTIVATE parameter specifies how full the table space must be before the event monitor automatically deactivates. The specified value, which represents a percentage, can range from 0 to 100. The default value is 100, where 100 means the event monitor deactivates when the table space becomes completely full. The default value assumed is 100 if PCTDEACTIVATE is not specified. This option is ignored for SMS table spaces.

Important: If the target table space has auto-resize enabled, set PCTDEACTIVATE to 100. Alternatively, omit this clause entirely to have the default of 100 apply. Otherwise, the event monitor might deactivate unexpectedly if the table space reaches the threshold specified by PCTDEACTIVTATE before the table space is automatically resized.

AUTOSTART
Specifies that the event monitor is to be automatically activated whenever the database partition on which the event monitor runs is activated. This is the default behavior of the locking event monitor.

MANUALSTART
Specifies that the event monitor must be activated manually using the SET EVENT MONITOR STATE statement. After a MANUALSTART event monitor has been activated, it can be deactivated only by using the SET EVENT MONITOR STATE statement or by stopping the instance.

Notes

- The target table is created when the CREATE EVENT MONITOR FOR LOCKING statement executes, if it doesn't already exist.

- During CREATE EVENT MONITOR FOR LOCKING processing, if a table is found to have already been defined for use by another event monitor, the CREATE EVENT MONITOR FOR LOCKING statement fails, and an error is passed back to the application program. A table is defined for use by another event monitor if the table name matches a value found in the SYSCAT.EVENTTABLES catalog view. If the table exists and is not defined for use by another event monitor, then the event monitor will re-use the table.
- Dropping the event monitor will not drop any tables. Any associated tables must be manually dropped after the event monitor is dropped.
- Lock event data is not automatically pruned from either unformatted event tables or regular tables created by this event monitor. An option for pruning data from UE tables is available when using the EVMON_FORMAT_UE_TO_TABLES procedure. For event monitors that write to regular tables, event data must be pruned manually.
- The FLUSH EVENT MONITOR statement is not applicable to this event monitor and will have no effect when issued against it.
- For unformatted event tables event data is inserted into the table into an inlined BLOB data column. Normally, BLOB data is stored in a separate LOB table space and can experience additional performance overhead as a result. When inlined into the data page of the base table, the BLOB data does not experience this overhead. The database manager will automatically inline the BLOB data portion of an unformatted event table record if the size of the BLOB data is less than the table space page size minus the record prefix. Therefore to achieve high efficiency and application throughput, it is suggested that you create the event monitor in as large a table space as possible up to and including a 32KB table space and associated bufferpool.

  Example

  The lock event monitor currently has the following two record types:

  – Application Info Record

  – Application Activity Record

  Application Info Record = maximum size 3.5KB

  Application Activity Record = 3KB + SQL statement text size (where SQL statement text size is max 2MB)

  The Application Info Record is very small and should always be inlined as long as a 4KB page size is being used. The Application Activity Record will be inlined based on the following formula:

  ```
  Application Activity Record < inline length (Pagesize - overhead non-LOB
  columns (0.5KB))
     3KB + SQL statement text < inline length (Pagesize - overhead non-LOB
  columns (0.5KB))

           SQL statement text < Pagesize - nonLOB overhead (1K) - 3KB
           SQL statement text < 16KB - 1KB - 3KB
                             < 12KB
  ```

  Therefore, when using a 16KB pagesize, the lock event monitor records will only be inlined if the SQL statement being captured is less than 12KB in size.
- Create only one locking event monitor per database. Creating more than one locking event monitor uses additional processor cycles and storage, without providing any additional data.

  Important: For compatibility with older versions of the product, all databases are created with the DB2DETAILDEADLOCK event monitor enabled. The locking event monitor introduced in Db2 Version 9.7 is the preferred mechanism for collecting data related to locks; the DB2DETAILEDDEALOCK event monitor is deprecated and might be removed in a future release. When you create a locking event monitor, disable and drop the DB2DETAILEDDEADLOCK event monitor to prevent the collection of duplicate, unnecessary information.

  To remove the DB2DETAILDEADLOCK event monitor, issue the following SQL statements:

  ```
  SET EVENT MONITOR DB2DETAILDEADLOCK state 0
  DROP EVENT MONITOR DB2DETAILDEADLOCK
  ```

- In a partitioned database environment, data is written only to target tables on the database partitions where their table spaces exist. If a table space for a target table does not exist on some database partition, data for that target table is ignored. This behavior allows users to choose a subset of database partitions for monitoring to be chosen, by creating a table space that exists only on certain database partitions.
- In a partitioned database environment, if some target tables do not reside on a database partition, but other target tables do reside on that same database partition, only the data for the target unformatted event tables that do reside on that database partition is recorded.

Examples

- *Example 1:* This example creates a locking event monitor LOCKEVMON that will collect locking events that occur on the database of creation.

```
CREATE EVENT MONITOR LOCKEVMON
  FOR LOCKING
  WRITE TO TABLE
```

  This event monitor writes its output to the following tables:

  LOCK_LOCKEVMON
  LOCK_PARTICIPANTS_LOCKEVMON
  LOCK_PARTICIPANT_ACTIVITIES_LOCKEVMON
  LOCK_ACTIVITY_VALUES_LOCKEVMON
  CONTROL_LOCKEVMON

- *Example 2:* This example creates a locking event monitor LOCKEVMON that will collect locking events that occur on the database of creation and store it in the unformatted event table IMRAN.LOCKEVENTS.

```
CREATE EVENT MONITOR LOCKEVMON
  FOR LOCKING
  WRITE TO UNFORMATTED EVENT TABLE (TABLE IMRAN.LOCKEVENTS)
```

- *Example 3:* This example creates a locking event monitor LOCKEVMON that will collect locking events that occur on the database of creation and store it in the unformatted event table IMRAN.LOCKEVENTS in table space APPSPACE. The event monitor will deactivate when the table space becomes 85% full.

```
CREATE EVENT MONITOR LOCKEVMON
  FOR LOCKING
  WRITE TO UNFORMATTED EVENT TABLE
     (TABLE IMRAN.LOCKEVENTS IN APPSPACE PCTDEACTIVATE 85)
```

CREATE EVENT MONITOR (package cache) statement

The CREATE EVENT MONITOR (package cache) statement creates an event monitor that will record events when the cache entry for a section is flushed from the package cache.

Invocation

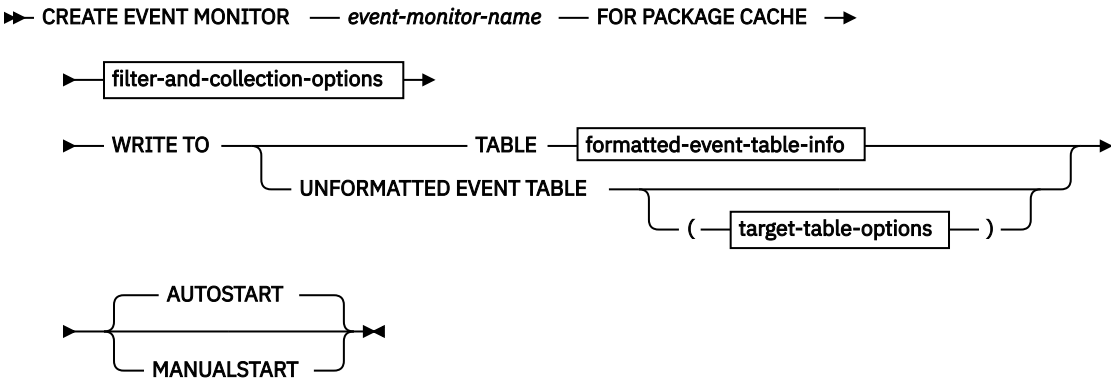
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

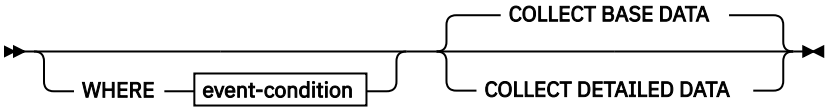
The privileges held by the authorization ID of the statement must include one of the following authorities:

- DBADM authority
- SQLADM authority

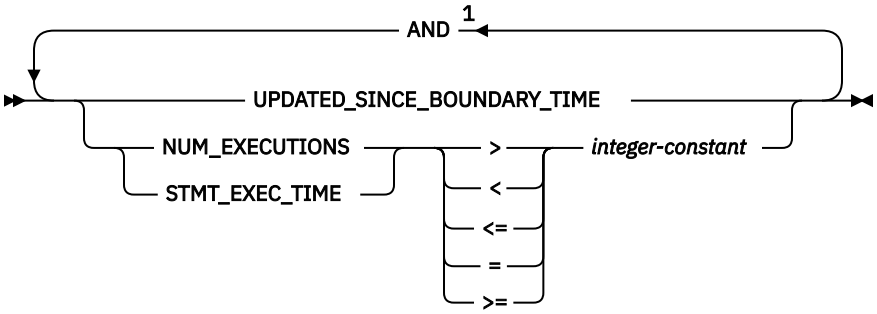
Syntax



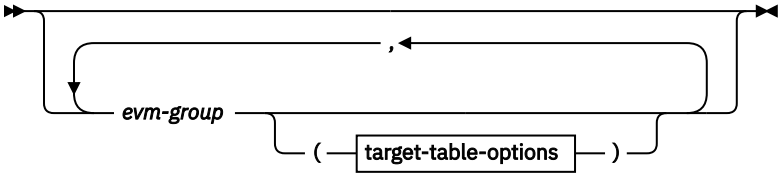
filter-and-collection-options



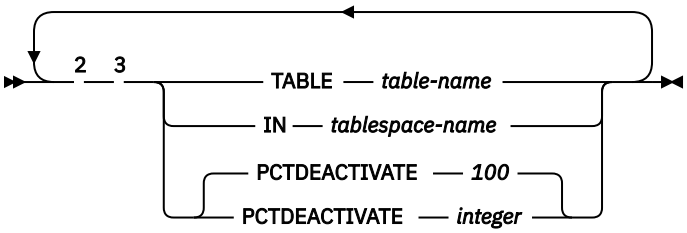
event-condition



formatted-event-table-info



target-table-options



Notes:

[1] Each condition can be specified only once (SQLSTATE 42613).
[2] Each table option can be specified a maximum of one time (SQLSTATE 42613).
[3] Clauses can be separated with a space or a comma.

Description

event-monitor-name
Name of the event monitor. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *event-monitor-name* must not identify an event monitor that already exists in the catalog (SQLSTATE 42710).

FOR
Introduces the type of event to record.

PACKAGE CACHE
Specifies that this event monitor will record an event when the cache entry for a static or dynamic SQL statement is flushed from the package cache. This event monitor is not passive and will start to record events once it is activated.

filter-and-collection-options
Specify a set of filter and collection options.

WHERE

event-condition
Defines a filter that determines whether entries that are flushed from the package cache should cause an event to occur. If the event condition is TRUE for a particular entry that is being flushed from the package cache, then that entry will be recorded as an event.

This clause is a special form of the WHERE clause that should not be confused with a standard search condition. This is a simple WHERE clause that includes the use of NOT, OR, and LIKE operators, unlike the WHERE clause specified for the CONNECTIONS, TRANSACTIONS, and STATEMENTS event monitors.

If the WHERE clause is not specified, all entries flushed from the package cache will be monitored.

UPDATED_SINCE_BOUNDARY_TIME
Specifies that evicted entries, whose metrics were updated after the boundary time, should be collected by this event monitor. The boundary time is set by calling the MON_GET_PKG_CACHE_STMT table function with the value of the input key "updated_boundary_time" set as the name of this event monitor.

The boundary time is initially set to the activation timestamp of the event monitor.

NUM_EXECUTIONS > | < | <= | = | >= *integer-constant*
Specifies that the monitor element **num_executions** should be compared with the *integer-constant* in order to determine whether to generate an event. NUM_EXECUTIONS is the number of times that the section of the evicted entry was executed.

Note: The **num_executions** monitor element counts all executions of a statement, whether or not the execution of the statement contributed to the activity metrics that are reported.

STMT_EXEC_TIME > | < | <= | = | >= *integer-constant*
Specifies that the monitor element **stmt_exec_time** should be compared with the *integer-constant* in order to determine whether to generate an event. STMT_EXEC_TIME is the total aggregated time spent executing the statement of the evicted entry. The unit of time for the *integer-constant* must be specified as milliseconds.

COLLECT BASE DATA
Specifies that the same level of information returned by the MON_GET_PKG_CACHE_STMT table function should be captured. This is the default collect option.

COLLECT DETAILED DATA
Specifies that the BASE level information should be collected as well as the runtime executable section of the flushed entry.

WRITE TO
Specifies the target for the data.

TABLE
Indicates that the target for the event monitor data is a set of database tables. The event monitor separates the data stream into one or more logical data groups and inserts each group into a separate table. Data for groups having a target table is kept, whereas data for groups not having a target table is discarded. Each monitor element contained within a group is mapped to a table column with the same name. Only elements that have a corresponding table column are inserted into the table. Other elements are discarded.

formatted-event-table-info
Defines the target formatted event tables for the event monitor. This clause should specify each grouping that is to be recorded. However, if no *evm-group* clauses are specified, all groups for the event monitor type are recorded.

For more information about logical data groups, see "Logical data groups and event monitor output tables" in *Database Monitoring Guide and Reference*.

evm-group
Identifies a logical data group for which a target table is being defined. The value depends upon the type of event monitor, as shown in the following table:

Type of Event Monitor	evm-group Value
Package Cache	• PKGCACHE • PKGCACHE_METRICS • PKGCACHE_STMT_ARGS • CONTROL

UNFORMATTED EVENT TABLE
Specifies that the target for the event monitor is an unformatted event table. The unformatted event table is used to store collected package cache event monitor data. Data is stored in its original binary format within an inlined BLOB column. The BLOB column can contain multiple binary records of different types. The data in the BLOB column is not in a readable format and requires conversion, through use of the **db2evmonfmt** Java-based tool, EVMON_FORMAT_UE_TO_XML table function, or EVMON_FORMAT_UE_TO_TABLES procedure, into a consumable format such as an XML document or a relational table.

target-table-options
Identifies options for the target table. If a value for **target-table-options** is not specified, CREATE EVENT MONITOR FOR PACKAGE CACHE processing proceeds as follows:

- A derived table name is used (as explained in the description for TABLE *table-name*).
- A default table space is chosen using the same process as when a table is created without a table space name using CREATE TABLE.
- PCTDEACTIVATE is set to 100.

TABLE *table-name*
Specifies the name of the target table. The target table must be a non-partitioned table. If the name is unqualified, the table schema defaults to the value in the CURRENT SCHEMA special register. If a name is not provided for an unformatted event table, the unqualified name is equal to the *event-monitor-name*, that is, the unformatted event table will be named after the event monitor. If no name is provided for a formatted event table, the unqualified name is derived from *evm-group* and *event-monitor-name* as follows:

```
substring(evm-group CONCAT '_'
  CONCAT event-monitor-name,1,128)
```

IN *tablespace-name*
Specifies the table space in which the table is to be created. The CREATE EVENT MONITOR FOR PACKAGE CACHE statement does not create table spaces.

If a table space name is not provided, the table space is chosen using the same process as when a table is created without a table space name using CREATE TABLE.

The table space's page size affects the INLINE LOB lengths used. Consider specifying a table space with as large a page size as possible in order to improve the INSERT performance of the event monitor.

PCTDEACTIVATE ***integer***
If a table for the event monitor is being created in an automatic storage (non-temporary) or DMS table space, the PCTDEACTIVATE parameter specifies how full the table space must be before the event monitor automatically deactivates. The specified value, which represents a percentage, can range from 0 to 100, where 100 means that the event monitor deactivates when the table space becomes completely full. The default value assumed is 100 if PCTDEACTIVATE is not specified. This option is ignored for SMS table spaces.

Important: If the target table space has auto-resize enabled, set PCTDEACTIVATE parameter to 100. Alternatively, omit this clause entirely to have the default of 100 apply. Otherwise, the event monitor might deactivate unexpectedly if the table space reaches the threshold specified by PCTDEACTIVTATE before the table space is automatically resized.

AUTOSTART
Specifies that the event monitor is to be automatically activated whenever the database partition on which the event monitor runs is activated. This is the default behavior of the package cache event monitor.

MANUALSTART
Specifies that the event monitor must be activated manually using the SET EVENT MONITOR STATE statement. After a MANUALSTART event monitor has been activated, it can be deactivated by using the SET EVENT MONITOR STATE statement or by stopping the instance.

Notes

- The target table is created when the CREATE EVENT MONITOR FOR PACKAGE CACHE statement executes, if it doesn't already exist.
- During CREATE EVENT MONITOR FOR PACKAGE CACHE processing, if a table is found to have already been defined for use by another event monitor, the CREATE EVENT MONITOR FOR PACKAGE CACHE statement fails, and an error is passed back to the application program. A table is defined for use by another event monitor if the table name matches a value found in the SYSCAT.EVENTTABLES catalog view. If the table exists and is not defined for use by another event monitor, then the event monitor will re-use the table.
- Dropping the event monitor will not drop any tables. Any associated tables must be manually dropped after the event monitor is dropped.
- Lock event data is not automatically pruned from either unformatted event tables or regular tables created by this event monitor. An option for pruning data from UE tables is available when using the EVMON_FORMAT_UE_TO_TABLES procedure. For event monitors that write to regular tables, event data must be pruned manually.
- In a partitioned database environment, data is written to target tables only on the members where their table spaces exist. If a table space for a target table does not exist on a member, the event data that would be written to that target table is not captured on that member. This behavior allows users to choose a subset of members for monitoring, by creating a table space that exists only on certain members.
- In a partitioned database environment, data is written to target tables only on the member where the entries are evicted from the database package cache.
- In a partitioned database environment, if some target tables do not reside on a member, but other target tables do reside on that same member, only the data for the target unformatted event tables that do reside on that member is recorded.
- The FLUSH EVENT MONITOR statement is not applicable to this event monitor and will have no effect when issued against it.

- After the package cache event monitor is created, the filter and control options cannot be changed or altered. To change the filter and control options, the event monitor must be deactivated, dropped, and then recreated with the new filter and control options.

Use large table space for high throughput
Event data is inserted into the unformatted event table into an inlined BLOB data column. Normally, BLOB data is stored in a separate LOB table space and can experience additional performance overhead as a result. When inlined into the data page of the base table, the BLOB data does not experience this overhead. The database manager will automatically inline the BLOB data portion of an unformatted event table record if the size of the BLOB data is less than the table space page size minus the record prefix. Therefore, to achieve high efficiency and application throughput, it is suggested that you create the event monitor in as large a table space as possible, up to and including a 32 KB table space, and associated bufferpool.

Inline of package cache records
For the package cache event monitor, the size of the **stmt_text**, **comp_env_desc**, and the **section_env** monitor elements will determine if the package cache record will be inlined or not. If the total of these fields exceeds the table space size, then the record will not be inlined.

Determine if EVENT_DATA is inlined
Use the ADMIN_IS_INLINED and ADMIN_EST_INLINE_LENGTH functions to determine whether the record is inlined and get an estimate of the inline length that is required.

Restrictions

- During database deactivation, evicted entries will not be collected by the package cache event monitor.

Examples

- *Example 1:* This example creates a package cache event monitor called CACHEEVMON that will collect data related to package cache section eviction events and write the data to tables.

```
CREATE EVENT MONITOR CACHEEVMON
  FOR PACKAGE CACHE
  WRITE TO TABLE
```

This event monitor writes its output to the following tables:

PKGCACHE_CACHEEVMON
PKGCACHE_METRICS_CACHEEVMON
PKGCACHE_STMT_ARGS
CONTROL_CACHEEVMON

- *Example 2:* This example creates a package cache event monitor called CACHESTMTEVMON that will collect data related to package cache section eviction events and store it in the unformatted event table ALAN.STMTEVENTS.

```
CREATE EVENT MONITOR CACHESTMTEVMON
  FOR PACKAGE CACHE
  WRITE TO UNFORMATTED EVENT TABLE (TABLE ALAN.STMTEVENTS)
```

- *Example 3:* This example creates a package cache event monitor called CACHESTMTEVMON that will collect data related to package cache section eviction events and store it in the unformatted event table ALAN.STMTEVENTS in table space APPSPACE. The event monitor will deactivate when the table space becomes 85% full.

```
CREATE EVENT MONITOR CACHESTMTEVMON
  FOR PACKAGE CACHE
  WRITE TO UNFORMATTED EVENT TABLE
     (TABLE ALAN.STMTEVENTS IN APPSPACE PCTDEACTIVATE 85)
```

CREATE EVENT MONITOR (statistics)

The CREATE EVENT MONITOR (statistics) statement defines a monitor that will record statistics events that occur when using the database. The definition of the statistics event monitor also specifies where the database should record the events.

Invocation

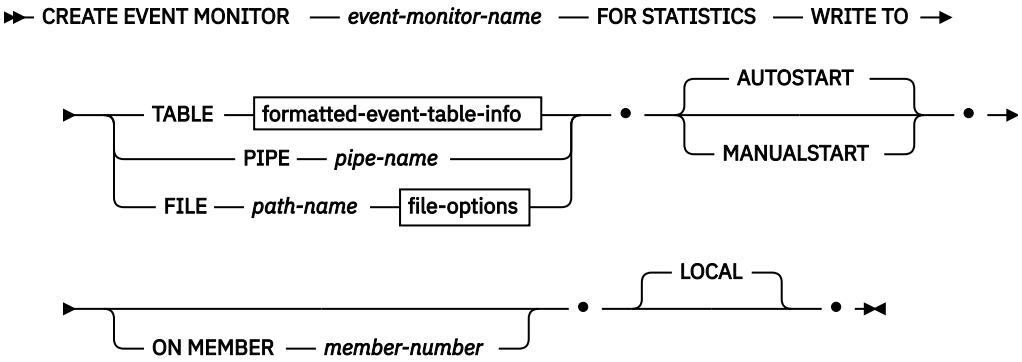
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

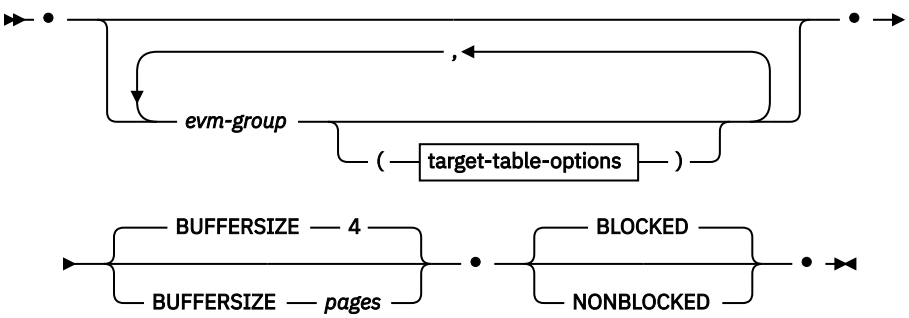
The privileges held by the authorization ID of the statement must include one of the following authorities:

- DBADM authority
- SQLADM authority

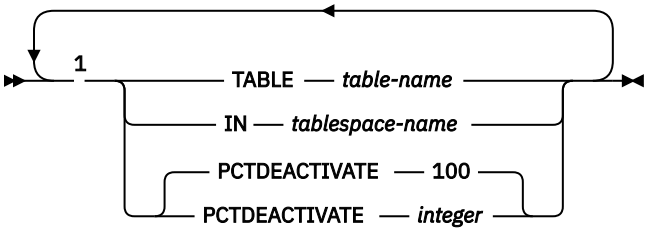
Syntax



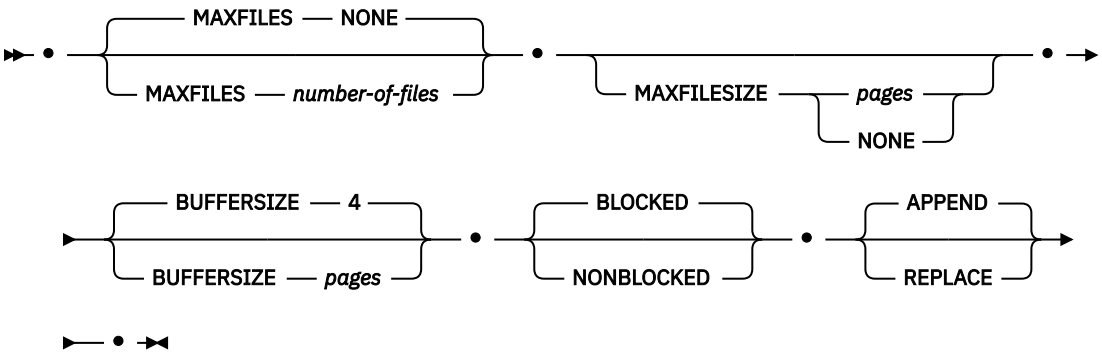
formatted-event-table-info



target-table-options



file-options



Notes:

[1] Each clause can be specified only once.
[2] Clauses can be separated with a space or a comma.

Description

event-monitor-name
Name of the event monitor. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *event-monitor-name* must not identify an event monitor that already exists in the catalog (SQLSTATE 42710).

FOR
Introduces the type of event to record.

STATISTICS
Specifies that the event monitor records a service class, workload, or work class event:

- Every *period* minutes, where *period* is the value of the **wlm_collect_int** database configuration parameter
- When the mon_collect_stats procedure is called

WRITE TO
Introduces the target for the data.

TABLE
Indicates that the target for the event monitor data is a set of database tables. The event monitor separates the data stream into one or more logical data groups and inserts each group into a separate table. Data for groups having a target table is kept, whereas data for groups not having a target table is discarded. Each monitor element contained within a group is mapped to a table column with the same name. Only elements that have a corresponding table column are inserted into the table. Other elements are discarded.

formatted-event-table-info
Defines the target tables for an event monitor. This clause should be specified for each grouping that is to be recorded. However, if no evm-group-info clauses are specified, all groups for the event monitor type are recorded.

For more information about logical data groups, see "Logical data groups and event monitor output tables" in *Database Monitoring Guide and Reference*.

evm-group
Identifies the logical data group for which a target table is being defined. The value depends upon the type of event monitor, as shown in the following table:

Type of Event Monitor	evm-group Value
Statistics	• CONTROL • HISTOGRAMBIN • OSMETRICS • QSTATS • SCMETRICS • SCSTATS • SUPERCLASSMETRICS • SUPERCLASSSTATS • WCSTATS • WLMETRICS • WLSTATS

target-table-options
Identifies the target table for the group.

TABLE *table-name*
Specifies the name of the target table. The target table must be a non-partitioned table. If the name is unqualified, the table schema defaults to the value in the CURRENT SCHEMA special register. If no name is provided, the unqualified name is derived from *evm-group* and *event-monitor-name* as follows:

```
substring(evm-group CONCAT '_'
  CONCAT event-monitor-name,1,128)
```

IN *tablespace-name*
Defines the table space in which the table is to be created. If no table space name is provided, the table space is chosen using the same process as when a table is created without a table space name using CREATE TABLE.

PCTDEACTIVATE *integer*
If a table for the event monitor is being created in an automatic storage (non-temporary) or DMS table space, the PCTDEACTIVATE parameter specifies how full the table space must be before the event monitor automatically deactivates. The specified value, which represents a percentage, can range from 0 to 100, where 100 means that the event monitor deactivates when the table space becomes completely full. The default value assumed is 100 if PCTDEACTIVATE is not specified. This option is ignored for SMS table spaces.

Important: If the target table space has auto-resize enabled, set PCTDEACTIVATE parameter to 100. Alternatively, omit this clause entirely to have the default of 100 apply. Otherwise, the event monitor might deactivate unexpectedly if the table space reaches the threshold specified by PCTDEACTIVTATE before the table space is automatically resized.

If a value for *target-table-options* is not specified, CREATE EVENT MONITOR processing proceeds as follows:

- A derived table name is used.
- A default table space is chosen.
- PCTDEACTIVATE defaults to 100.

BUFFERSIZE *pages*
Specifies the size of the event monitor buffers (in units of 4K pages). Table event monitors insert all data from a buffer, and issues a COMMIT once the buffer has been processed. The larger the buffers, the larger the commit scope used by the event monitor. Highly

active event monitors should have larger buffers than relatively inactive event monitors. When a monitor is started, two buffers of the specified size are allocated. Event monitors use double buffering to permit asynchronous I/O.

The default size of each buffer is 4 pages (two 16K buffers are allocated). The minimum size is 1 page. The maximum size of the buffers is limited by the size of the monitor heap, because the buffers are allocated from that heap. If many event monitors are being used at the same time, increase the size of the **mon_heap_sz** database manager configuration parameter.

Note: This keyword is not supported for statistics event monitors. The compiler accepts this keyword, but the keyword has no effect on the behavior of the event monitor.

BLOCKED
Specifies that each agent that generates an event should wait for an event buffer to be written out to disk if the agent determines that both event buffers are full. BLOCKED should be selected to guarantee no event data loss. This is the default option.

Note: This keyword is not supported for statistics event monitors. The compiler accepts this keyword, but the keyword has no effect for statistics event monitors. The event monitor is created as if the BLOCKED keyword was specified.

NONBLOCKED
Specifies that each agent that generates an event should not wait for the event buffer to be written out to disk if the agent determines that both event buffers are full. NONBLOCKED event monitors do not slow down database operations to the extent of BLOCKED event monitors. However, NONBLOCKED event monitors are subject to data loss on highly active systems.

PIPE
Specifies that the target for the event monitor data is a named pipe. The event monitor writes the data to the pipe in a single stream (that is, as if it were a single, infinitely long file). When writing the data to a pipe, an event monitor does not perform blocked writes. If there is no room in the pipe buffer, then the event monitor will discard the data. It is the monitoring application's responsibility to read the data promptly if it wishes to ensure no data loss.

pipe-name
The name of the pipe (FIFO on AIX) to which the event monitor will write the data.

The naming rules for pipes are platform specific.

Operating system	Naming rules
AIX	Pipe names are treated like file names. As a result, relative pipe names are permitted, and are treated like relative path-names (see description for *path-name*).
Linux	Pipe names are treated like file names. As a result, relative pipe names are permitted, and are treated like relative path-names (see description for *path-name*).
Windows	There is a special syntax for a pipe name and, as a result, absolute pipe names are required.

The existence of the pipe will not be checked at event monitor creation time. It is the responsibility of the monitoring application to have created and opened the pipe for reading at the time that the event monitor is activated. If the pipe is not available at this time, then the event monitor will turn itself off, and will log an error. (That is, if the event monitor was activated at database start time as a result of the AUTOSTART option, then the event monitor will log an error in the system error log.) If the event monitor is activated via the SET EVENT MONITOR STATE SQL statement, then that statement will fail (SQLSTATE 58030).

FILE

Indicates that the target for the event monitor data is a file (or set of files). The event monitor writes out the stream of data as a series of 8 character numbered files, with the extension "evt". (for example, 00000000.evt, 00000001.evt, and 00000002.evt). The data should be considered to be one logical file even though the data is broken up into smaller pieces (that is, the start of the data stream is the first byte in the file 00000000.evt; the end of the data stream is the last byte in the file nnnnnnnn.evt).

The maximum size of each file can be defined as well as the maximum number of files. An event monitor will never split a single event record across two files. However, an event monitor may write related records in two different files. It is the responsibility of the application that uses this data to keep track of such related information when processing the event files.

path-name

The name of the directory in which the event monitor should write the event files data. The path must be known at the server; however, the path itself could reside on another database partition (for example, an NFS mounted file). A string constant must be used when specifying the *path-name*.

The directory does not have to exist at CREATE EVENT MONITOR time. However, a check is made for the existence of the target path when the event monitor is activated. At that time, if the target path does not exist, an error (SQLSTATE 428A3) is raised.

If an absolute path is specified, the specified path will be the one used.

In a Db2 pureScale environment, if a relative path is specified, then the path relative to the database owning directory in the database directory will be used.

In environments other than Db2 pureScale, if a relative path (a path that does not start with the root) is specified, then the path relative to the DB2EVENT directory in the database directory will be used.

It is possible to specify two or more event monitors that have the same target path. However, once one of the event monitors has been activated for the first time, and as long as the target directory is not empty, it will be impossible to activate any of the other event monitors.

file-options

Specifies the options for the file format.

MAXFILES NONE

Specifies that there is no limit to the number of event files that the event monitor will create. This is the default.

MAXFILES *number-of-files*

Specifies that there is a limit on the number of event monitor files that will exist for a particular event monitor at any time. Whenever an event monitor has to create another file, it will check to make sure that the number of .evt files in the directory is less than *number-of-files*. If this limit has already been reached, then the event monitor will turn itself off.

If an application removes the event files from the directory after they have been written, then the total number of files that an event monitor can produce can exceed *number-of-files*. This option has been provided to allow a user to guarantee that the event data will not consume more than a specified amount of disk space.

MAXFILESIZE *pages*

Specifies that there is a limit to the size of each event monitor file. Whenever an event monitor writes a new event record to a file, it checks that the file will not grow to be greater than *pages* (in units of 4K pages). If the resulting file would be too large, then the event monitor switches to the next file. The default for this option is:

- Linux - 1000 4K pages
- UNIX - 1000 4K pages
- Windows - 200 4K pages

The number of pages must be greater than at least the size of the event buffer in pages. If this requirement is not met, then an error (SQLSTATE 428A4) is raised.

MAXFILESIZE NONE
Specifies that there is no set limit on a file's size. If MAXFILESIZE NONE is specified, then MAXFILES 1 must also be specified. This option means that one file will contain all of the event data for a particular event monitor. In this case the only event file will be 00000000.evt.

BUFFERSIZE *pages*
Specifies the size of the event monitor buffers (in units of 4K pages). All event monitor file I/O is buffered to improve the performance of the event monitors. The larger the buffers, the less I/O will be performed by the event monitor. Highly active event monitors should have larger buffers than relatively inactive event monitors. When the monitor is started, two buffers of the specified size are allocated. Event monitors use double buffering to permit asynchronous I/O.

The default size of each buffer is 4 pages (two 16K buffers are allocated). The minimum size is 1 page. The maximum size of the buffers is limited by the value of the MAXFILESIZE parameter, as well as the size of the monitor heap, because the buffers are allocated from that heap. If many event monitors are being used at the same time, increase the size of the **mon_heap_sz** database manager configuration parameter.

Event monitors that write their data to a pipe also have two internal (non-configurable) buffers that are each 1 page in size. These buffers are also allocated from the monitor heap (MON_HEAP). For each active event monitor that has a pipe target, increase the size of the database heap by 2 pages.

Note: This keyword is not supported for statistics event monitors. The compiler accepts this keyword, but the keyword has no effect on the behavior of the event monitor.

BLOCKED
Specifies that each agent that generates an event should wait for an event buffer to be written out to disk if the agent determines that both event buffers are full. BLOCKED should be selected to guarantee no event data loss. This is the default option.

NONBLOCKED
Specifies that each agent that generates an event should not wait for the event buffer to be written out to disk if the agent determines that both event buffers are full. NONBLOCKED event monitors do not slow down database operations to the extent of BLOCKED event monitors. However, NONBLOCKED event monitors are subject to data loss on highly active systems.

Note: This keyword is not supported for statistics event monitors. The compiler accepts this keyword, but the keyword has no effect for statistics event monitors. The event monitor is created as if the BLOCKED keyword was specified.

APPEND
Specifies that if event data files already exist when the event monitor is turned on, then the event monitor will append the new event data to the existing stream of data files. When the event monitor is reactivated, it will resume writing to the event files as if it had never been turned off. APPEND is the default option.

The APPEND option does not apply at CREATE EVENT MONITOR time, if there is existing event data in the directory where the newly created event monitor is to write its event data.

REPLACE
Specifies that if event data files already exist when the event monitor is turned on, then the event monitor will erase all of the event files and start writing data to file 00000000.evt.

MANUALSTART
Specifies that the event monitor must be activated manually using the SET EVENT MONITOR STATE statement. After a MANUALSTART event monitor has been activated, it can be deactivated only by using the SET EVENT MONITOR STATE statement or by stopping the instance.

AUTOSTART
Specifies that the event monitor is to be automatically activated whenever the member on which the event monitor runs is activated. This is the default behavior of the statistics event monitor.

ON MEMBER *member-number*
Specifies the member on which a file or pipe event monitor is to run. When the monitoring scope is defined as LOCAL, data is collected only on the specified member. The I/O component will physically run on the specified member, writing records to the specified file or pipe.

When Db2 pureScale feature is enabled, -1 is the default.

If -1 is specified, it allows the I/O component to run from any active member. Additionally, in the event that the I/O component is no longer able to run on a given member, the event monitor will be restarted with the I/O component running on another available active member.

This clause is not valid for table event monitors. In a partitioned database environment, write-to-table event monitors will run and write events on all database partitions where table spaces for target tables are defined.

In a Db2 pureScale environment, write-to-table event monitors will record events on all active members.

If this clause is not specified and the Db2 pureScale feature is not enabled, the currently connected database partition number (for the application) is used.

If this clause is not specified and Db2 pureScale is enabled, the I/O component is able to run on any currently connected member.

LOCAL
The event monitor reports only on the member that is running. It gives a partial trace of the database activity. This is the default.

This clause is valid for file or pipe monitors. It is not valid for table event monitors.

GLOBAL is not a valid scope for this type of event monitor.

Rules

- The STATISTICS event type cannot be combined with any other event types in a particular event monitor definition.

Notes

- Event monitor definitions are recorded in the SYSCAT.EVENTMONITORS catalog view. The events themselves are recorded in the SYSCAT.EVENTS catalog view. The names of target tables are recorded in the SYSCAT.EVENTTABLES catalog view.
- If the member on which the event monitor is to run is not active, event monitor activation occurs when that member next activates.
- After an event monitor is activated, it behaves like an autostart event monitor until that event monitor is explicitly deactivated or the instance is recycled. That is, if an event monitor is active when a member is deactivated, and that member is subsequently reactivated, the event monitor is also explicitly reactivated.
- If you create the event monitor such that the logical data groups event_scstats or event_wlstats are included in the event monitor output, metrics are reported in two XML documents contained in the event monitor output. The monitor elements reported in the `metrics` document show the change in value for the monitor elements since the last time statistics were collected. The elements reported in

details_xml are the same monitor elements, however, they show the values since the database was activated. That is, they continue to increase until the database is deactivated.

Important: The XML document details_xml is deprecated in the statistics event monitor, and might be removed in a future release. For more information, see "Reporting of metrics in details_xml by the statistics event monitor has been deprecated" at http://www.ibm.com/support/knowledgecenter/SSEPGG_10.1.0/com.ibm.db2.luw.wn.doc/doc/i0060390.html.

- ***Write to table event monitors***: General notes:
  - All target tables are created when the CREATE EVENT MONITOR statement executes.
  - If the creation of a table fails for any reason, an error is passed back to the application program, and the CREATE EVENT MONITOR statement fails.
  - A target table can only be used by one event monitor. During CREATE EVENT MONITOR processing, if a target table is found to have already been defined for use by another event monitor, the CREATE EVENT MONITOR statement fails, and an error is passed back to the application program. A table is defined for use by another event monitor if the table name matches a value found in the SYSCAT.EVENTTABLES catalog view.
  - During CREATE EVENT MONITOR processing, if a table already exists, but is *not* defined for use by another event monitor, no table is created, and processing continues. A warning is passed back to the application program.
  - Any table spaces must exist before the CREATE EVENT MONITOR statement is executed. The CREATE EVENT MONITOR statement does not create table spaces.
  - If specified, the LOCAL and GLOBAL keywords are ignored. With WRITE TO TABLE event monitors, an event monitor output process or thread is started on each member in the instance, and each of these processes reports data only for the member on which it is running.
  - The following event types from the flat monitor log file or pipe format are not recorded by write to table event monitors:
    - LOG_STREAM_HEADER
    - LOG_HEADER
    - DB_HEADER (Elements db_name and db_path are not recorded. The element conn_time is recorded in CONTROL.)
  - In a partitioned database environment, data is only written to target tables on the database partitions where their table spaces exist. If a table space for a target table does not exist on some database partition, data for that target table is ignored. This behavior allows users to choose a subset of member for monitoring, by creating a table space that exists only on certain member.

    In a Db2 pureScale environment, data will be written from every member.

    In a partitioned database environment, if some target tables do not reside on a database partition, but other target tables do reside on that same database partition, only the data for the target tables that do reside on that database partition is recorded.
  - Users must manually prune all target tables.

  Table Columns:

  - Column names in a table match an event monitor element identifier. Any event monitor element that does not have a corresponding target table column is ignored.
  - Use the db2evtbl command to build a CREATE EVENT MONITOR command that includes a complete list of elements for a group.
  - The types of columns being used for monitor elements correlate to the following mapping:

```
SQLM_TYPE_STRING                    CHAR[n], VARCHAR[n] or CLOB(n)
                                    (If the data in the event monitor
                                    record exceeds n bytes,
                                    it is truncated.)
SQLM_TYPE_U8BIT and SQLM_TYPE_8BIT   SMALLINT, INTEGER or BIGINT
SQLM_TYPE_16BIT and SQLM_TYPE_U16BIT SMALLINT, INTEGER or BIGINT
SQLM_TYPE_32BIT and SQLM_TYPE_U32BIT INTEGER or BIGINT
```

```
SQLM_TYPE_U64BIT and SQLM_TYPE_64BIT BIGINT
sqlm_timestamp                       TIMESTAMP
sqlm_time(elapsed time)              BIGINT
sqlca:
  sqlerrmc                           VARCHAR[72]
  sqlstate                           CHAR[5]
  sqlwarn                            CHAR[11]
  other fields                       INTEGER or BIGINT
```

- Columns are defined to be NOT NULL.
- Unlike other target tables, the columns in the CONTROL table do not match monitor element identifiers. Columns are defined as follows:

Column Name	Data Type	Nullable	Description
PARTITION_KEY	INTEGER	N	Distribution key (partitioned database only)
PARTITION_NUMBER	INTEGER	N	Database partition number (partitioned database only)
EVMONNAME	VARCHAR(128)	N	Name of the event monitor
MESSAGE	VARCHAR(128)	N	Describes the nature of the MESSAGE_TIME column. For more details see, "message - Control Table Message monitor element" in the *Database Monitoring Guide and Reference*
MESSAGE_TIME	TIMESTAMP	N	Timestamp

- In a partitioned database environment, the first column of each table is named PARTITION_KEY, is NOT NULL, and is of type INTEGER. This column is used as the distribution key for the table. The value of this column is chosen so that each event monitor process inserts data into the member on which the process is running; that is, insert operations are performed locally on the member where the event monitor process is running. On any database partition, the PARTITION_KEY field will contain the same value. This means that if a database partition is dropped and data redistribution is performed, all data on the dropped database partition will go to one other database partition instead of being evenly distributed. Therefore, before removing a database partition, consider deleting all table rows on that database partition.
- In a partitioned database environment, a column named PARTITION_NUMBER can be defined for each table. This column is NOT NULL and is of type INTEGER. It contains the number of the database partition on which the data was inserted. Unlike the PARTITION_KEY column, the PARTITION_NUMBER column is not mandatory. The PARTITION_NUMBER column is not allowed in a non-partitioned database environment.

Table Attributes:

- Default table attributes are used. Besides distribution key (partitioned databases only), no extra options are specified when creating tables.
- Indexes on the table can be created.
- Extra table attributes (such as volatile, RI, triggers, constraints, and so on) can be added, but the event monitor process (or thread) will ignore them.

– If "not logged initially" is added as a table attribute, it is turned off at the first COMMIT, and is not set back on.

Event Monitor Activation:

– When an event monitor activates, all target table names are retrieved from the SYSCAT.EVENTTABLES catalog view.
– In a partitioned database environment, activation processing occurs on every member of the instance. On a particular member, activation processing determines the table spaces and database partition groups for each target table. The event monitor only activates on a member if at least one target table exists on that database partition. Moreover, if some target table is not found on a database partition, that target table is flagged so that data destined for that table is dropped during runtime processing.
– If a target table does not exist when the event monitor activates (or, in a partitioned database environment, if the table space does not reside on a database partition), activation continues, and data that would otherwise be inserted into this table is ignored.
– Activation processing validates each target table. If validation fails, activation of the event monitor fails, and messages are written to the administration log.
– During activation in a partitioned database environment, the CONTROL table rows for FIRST_CONNECT and EVMON_START are only inserted on the catalog database partition. This requires that the table space for the control table exist on the catalog database partition. If it does not exist on the catalog database partition, these inserts are not performed.
– In a partitioned database environment, if a member is not yet active when a write to table event monitor is activated, the event monitor will be activated the next time that member is activated.

Run Time:

– An event monitor runs with DATAACCESS authority.
– If, while an event monitor is active, an insert operation into a target table fails:
  - Uncommitted changes are rolled back.
  - A message is written to the administration log.
  - The event monitor is deactivated.
– If an event monitor is active, it performs a local COMMIT when it has finished processing an event monitor buffer.
– In an environment other than a partitioned database or a Db2 pureScale environment, all write to table event monitors are deactivated when the last application terminates (and the database has not been explicitly activated).

  In a Db2 pureScale environment, write to table event monitors are deactivated on a given member when the member stops and is reactivated when the member restarts.

  In a partitioned database environment, write to table event monitors are deactivated when the catalog partition deactivates.
– The DROP EVENT MONITOR statement does not drop target tables.
– Whenever a write-to-table event monitor activates, it will acquire IN table locks on each target table in order to prevent them from being modified while the event monitor is active. Table locks are maintained on all tables while the event monitor is active. If exclusive access is required on any of the target tables (for example, when a utility is to be run), first deactivate the event monitor to release the table locks before attempting such access.

- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  – DBPARTITIONNUM or NODE can be specified in place of MEMBER, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
  – Commas can be used to separate multiple options in the *target-table-options* clause

Example

Define a statistics event monitor named DBSTATISTICS

```
CREATE EVENT MONITOR DBSTATISTICS
    FOR STATISTICS
    WRITE TO TABLE
    SCSTATS (TABLE SCSTATS_DBSTATISTICS
             IN USERSPACE1
             PCTDEACTIVATE 100),
    SCMETRICS (TABLE SCMETRICS_DBSTATISTICS
             IN USERSPACE1
             PCTDEACTIVATE 100),
    SUPERCLASSSTATS (TABLE SUPERCLASSTATS_DBSTATISTICS
             IN USERSPACE1
             PCTDEACTIVATE 100),
    SUPERCLASSMETRICS (TABLE SUPERCLASSMETRICS_DBSTATISTICS
             IN USERSPACE1
             PCTDEACTIVATE 100),
    WCSTATS (TABLE WCSTATS_DBSTATISTICS
             IN USERSPACE1
             PCTDEACTIVATE 100),
    WLSTATS (TABLE WLSTATS_DBSTATISTICS
             IN USERSPACE1
             PCTDEACTIVATE 100),
    QSTATS (TABLE QSTATS_DBSTATISTICS
             IN USERSPACE1
             PCTDEACTIVATE 100),
    HISTOGRAMBIN (TABLE HISTOGRAMBIN_DBSTATISTICS
             IN USERSPACE1
             PCTDEACTIVATE 100),
    OSMETRICS (TABLE OSMETRICS_DBSTATISTICS
             IN USERSPACE1
             PCTDEACTIVATE 100),
    CONTROL (TABLE CONTROL_DBSTATISTICS
             IN USERSPACE1
             PCTDEACTIVATE 100)
    AUTOSTART;
```

CREATE EVENT MONITOR (threshold violations)

The CREATE EVENT MONITOR (threshold violations) statement defines a monitor that will record threshold violation events that occur when using the database. The definition of the threshold violations event monitor also specifies where the database should record the events.

Invocation

The threshold violations event monitor can collect more information about the application that violated the threshold. The addition of these monitor elements does not affect existing threshold violations event monitors, but in order to collect the additional application information existing monitors must dropped and recreated.

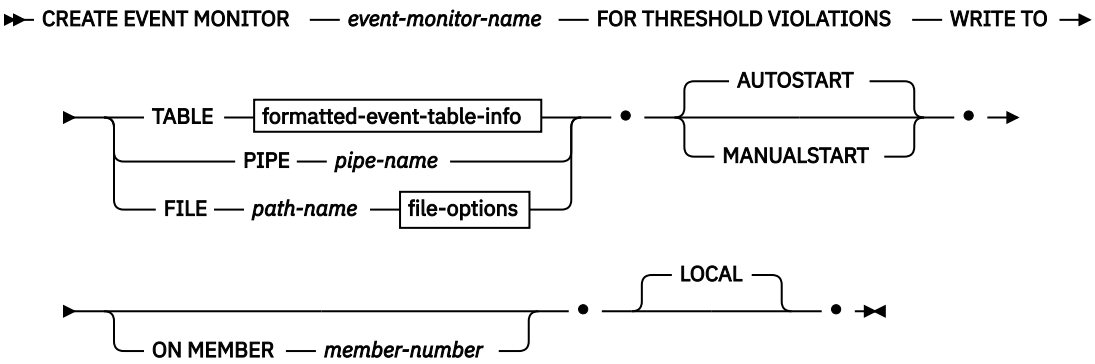
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

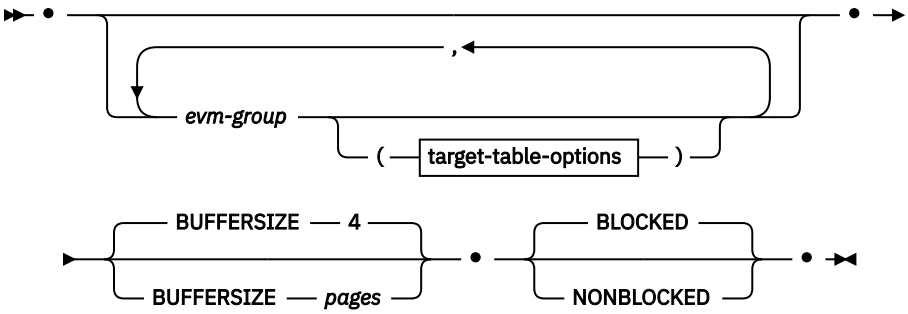
The privileges held by the authorization ID of the statement must include one of the following authorities:

- DBADM authority
- SQLADM authority

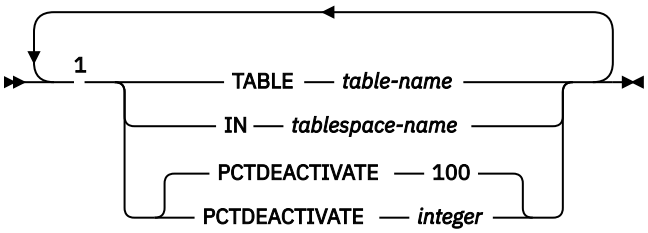
Syntax



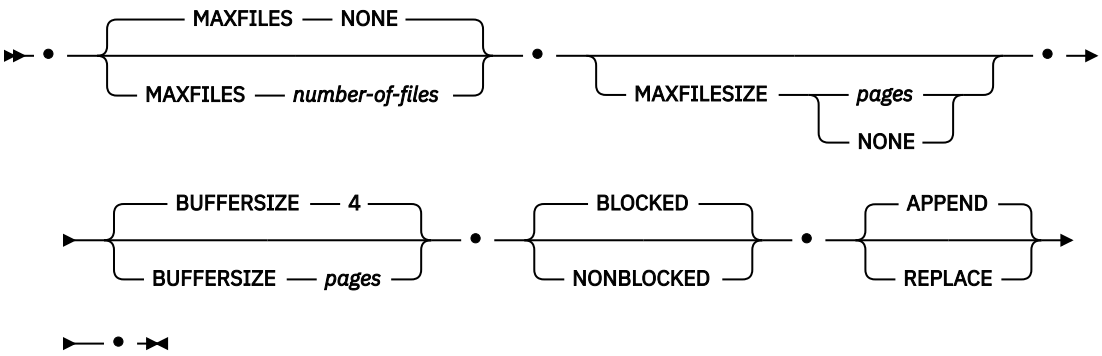
formatted-event-table-info



target-table-options



file-options



Notes:

[1] Each clause can be specified only once.
[2] Clauses can be separated with a space or a comma.

Description

event-monitor-name
Name of the event monitor. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *event-monitor-name* must not identify an event monitor that already exists in the catalog (SQLSTATE 42710).

FOR
Introduces the type of event to record.

THRESHOLD VIOLATIONS
Specifies that the event monitor records a threshold violation event when a threshold is violated. Such events can be recorded at any point in the life of an activity, not just at completion.

WRITE TO
Introduces the target for the data.

TABLE
Indicates that the target for the event monitor data is a set of database tables. The event monitor separates the data stream into one or more logical data groups and inserts each group into a separate table. Data for groups having a target table is kept, whereas data for groups not having a target table is discarded. Each monitor element contained within a group is mapped to a table column with the same name. Only elements that have a corresponding table column are inserted into the table. Other elements are discarded.

formatted-event-table-info
Defines the target tables for an event monitor. This clause should be specified for each grouping that is to be recorded. However, if no evm-group-info clauses are specified, all groups for the event monitor type are recorded.

For more information about logical data groups, see "Logical data groups and event monitor output tables" in *Database Monitoring Guide and Reference*.

evm-group
Identifies the logical data group for which a target table is being defined. The value depends upon the type of event monitor, as shown in the following table:

Type of Event Monitor	evm-group Value
Threshold violations	• THRESHOLDVIOLATIONS • CONTROL

target-table-options
Identifies the target table for the group.

TABLE *table-name*
Specifies the name of the target table. The target table must be a non-partitioned table. If the name is unqualified, the table schema defaults to the value in the CURRENT SCHEMA special register. If no name is provided, the unqualified name is derived from *evm-group* and *event-monitor-name* as follows:

```
substring(evm-group CONCAT '_'
  CONCAT event-monitor-name,1,128)
```

IN *tablespace-name*
Defines the table space in which the table is to be created. If no table space name is provided, the table space is chosen using the same process as when a table is created without a table space name using CREATE TABLE.

PCTDEACTIVATE *integer*
If a table for the event monitor is being created in an automatic storage (non-temporary) or DMS table space, the PCTDEACTIVATE parameter specifies how full the table space must be before the event monitor automatically deactivates. The specified value, which represents a percentage, can range from 0 to 100, where

100 means that the event monitor deactivates when the table space becomes completely full. The default value assumed is 100 if PCTDEACTIVATE is not specified. This option is ignored for SMS table spaces.

Important: If the target table space has auto-resize enabled, set PCTDEACTIVATE parameter to 100. Alternatively, omit this clause entirely to have the default of 100 apply. Otherwise, the event monitor might deactivate unexpectedly if the table space reaches the threshold specified by PCTDEACTIVTATE before the table space is automatically resized.

If a value for *target-table-options* is not specified, CREATE EVENT MONITOR processing proceeds as follows:

- A derived table name is used.
- A default table space is chosen.
- The PCTDEACTIVATE parameter defaults to 100.

BUFFERSIZE ***pages***
Specifies the size of the event monitor buffers (in units of 4K pages). Table event monitors insert all data from a buffer, and issues a COMMIT once the buffer has been processed. The larger the buffers, the larger the commit scope used by the event monitor. Highly active event monitors should have larger buffers than relatively inactive event monitors. When a monitor is started, two buffers of the specified size are allocated. Event monitors use double buffering to permit asynchronous I/O.

The default size of each buffer is 4 pages (two 16K buffers are allocated). The minimum size is 1 page. The maximum size of the buffers is limited by the size of the monitor heap, because the buffers are allocated from that heap. If many event monitors are being used at the same time, increase the size of the **mon_heap_sz** database manager configuration parameter.

Note: This keyword is not supported for threshold violation event monitors. The compiler accepts this keyword, but the keyword has no effect on the behavior of the event monitor.

BLOCKED
Specifies that each agent that generates an event should wait for an event buffer to be written out to disk if the agent determines that both event buffers are full. BLOCKED should be selected to guarantee no event data loss. This is the default option.

NONBLOCKED
Specifies that each agent that generates an event should not wait for the event buffer to be written out to disk if the agent determines that both event buffers are full. NONBLOCKED event monitors do not slow down database operations to the extent of BLOCKED event monitors. However, NONBLOCKED event monitors are subject to data loss on highly active systems.

Note: This keyword is not supported for threshold violation event monitors. The compiler accepts this keyword, but the keyword has no effect for threshold violation event monitors. The event monitor is created as if the BLOCKED keyword was specified.

PIPE
Specifies that the target for the event monitor data is a named pipe. The event monitor writes the data to the pipe in a single stream (that is, as if it were a single, infinitely long file). When writing the data to a pipe, an event monitor does not perform blocked writes. If there is no room in the pipe buffer, then the event monitor will discard the data. It is the monitoring application's responsibility to read the data promptly if it wishes to ensure no data loss.

pipe-name
The name of the pipe (FIFO on AIX) to which the event monitor will write the data.

The naming rules for pipes are platform specific.

Operating system	Naming rules
AIX	Pipe names are treated like file names. As a result, relative pipe names are permitted, and are treated like relative path-names (see description for *path-name*).
Linux	Pipe names are treated like file names. As a result, relative pipe names are permitted, and are treated like relative path-names (see description for *path-name*).
Windows	There is a special syntax for a pipe name and, as a result, absolute pipe names are required.

The existence of the pipe will not be checked at event monitor creation time. It is the responsibility of the monitoring application to have created and opened the pipe for reading at the time that the event monitor is activated. If the pipe is not available at this time, then the event monitor will turn itself off, and will log an error. (That is, if the event monitor was activated at database start time as a result of the AUTOSTART option, then the event monitor will log an error in the system error log.) If the event monitor is activated via the SET EVENT MONITOR STATE SQL statement, then that statement will fail (SQLSTATE 58030).

FILE

Indicates that the target for the event monitor data is a file (or set of files). The event monitor writes out the stream of data as a series of 8 character numbered files, with the extension "evt". (for example, 00000000.evt, 00000001.evt, and 00000002.evt). The data should be considered to be one logical file even though the data is broken up into smaller pieces (that is, the start of the data stream is the first byte in the file 00000000.evt; the end of the data stream is the last byte in the file nnnnnnnn.evt).

The maximum size of each file can be defined as well as the maximum number of files. An event monitor will never split a single event record across two files. However, an event monitor may write related records in two different files. It is the responsibility of the application that uses this data to keep track of such related information when processing the event files.

path-name

The name of the directory in which the event monitor should write the event files data. The path must be known at the server; however, the path itself could reside on another database partition (for example, an NFS mounted file). A string constant must be used when specifying the *path-name*.

The directory does not have to exist at CREATE EVENT MONITOR time. However, a check is made for the existence of the target path when the event monitor is activated. At that time, if the target path does not exist, an error (SQLSTATE 428A3) is raised.

If an absolute path is specified, the specified path will be the one used.

In environments other than Db2 pureScale, if a relative path (a path that does not start with the root) is specified, then the path relative to the DB2EVENT directory in the database directory will be used.

In a Db2 pureScale environment, if a relative path is specified, then the path relative to the database owning directory in the database directory will be used.

It is possible to specify two or more event monitors that have the same target path. However, once one of the event monitors has been activated for the first time, and as long as the target directory is not empty, it will be impossible to activate any of the other event monitors.

file-options

Specifies the options for the file format.

MAXFILES NONE
Specifies that there is no limit to the number of event files that the event monitor will create. This is the default.

MAXFILES *number-of-files*
Specifies that there is a limit on the number of event monitor files that will exist for a particular event monitor at any time. Whenever an event monitor has to create another file, it will check to make sure that the number of .evt files in the directory is less than *number-of-files*. If this limit has already been reached, then the event monitor will turn itself off.

If an application removes the event files from the directory after they have been written, then the total number of files that an event monitor can produce can exceed *number-of-files*. This option has been provided to allow a user to guarantee that the event data will not consume more than a specified amount of disk space.

MAXFILESIZE *pages*
Specifies that there is a limit to the size of each event monitor file. Whenever an event monitor writes a new event record to a file, it checks that the file will not grow to be greater than *pages* (in units of 4K pages). If the resulting file would be too large, then the event monitor switches to the next file. The default for this option is:

- Linux - 1000 4K pages
- UNIX - 1000 4K pages
- Windows - 200 4K pages

The number of pages must be greater than at least the size of the event buffer in pages. If this requirement is not met, then an error (SQLSTATE 428A4) is raised.

MAXFILESIZE NONE
Specifies that there is no set limit on a file's size. If MAXFILESIZE NONE is specified, then MAXFILES 1 must also be specified. This option means that one file will contain all of the event data for a particular event monitor. In this case the only event file will be 00000000.evt.

BUFFERSIZE *pages*
Specifies the size of the event monitor buffers (in units of 4K pages). All event monitor file I/O is buffered to improve the performance of the event monitors. The larger the buffers, the less I/O will be performed by the event monitor. Highly active event monitors should have larger buffers than relatively inactive event monitors. When the monitor is started, two buffers of the specified size are allocated. Event monitors use double buffering to permit asynchronous I/O.

The default size of each buffer is 4 pages (two 16K buffers are allocated). The minimum size is 1 page. The maximum size of the buffers is limited by the value of the MAXFILESIZE parameter, as well as the size of the monitor heap, because the buffers are allocated from that heap. If many event monitors are being used at the same time, increase the size of the **mon_heap_sz** database manager configuration parameter.

Event monitors that write their data to a pipe also have two internal (non-configurable) buffers that are each 1 page in size. These buffers are also allocated from the monitor heap (MON_HEAP). For each active event monitor that has a pipe target, increase the size of the database heap by 2 pages.

Note: This keyword is not supported for threshold violation event monitors. The compiler accepts this keyword, but the keyword has no effect on the behavior of the event monitor.

BLOCKED
Specifies that each agent that generates an event should wait for an event buffer to be written out to disk if the agent determines that both event buffers are full. BLOCKED should be selected to guarantee no event data loss. This is the default option.

NONBLOCKED
Specifies that each agent that generates an event should not wait for the event buffer to be written out to disk if the agent determines that both event buffers are full. NONBLOCKED event monitors do not slow down database operations to the extent of BLOCKED event monitors. However, NONBLOCKED event monitors are subject to data loss on highly active systems.

Note: This keyword is not supported for threshold violation event monitors. The compiler accepts this keyword, but the keyword has no effect for threshold violation event monitors. The event monitor is created as if the BLOCKED keyword was specified.

APPEND
Specifies that if event data files already exist when the event monitor is turned on, then the event monitor will append the new event data to the existing stream of data files. When the event monitor is reactivated, it will resume writing to the event files as if it had never been turned off. APPEND is the default option.

The APPEND option does not apply at CREATE EVENT MONITOR time, if there is existing event data in the directory where the newly created event monitor is to write its event data.

REPLACE
Specifies that if event data files already exist when the event monitor is turned on, then the event monitor will erase all of the event files and start writing data to file 00000000.evt.

MANUALSTART
Specifies that the event monitor must be activated manually using the SET EVENT MONITOR STATE statement. After a MANUALSTART event monitor has been activated, it can be deactivated only by using the SET EVENT MONITOR STATE statement or by stopping the instance.

AUTOSTART
Specifies that the event monitor is to be automatically activated whenever the database partition on which the event monitor runs is activated. This is the default behavior of the threshold violations event monitor.

ON MEMBER *member-number*
Specifies the member on which a file or pipe event monitor is to run. When the monitoring scope is defined as LOCAL, data is collected only on the member. The I/O component will physically run on the specified member, writing records to the specified file or pipe.

When the Db2 pureScale is enabled, -1 is the default.

If the value is -1, it allows the I/O component to run from any active member. Additionally, in the event that the I/O component is no longer able to run on a given member, the event monitor will be restarted with the I/O component running on another available active member.

This clause is not valid for table event monitors. In a partitioned database environment, write-to-table event monitors will run and write events on all database partitions where table spaces for target tables are defined.

In a Db2 pureScale environment, write-to-table event monitors will record events on all active members.

If this clause is not specified and Db2 pureScale is not enabled, the currently connected member is used.

If this clause is not specified and Db2 pureScale is enabled, the I/O component is able to run on any currently connected member.

LOCAL
The event monitor reports only on the member that is running. It gives a partial trace of the database activity. This is the default.

This clause is valid for file or pipe monitors. It is not valid for table event monitors.

GLOBAL is not a valid scope for this type of event monitor.

Rules

- The THRESHOLD VIOLATIONS event type cannot be combined with any other event types in a particular event monitor definition.

Notes

- Event monitor definitions are recorded in the SYSCAT.EVENTMONITORS catalog view. The events themselves are recorded in the SYSCAT.EVENTS catalog view. The names of target tables are recorded in the SYSCAT.EVENTTABLES catalog view.
- If the member on which the event monitor is to run is not active, event monitor activation occurs when that member next activates.
- After an event monitor is activated, it behaves like an autostart event monitor until that event monitor is explicitly deactivated or the instance is recycled. That is, if an event monitor is active when a member is deactivated, and that member is subsequently reactivated, the event monitor is also explicitly reactivated.
- ***Write to table event monitors***: General notes:
  - All target tables are created when the CREATE EVENT MONITOR statement executes.
  - If the creation of a table fails for any reason, an error is passed back to the application program, and the CREATE EVENT MONITOR statement fails.
  - A target table can only be used by one event monitor. During CREATE EVENT MONITOR processing, if a target table is found to have already been defined for use by another event monitor, the CREATE EVENT MONITOR statement fails, and an error is passed back to the application program. A table is defined for use by another event monitor if the table name matches a value found in the SYSCAT.EVENTTABLES catalog view.
  - During CREATE EVENT MONITOR processing, if a table already exists, but is *not* defined for use by another event monitor, no table is created, and processing continues. A warning is passed back to the application program.
  - Any table spaces must exist before the CREATE EVENT MONITOR statement is executed. The CREATE EVENT MONITOR statement does not create table spaces.
  - If specified, the LOCAL and GLOBAL keywords are ignored. With WRITE TO TABLE event monitors, an event monitor output process or thread is started on each member in the instance, and each of these processes reports data only for the member on which it is running.
  - The following event types from the flat monitor log file or pipe format are not recorded by write to table event monitors:
    - LOG_STREAM_HEADER
    - LOG_HEADER
    - DB_HEADER (Elements db_name and db_path are not recorded. The element conn_time is recorded in CONTROL.)
  - In a partitioned database environment, data is only written to target tables on the database partitions where their table spaces exist. If a table space for a target table does not exist on some database partition, data for that target table is ignored. This behavior allows users to choose a subset of database partitions for monitoring, by creating a table space that exists only on certain database partitions.

    In a Db2 pureScale environment, data will be written from every member.

    In a partitioned database environment, if some target tables do not reside on a database partition, but other target tables do reside on that same database partition, only the data for the target tables that do reside on that database partition is recorded.
  - Users must manually prune all target tables.

Table Columns:

- Column names in a table match an event monitor element identifier. Any event monitor element that does not have a corresponding target table column is ignored.
- Use the db2evtbl command to build a CREATE EVENT MONITOR command that includes a complete list of elements for a group.
- The types of columns being used for monitor elements correlate to the following mapping:

```
SQLM_TYPE_STRING                    CHAR[n], VARCHAR[n] or CLOB(n)
                                    (If the data in the event monitor
                                    record exceeds n bytes,
                                    it is truncated.)
SQLM_TYPE_U8BIT and SQLM_TYPE_8BIT  SMALLINT, INTEGER or BIGINT
SQLM_TYPE_16BIT and SQLM_TYPE_U16BIT SMALLINT, INTEGER or BIGINT
SQLM_TYPE_32BIT and SQLM_TYPE_U32BIT INTEGER or BIGINT
SQLM_TYPE_U64BIT and SQLM_TYPE_64BIT BIGINT
sqlm_timestamp                      TIMESTAMP
sqlm_time(elapsed time)             BIGINT
sqlca:
  sqlerrmc                          VARCHAR[72]
  sqlstate                          CHAR[5]
  sqlwarn                           CHAR[11]
  other fields                      INTEGER or BIGINT
```

- Columns are defined to be NOT NULL.
- Unlike other target tables, the columns in the CONTROL table do not match monitor element identifiers. Columns are defined as follows:

Column Name	Data Type	Nullable	Description
PARTITION_KEY	INTEGER	N	Distribution key (partitioned database only)
PARTITION_NUMBER	INTEGER	N	Database partition number (partitioned database only)
EVMONNAME	VARCHAR(128)	N	Name of the event monitor
MESSAGE	VARCHAR(128)	N	Describes the nature of the MESSAGE_TIME column. For more details see, "message - Control Table Message monitor element" in the *Database Monitoring Guide and Reference*
MESSAGE_TIME	TIMESTAMP	N	Timestamp

- In a partitioned database environment, the first column of each table is named PARTITION_KEY, is NOT NULL, and is of type INTEGER. This column is used as the distribution key for the table. The value of this column is chosen so that each event monitor process inserts data into the database partition on which the process is running; that is, insert operations are performed locally on the database partition where the event monitor process is running. On any database partition, the PARTITION_KEY field will contain the same value. This means that if a database partition is dropped and data redistribution is performed, all data on the dropped database partition will go to one other database partition instead of being evenly distributed. Therefore, before removing a database partition, consider deleting all table rows on that database partition.

– In a partitioned database environment, a column named PARTITION_NUMBER can be defined for each table. This column is NOT NULL and is of type INTEGER. It contains the number of the database partition on which the data was inserted. Unlike the PARTITION_KEY column, the PARTITION_NUMBER column is not mandatory. The PARTITION_NUMBER column is not allowed in a non-partitioned database environment.

Table Attributes:

– Default table attributes are used. Besides distribution key (partitioned databases only), no extra options are specified when creating tables.
– Indexes on the table can be created.
– Extra table attributes (such as volatile, RI, triggers, constraints, and so on) can be added, but the event monitor process (or thread) will ignore them.
– If "not logged initially" is added as a table attribute, it is turned off at the first COMMIT, and is not set back on.

Event Monitor Activation:

– When an event monitor activates, all target table names are retrieved from the SYSCAT.EVENTTABLES catalog view.
– In a partitioned database environment, activation processing occurs on every database partition of the instance. On a particular database partition, activation processing determines the table spaces and database partition groups for each target table. The event monitor only activates on a database partition if at least one target table exists on that database partition. Moreover, if some target table is not found on a database partition, that target table is flagged so that data destined for that table is dropped during runtime processing.
– If a target table does not exist when the event monitor activates (or, in a partitioned database environment, if the table space does not reside on a database partition), activation continues, and data that would otherwise be inserted into this table is ignored.
– Activation processing validates each target table. If validation fails, activation of the event monitor fails, and messages are written to the administration log.
– During activation in a partitioned database environment, the CONTROL table rows for FIRST_CONNECT and EVMON_START are only inserted on the catalog database partition. This requires that the table space for the control table exist on the catalog database partition. If it does not exist on the catalog database partition, these inserts are not performed.
– In a partitioned database environment, if a partition is not yet active when a write to table event monitor is activated, the event monitor will be activated the next time that partition is activated.

Run Time:

– An event monitor runs with DATAACCESS authority.
– If, while an event monitor is active, an insert operation into a target table fails:
  - Uncommitted changes are rolled back.
  - A message is written to the administration log.
  - The event monitor is deactivated.
– If an event monitor is active, it performs a local COMMIT when it has finished processing an event monitor buffer.
– In an environment other than a partitioned database or a Db2 pureScale environment, all write to table event monitors are deactivated when the last application terminates (and the database has not been explicitly activated).

  In a Db2 pureScale environment, write to table event monitors are deactivated on a given member when the member stops and is reactivated when the member restarts.

  In a partitioned database environment, write to table event monitors are deactivated when the catalog partition deactivates.

  - The DROP EVENT MONITOR statement does not drop target tables.
  - Whenever a write-to-table event monitor activates, it will acquire IN table locks on each target table in order to prevent them from being modified while the event monitor is active. Table locks are maintained on all tables while the event monitor is active. If exclusive access is required on any of the target tables (for example, when a utility is to be run), first deactivate the event monitor to release the table locks before attempting such access.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - DBPARTITIONNUM or NODE can be specified in place of MEMBER, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
  - Commas can be used to separate multiple options in the *target-table-options* clause

Example

Define a threshold violation event monitor named DBTHRESHOLDVIOLATIONS

```
CREATE EVENT MONITOR DBTHRESHOLDVIOLATIONS
    FOR THRESHOLD VIOLATIONS
    WRITE TO TABLE
    THRESHOLDVIOLATIONS (TABLE THRESHOLDVIOLATIONS_DBTHRESHOLDVIOLATIONS
                         IN USERSPACE1
                         PCTDEACTIVATE 100),
    CONTROL (TABLE CONTROL_DBTHRESHOLDVIOLATIONS
             IN USERSPACE1
             PCTDEACTIVATE 100)
    AUTOSTART;
```

CREATE EVENT MONITOR (unit of work)

The CREATE EVENT MONITOR (unit of work) statement creates an event monitor that will record events when a unit of work completes.

Invocation

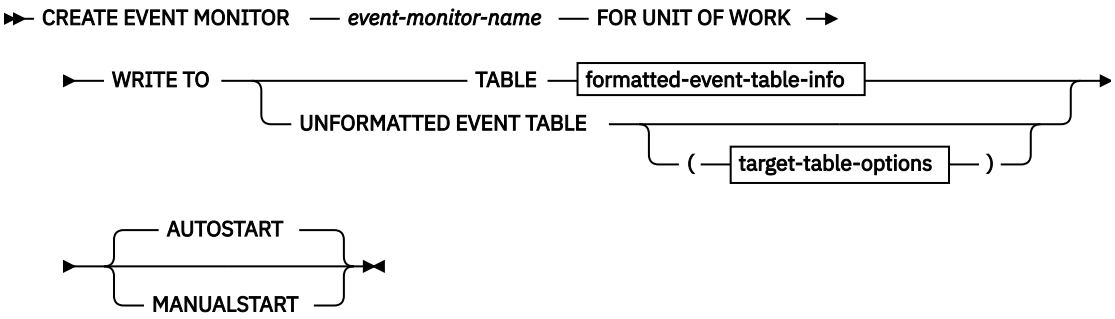
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

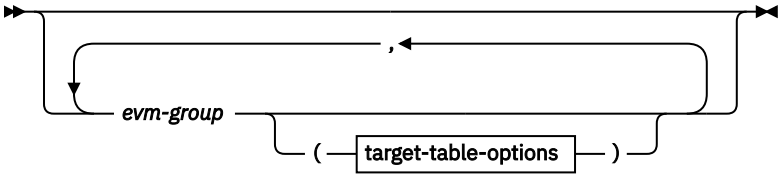
The privileges held by the authorization ID of the statement must include one of the following authorities:

- DBADM authority
- SQLADM authority

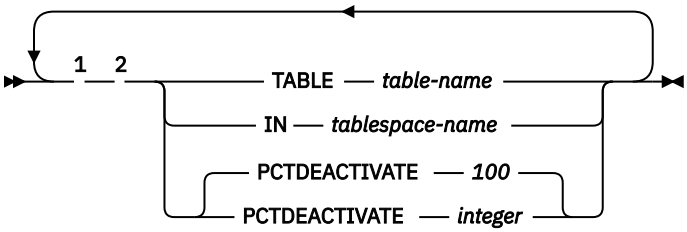
Syntax



formatted-event-table-info



target-table-options



Notes:

[1] Each table option can be specified a maximum of one time (SQLSTATE 42613).
[2] Clauses can be separated with a space or a comma.

Description

event-monitor-name
Name of the event monitor. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *event-monitor-name* must not identify an event monitor that already exists in the catalog (SQLSTATE 42710).

FOR
Introduces the type of event to record.

UNIT OF WORK
Specifies that this passive event monitor will record an event whenever a unit of work is completed (that is, whenever there is a commit or rollback).

The creation of the unit of work event monitor does not indicate that the unit of work data will be collected immediately. The actual unit of work event of interest is controlled at the workload level.

WRITE TO
Specifies the target for the data.

TABLE
Indicates that the target for the event monitor data is a set of database tables. The event monitor separates the data stream into one or more logical data groups and inserts each group into a separate table. Data for groups having a target table is kept, whereas data for groups not having a target table is discarded. Each monitor element contained within a group is mapped to a table column with the same name. Only elements that have a corresponding table column are inserted into the table. Other elements are discarded.

formatted-event-table-info
Defines the target formatted event tables for the event monitor. This clause should specify each grouping that is to be recorded. However, if no *evm-group* clauses are specified, all groups for the event monitor type are recorded.

For more information about logical data groups, see "Logical data groups and event monitor output tables" in *Database Monitoring Guide and Reference*.

evm-group
Identifies a logical data group for which a target table is being defined. The value depends upon the type of event monitor, as shown in the following table:

Type of Event Monitor	evm-group Value
Unit of work	• UOW • UOW_METRICS • UOW_PACKAGE_LIST • UOW_EXECUTABLE_LIST • CONTROL

UNFORMATTED EVENT TABLE
Specifies that the target for the event monitor is an unformatted event table. The unformatted event table is used to store collected unit of work event monitor data. Data is stored in its original binary format within an inlined BLOB column. The BLOB column can contain multiple binary records of different types. The data in the BLOB column is not in a readable format and requires conversion, through use of the **db2evmonfmt** Java-based tool, EVMON_FORMAT_UE_TO_XML table function, or EVMON_FORMAT_UE_TO_TABLES procedure, into a consumable format such as an XML document or a relational table.

target-table-options
Identifies options for the target table. If a value for **target-table-options** is not specified, CREATE EVENT MONITOR processing proceeds as follows:

- A derived table name is used (as explained in the description for TABLE *table-name*).
- A default table space is chosen using the same process as when a table is created without a table space name using CREATE TABLE.
- PCTDEACTIVATE is set to 100.

TABLE *table-name*
Specifies the name of the target table. The target table must be a non-partitioned table. If the name is unqualified, the table schema defaults to the value in the CURRENT SCHEMA special register. For an unformatted event table if a name is not provided, the unqualified name is equal to the *event-monitor-name*, that is, the unformatted event table will be named after the event monitor. For a formatted event table if no name is provided, the unqualified name is derived from *evm-group* and *event-monitor-name* as follows:

```
substring(evm-group CONCAT '_'
  CONCAT event-monitor-name,1,128)
```

IN *tablespace-name*
Specifies the table space in which the table is to be created. The CREATE EVENT MONITOR FOR UNIT OR WORK statement does not create table spaces.

If a table space name is not provided, the table space is chosen using the same process as when a table is created without a table space name using CREATE TABLE.

Since the page size affects the INLINE LOB lengths used, consider specifying a table space with as large a page size as possible in order to improve the INSERT performance of the event monitor.

PCTDEACTIVATE *integer*
If a table for the event monitor is being created in an automatic storage (non-temporary) or DMS table space, the PCTDEACTIVATE parameter specifies how full the table space must be before the event monitor automatically deactivates. The specified value, which represents a percentage, can range from 0 to 100, where 100 means that the event monitor deactivates when the table space becomes completely full. The default value assumed is 100 if PCTDEACTIVATE is not specified. This option is ignored for SMS table spaces.

Important: If the target table space has auto-resize enabled, set PCTDEACTIVATE parameter to 100. Alternatively, omit this clause entirely to have the default of 100 apply. Otherwise,

the event monitor might deactivate unexpectedly if the table space reaches the threshold specified by PCTDEACTIVTATE before the table space is automatically resized.

AUTOSTART
Specifies that the event monitor is to be automatically activated whenever the database partition on which the event monitor runs is activated. This is the default behavior of the unit of work event monitor.

MANUALSTART
Specifies that the event monitor must be activated manually using the SET EVENT MONITOR STATE statement. After a MANUALSTART event monitor has been activated, it can be deactivated only by using the SET EVENT MONITOR STATE statement or by stopping the instance.

Notes

- The table is created when the CREATE EVENT MONITOR FOR UNIT OF WORK statement executes, if it doesn't already exist.
- During CREATE EVENT MONITOR FOR UNIT OF WORK processing, if a table is found to have already been defined for use by another event monitor, the CREATE EVENT MONITOR FOR UNIT OF WORK statement fails, and an error is passed back to the application program. A table is defined for use by another event monitor if the table name matches a value found in the SYSCAT.EVENTTABLES catalog view. If the table exists and is not defined for use by another event monitor, then no table is created, any other table **target-table-options** parameters are ignored, and processing continues. A warning is passed back to the application program.
- Dropping the event monitor will not drop any tables. Any associated tables must be manually dropped after the event monitor is dropped.
- Lock event data is not automatically pruned from either unformatted event tables or regular tables created by this event monitor. An option for pruning data from UE tables is available when using the EVMON_FORMAT_UE_TO_TABLES procedure. For event monitors that write to regular tables, event data must be pruned manually.
- For unformatted event tables event data is inserted into the table into an inlined BLOB data column. Normally, BLOB data is stored in a separate LOB table space and can experience additional performance overhead as a result. When inlined into the data page of the base table, the BLOB data does not experience this overhead. The database manager will automatically inline the BLOB data portion of an unformatted event table record if the size of the BLOB data is less than the table space page size minus the record prefix. Therefore to achieve high efficiency and application throughput, it is suggested that you create the event monitor in as large a table space as possible up to and including a 32 KB table space and associated bufferpool.
- Create only one unit of work event monitor per database and not create multiple unit of work event monitors on the same database.
- In a partitioned database environment, data is written only to target tables on the database partitions where their table spaces exist. If a table space for a target unformatted event table does not exist on some database partition, data for that target table is ignored. This behavior allows users to choose a subset of database partitions for monitoring to be chosen, by creating a table space that exists only on certain database partitions.
- In a multi-member environment, data is only written to target tables on the member where work occurs within the unit of work.
- In a partitioned database environment, if some target tables do not reside on a database partition, but other target tables do reside on that same database partition, only the data for the target tables that do reside on that database partition is recorded.
- The unit of work event monitor is not affected by the unit or work event monitor switch. The unit of work event monitor switch is not changed when a unit or work event monitor is created, and the contents of the unit or work event monitor are not affected by changes to the unit of work event monitor switch.
- The FLUSH EVENT MONITOR statement is not applicable to this event monitor and will have no effect when issued against it.

- Creation of the unit of work event monitor does not cause events to be written to the event monitor. The unit of work event monitor must be activated with SET EVENT MONITOR STATE, and the unit of work data must be collected by either altering the appropriate workload to specify COLLECT UNIT OF WORK DATA or setting the **mon_uow_data** database configuration parameter to a value other than NONE.
- When using unformatted event tables, create the unit of work event monitor in a table space with at least 8 KB page size to ensure that the event data is contained within the inlined BLOB column of the unformatted event table. If the BLOB column is not inlined, then the performance of writing and reading the events to the unformatted event table might not be efficient.

Examples

- *Example 1:* This example creates a unit of work event monitor UOWEVMON that collects data for unit of work events that occur on the database of creation, and writes data tables using default table names:

```
CREATE EVENT MONITOR UOWEVMON
  FOR UNIT OF WORK
  WRITE TO TABLE
```

  This event monitor writes its output to the following tables:

  UOW_UOWEVMON
  UOW_METRICS_UOWEVMON
  UOW_PACKAGE_LIST_UOWEVMON
  UOW_EXECUTABLE_LIST_UOWEVMON
  UOW_CONTROL_UOWEVMON

  Note: Whether the tables for package list and executable list information are populated with data is dependent on whether you specify that that data is to be collected. You control the collection of this data is using the mon_uow_pkglist or mon_uow_execlist configuration parameters, or with the appropriate COLLECT UNIT OF WORK DATA clause on the CREATE or ALTER WORKLOAD statements.
- *Example 2:* This example creates a unit of work event monitor UOWEVMON that will collect unit of work events that occur on the database of creation and store it in the unformatted event table GREG.UOWEVENTS.

```
CREATE EVENT MONITOR UOWEVMON
  FOR UNIT OF WORK
  WRITE TO UNFORMATTED EVENT TABLE (TABLE GREG.UOWEVENTS)
```

- *Example 3:* This example creates a unit of work event monitor UOWEVMON that will collect unit of work events that occur on the database of creation and store it in the unformatted event table GREG.UOWEVENTS in table space APPSPACE. The event monitor will deactivate when the table space becomes 85% full.

```
CREATE EVENT MONITOR UOWEVMON
  FOR UNIT OF WORK
  WRITE TO UNFORMATTED EVENT TABLE
     (TABLE GREG.UOWEVENTS IN APPSPACE PCTDEACTIVATE 85)
```

CREATE EXTERNAL TABLE

While tables typically reside in a database, an external table resides in a text-based, delimited file, or in a fixed-length-format file outside of a database.

Use an external table to:

- Store data outside the database while retaining the ability to query that data. To unload data from the database into an external file, specify the external table as the target table in one of the following SQL statements:
  - INSERT SQL
  - SELECT INTO SQL

  - CREATE EXTERNAL TABLE AS SELECT SQL
- Load data from an external file into a table in the database. You can perform operations such as casts, joins, and dropping columns to manipulate data during loading. To load data into the database from an external table, use a FROM clause in a SELECT SQL statement as you would for any other table.
- Transfer data to another application.

An advantage of using external tables for Extraction-Transformation-Loading (ETL) processes is that they can be carried out using plain SQL. Because an SQL-based ETL process can be initiated from any SQL client, it eliminates the need for special ETL tools.

An external table is of one of the following types:

Named
The external table has a name and catalog entry similar to a normal table.

Transient
The external table has a system-generated name of the form SYSTET<number> and does not have a catalog entry. For example, the system might create a transient external table to hold the result of a query. The lifetime of such a table is the duration of the query.

Invocation

This statement can be embedded in an application program or issued using dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges that are held by the authorization ID of the statement must have either DBADM authority or the following privileges or authorities:

- CREATETAB authority
- One of the following privileges or authorities:
  - USE privilege on the table space
  - SYSADM authority
  - SYSCTRL authority
- One of the following privileges or authorities:
  - IMPLICIT_SCHEMA authority on the database (if the implicit or explicit schema name of the table does not exist)
  - CREATEIN privilege on the schema (if the schema name of the table refers to an existing schema)
  - SCHEMAADM authority on the schema (if the schema name of the table refers to an existing schema)

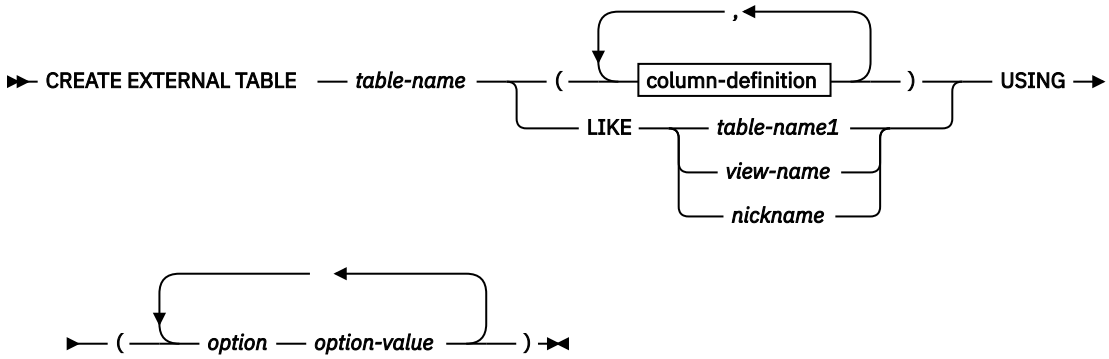
If a subtable is being defined, one of the following conditions must be met:

- The authorization ID must be the same as the owner of the root table of the table hierarchy.
- The privileges that are held by the authorization ID must include SCHEMAADM authority on the schema that contains the root table of the table hierarchy.
- The privileges that are held by the authorization ID must include DBADM authority.

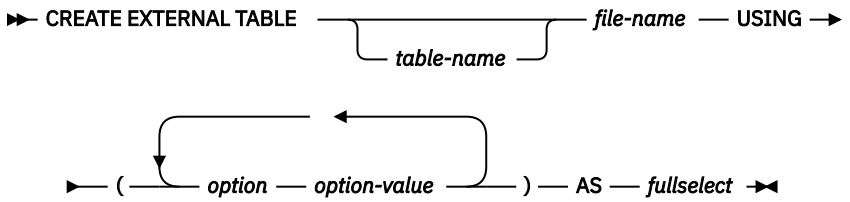
Syntax

The syntax of this statement depends on the nature of the external table that is to be created:

- Use the following syntax to create, in the catalog, a table definition for a new external table. Specifying a table name is mandatory, so the resulting external table is a named table. A DATAOBJECT or FILE_NAME option must be specified to identify the target file.

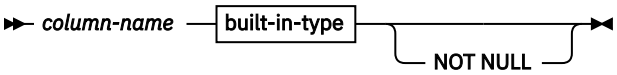


- Use the following syntax to use an existing table as a template for a new external table and to populate it with the contents of the source table. If this statement specifies a table name explicitly, the resulting external table is a named table; otherwise, the resulting external table is a transient table. The file name must be specified by either the `file-name` parameter or a DATAOBJECT or FILE_NAME option.

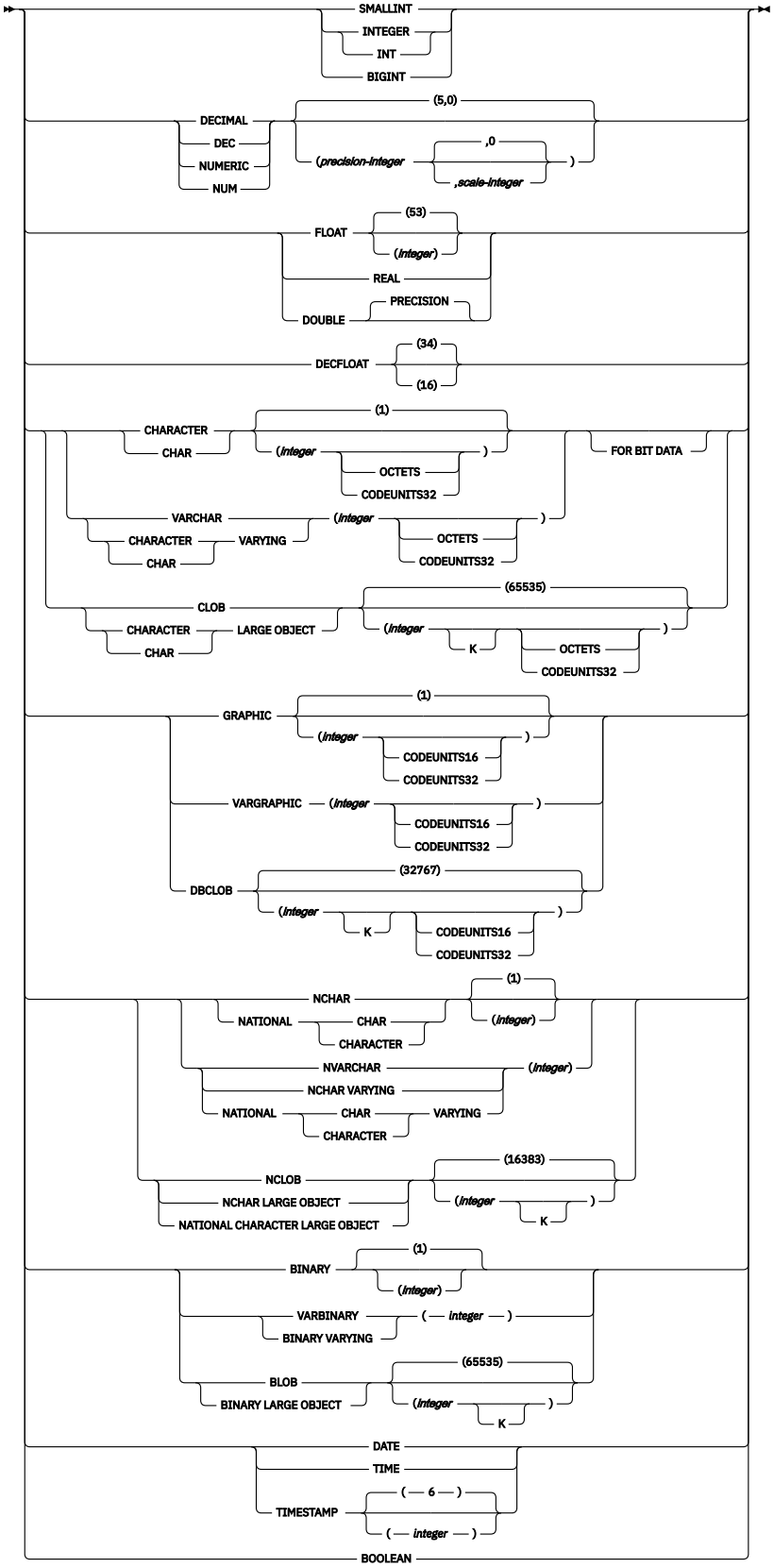


If you specify one or more column definitions, the following additional parameters apply:

column-definition



built-in-type


SMALLINT
INTEGER
INT
BIGINT
DECIMAL
DEC
NUMERIC
NUM
(5,0)
(precision-integer
,0
,scale-integer
)
FLOAT
(53)
(integer)
REAL
DOUBLE
PRECISION
DECFLOAT
(34)
(16)
CHARACTER
CHAR
(1)
(integer
OCTETS
CODEUNITS32
)
FOR BIT DATA
VARCHAR
CHARACTER
VARYING
CHAR
CLOB
LARGE OBJECT
(65535)
K
GRAPHIC
(integer)
CODEUNITS16
CODEUNITS32
VARGRAPHIC
DBCLOB
(32767)
NCHAR
NATIONAL
NVARCHAR
NCHAR VARYING
NCLOB
NCHAR LARGE OBJECT
NATIONAL CHARACTER LARGE OBJECT
(16383)
BINARY
VARBINARY
BINARY VARYING
BLOB
BINARY LARGE OBJECT
DATE
TIME
TIMESTAMP
(6)
(integer)
BOOLEAN

Description

table-name
The names of the external table. The name, including the implicit or explicit qualifier, must not identify a table, view, nickname, or alias that is already described in the catalog. The schema name cannot be SYSIBM, SYSCAT, SYSFUN, or SYSSTAT (SQLSTATE 42939).

file-name
The fully-qualified name of the file (or any medium that can be treated as a file) that is to contain the external table to be created. If this parameter is specified, the DATAOBJECT or FILE_NAME option cannot be specified.

If the input data file is `gzip` compressed, this will be detected and the data is uncompressed by the Db2 server. The `COMPRESS GZIP` option can be used, but it is not mandatory for loading `gzip` compressed files.

If the input data file is `lz4` compressed, the `COMPRESS LZ4` option must be specified.

When both the REMOTESOURCE option is set to LOCAL (this is its default value) and the **`extbl_strict_io`** configuration parameter is set to NO, the path to the external table file is an absolute path and must be one of the paths specified by the **`extbl_location`** configuration parameter. Otherwise, the path to the external table file is relative to the path that is specified by the **`extbl_location`** configuration parameter followed by the authorization ID of the table definer. For example, if **`extbl_location`** is set to `/home/xyz` and the authorization ID of the table definer is `user1`, the path to the external table file is relative to `/home/xyz/user1/`.

The file name must be a valid UTF-8 string.
For a load operation, the following conditions apply:

- The file must already exist.
- Required permissions:
  - If the external table is a named external table, the owner must have read permission for the file and write permission for the LOGDIR directory.
  - If the external table is a transient external table, the authorization ID of the statement must have read permission for the file and write permission for the LOGDIR directory.

For an unload operation, the following conditions apply:

- If the file exists, it is overwritten.
- Required permissions:
  - If the external table is a named external table, the owner must have read and write permission for the directory of this file.
  - If the external table is transient, the authorization ID of the statement must have read and write permission for the directory of this file.

column-definition
Defines the attributes of a column.

column-name
Names a column of the table. The name cannot be qualified, and the same name cannot be used for more than one column of the table (SQLSTATE 42711).

built-in-type
One of the following built-in data types:

SMALLINT
A small integer.

[INTEGER | INT]
A large integer.

BIGINT
A big integer.

[DECIMAL | DEC | NUMERIC | NUM](*precision-integer, scale-integer*)
A decimal number.

- The precision integer specifies the total number of digits. It must be in the range 1 - 31. The default is 5.
- The scale integer specifies the number of digits to the right of the decimal point. It cannot be negative and cannot exceed the precision. The default is 0.

FLOAT(*integer*)
A single or double-precision floating-point number. If the specified length is in the range:

- 1 - 24, the number uses single precision
- 25 - 53, the number uses double-precision

Instead of FLOAT, you can specify:

REAL
For single precision floating-point.

DOUBLE
For double-precision floating-point.

DOUBLE PRECISION
For double-precision floating-point.

FLOAT
For double-precision floating-point.

DECFLOAT*(precision-integer)*
A decimal floating-point number. The precision integer specifies the total number of digits, which can be either 16 or 34. The default is 34.

[CHARACTER | CHAR](*integer* [OCTETS | CODEUNITS32])
A fixed-length character string of the specified number of code units. This number can range from 1 - 255 OCTETS or from 1 - 63 CODEUNITS32. The default is 1.

[VARCHAR | CHARACTER VARYING | CHAR VARYING](*integer* [OCTETS | CODEUNITS32])
A varying-length character string with a maximum length of the specified number of code units. This number can range from 1 - 32672 OCTETS or from 1 - 8168 CODEUNITS32.

FOR BIT DATA
Specifies that the contents of the column are to be treated as bit (binary) data. During data exchange with other systems, code page conversions are not performed. Comparisons are done in binary, irrespective of the database collating sequence.

The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

[CLOB | CHARACTER LARGE OBJECT | CHAR LARGE OBJECT](*integer* [K] [OCTETS | CODEUNITS32])
A character large object string with a maximum length of the specified number of code units. The default maximum length is 65,535 bytes.

If you want to multiply the length integer by 1024, specify a K (kilo) multiplier.

- Regardless of whether you use a K multiplier, the resulting length is limited by the maximum length of a CLOB column in an external table, which is 65,535 OCTETS, 32,767 CODEUNITS16, or 16,383 CODEUNITS32. Note that 64K OCTETS and 16K CODEUNITS32 each exceed the maximum length by one, and so are not allowed.
- Any number of spaces (including zero spaces) are allowed between the data type and the length specification or between the length integer and the K multiplier. For example, the following specifications are all equivalent and valid:

```
CLOB(50K)
CLOB(50 K)
CLOB (50   K)
```

- The K multiplier can be specified in either uppercase or lowercase.

In a Unicode database, the default string units for a character string data type are determined by the value of the NLS_STRING_UNITS global variable or *string_units* database configuration parameter. In a non-Unicode database, the default string units for character string data types are OCTETS.

OCTETS
Specifies that the units of the length attribute are bytes.

CODEUNITS32
Specifies that the units of the length attribute are Unicode UTF-32 code units, which approximates counting in characters. This does not affect the underlying code page of the data type. The actual length of a data value is determined by counting the UTF-32 code units as if the data were converted to UTF-32. CODEUNITS32 can be specified only in a Unicode database (SQLSTATE 560AA).

GRAPHIC(*integer* [CODEUNITS16 | CODEUNITS32])
A fixed-length graphic string of the specified length, which can range from 1 - 127 double bytes, 1 - 127 CODEUNITS16, or 1 - 63 CODEUNITS32. The default length is 1.

VARGRAPHIC(*integer* [CODEUNITS16 | CODEUNITS32])
A varying-length graphic string of the specified maximum length, which can range from 1 - 16336 double bytes, 1 - 16336 CODEUNITS16, or 1 - 8168 CODEUNITS32.

DBCLOB(*integer* [K] [CODEUNITS16 | CODEUNITS32])
A character large object string of the specified maximum length in double bytes, Unicode UTF-16 code units, or Unicode UTF-32 code units. The default maximum length is 32,767 double bytes.

If you want to multiply the length integer by 1024, specify a K (kilo) multiplier.

- Regardless of whether you use a K multiplier, the resulting length is limited by the maximum length of a DBCLOB column in an external table, which is 32,767 CODEUNITS16 or 16,383 CODEUNITS32. Note that 32K CODEUNITS16 and 16K CODEUNITS32 each exceed the maximum length by one, and so are not allowed.
- Any number of spaces (including zero spaces) are allowed between the data type and the length specification or between the length integer and the K multiplier. For example, the following specifications are all equivalent and valid:

```
DBCLOB(50K)
DBCLOB(50 K)
DBCLOB (50   K)
```

- The K multiplier can be specified in either uppercase or lowercase.

In a Unicode database, the default string units for a character string data type are determined by the value of the NLS_STRING_UNITS global variable or *string_units* database configuration parameter. In a non-Unicode database, the default string units for character string data types is CODEUNITS16.

CODEUNITS16
Specifies that the units of the length attribute are Unicode UTF-16 code units, which is the same as counting in double bytes. CODEUNITS16 can be specified only in a Unicode database (SQLSTATE 560AA).

CODEUNITS32
Specifies that the units of the length attribute are Unicode UTF-32 code units. This does not affect the underlying code page of the data type. The actual length of a data value is determined by counting the UTF-32 code units as if the data were converted to UTF-32. CODEUNITS32 can be specified only in a Unicode database (SQLSTATE 560AA).

[NATIONAL CHARACTER | NATIONAL CHAR | NCHAR](*integer*)
A fixed-length string of the specified length. The default length is 1.

The NATIONAL CHARACTER type maps to either a fixed-length character or a fixed-length graphic string, depending on the value of the *nchar_mapping* database configuration parameter, which also defines the string units.

[NATIONAL CHARACTER VARYING | NATIONAL CHAR VARYING | NCHAR VARYING | NVARCHAR](*integer*)
A varying-length string of the specified maximum length.

The NATIONAL CHARACTER VARYING type maps to either a varying-length character or a varying-length graphic string, depending on the value of the *nchar_mapping* database configuration parameter, which also defines the string units.

[NATIONAL CHARACTER LARGE OBJECT | NCHAR LARGE OBJECT | NCLOB](*integer* [K])
A large object string of the specified maximum length. The default maximum length is 16,383 double bytes.

This data type maps to either a character large object (CLOB) or a double-byte character large object (DBCLOB), depending on the current value of the *nchar_mapping* database configuration parameter, which also defines the string units. See the description of the CLOB or DBCLOB parameter (whichever applies) for information about possible values for the length integer and how to use a K (kilo) multiplier.

BINARY(*integer*)
A fixed-length binary string of the specified length, which must be in the range 1 - 255 bytes. The default length is 1.

[VARBINARY | BINARY VARYING](*integer*)
A varying-length binary string of the specified maximum length, which must be in the range 1 - 32672 bytes.

[BLOB | BINARY LARGE OBJECT](*integer* [K])
A binary large object string with a maximum length of the specified number of code units. The default maximum length is 65,535 bytes.

If you want to multiply the length integer by 1024, specify a K (kilo) multiplier.

- Regardless of whether you use a K multiplier, the resulting length is limited by the maximum length of a BLOB column in an external table, which is 65,535 bytes. Note that 64K exceeds the maximum length by one, and so is not allowed.
- Any number of spaces (including zero spaces) are allowed between the data type and the length specification or between the length integer and the K multiplier. For example, the following specifications are all equivalent and valid:

```
BLOB(50K)
BLOB(50 K)
BLOB (50   K)
```

- The K multiplier can be specified in either uppercase or lowercase.

DATE
A date.

TIME
A time.

TIMESTAMP(*integer*) or TIMESTAMP
A timestamp. The integer specifies the number of decimal places for fractions of seconds, from 0 (seconds) to 12 (picoseconds). The default is 6 (microseconds).

BOOLEAN
A Boolean value.

LIKE *table-name1* or *view-name* or *nickname*
Specifies that the columns of the table have the same name and description as the columns of the specified table (*table-name1*), view (*view-name*), or nickname (*nickname*). The specified table, view,

or nickname must either exist in the catalog or must be a declared temporary table. A typed table or typed view cannot be specified (SQLSTATE 428EC).

The use of LIKE is an implicit definition of *n* columns, where *n* is the number of columns in the identified table (including implicitly hidden columns), view, or nickname. A column of the new table that corresponds to an implicitly hidden column in the existing table will also be defined as implicitly hidden. The implicit definition depends on what is specified after LIKE:

- If a table is specified, then the implicit definition includes the column name, data type, hidden attribute, and nullability characteristic of each of the columns of that table. If EXCLUDING COLUMN DEFAULTS is not specified, then the column default is also included.
- If a view is specified, then the implicit definition includes the column name, data type, and nullability characteristic of each of the result columns of the fullselect defined in that view. The data types of the view columns must be data types that are valid for columns of a table.
- If a nickname is specified, then the implicit definition includes the column name, data type, and nullability characteristic of each column of that nickname.
- If a protected table is specified, the new table inherits the same security policy and protected columns as the identified table.
- If a table is specified and if that table contains a row-begin column, row-end column, or transaction-start-ID column, the corresponding column of the new table inherits only the data type of the source column. The new column is not considered a generated column.
- If a table that includes a period is specified, the new table does not inherit the period definition.
- If a system-period temporal table is specified, the new table is not a system-period temporal table.
- If a random distribution table that uses the **random by generation** method is specified, and if the new table that is being created does not share the same table distribution, the **RANDOM_DISTRIBUTION_KEY** column that is used to generate the random distribution values is not included.

Column default attributes can be included or excluded, based on the copy-attributes clauses. The implicit definition does not include any other attributes of the specified table, view, or nickname. Consequently, the new table does not have any primary key, unique constraints, foreign key constraints, referential integrity constraints, triggers, indexes, ORGANIZE BY specification, or PARTITIONING KEY specification.

When a table is identified in the LIKE clause and that table contains a ROW CHANGE TIMESTAMP column, the corresponding column of the new table inherits only the data type of the ROW CHANGE TIMESTAMP column. The new column is not considered to be a generated column.

If a table is specified, and if row or column level access control is activated for that table, it is not inherited by the new table.

option

The following options control the loading of data to or retrieval of data from an external-table file. The value of each option is a text string and is not case-sensitive.

BOOLSTYLE or BOOLEAN_STYLE

During a load operation, all Boolean values must use the same style. This option specifies the Boolean style that is to be used:

- 1_0 (this is the default)
- T_F
- Y_N
- YES_NO
- TRUE_FALSE

CARDINALITY

Non-zero positive integer value to override the estimation of the expected number of returned rows.

CCSID
The coded character set identifier (CCSID) of the input data file. The value can be any valid integer value from the CCSID specification. There is no default value. The CCSID and ENCODING options are mutually exclusive when the value of the ENCODING option is UTF8, LATIN9, or INTERNAL.

Which styles are used for dates and times depends on whether a CCSID is specified:

- When a CCSID is specified, and when DATESTYLE, TIMESTYLE, DATEDELIM, or TIMEDELIM are not specified, the values or defaults for DATE_FORMAT, TIME_FORMAT, and TIMESTAMP_FORMAT are used.
- When a CCSID is not specified, and when TIMESTAMP_FORMAT, DATE_FORMAT or TIME_FORMAT are not specified, the values or defaults for DATESTYLE, TIMESTYLE, DATEDELIM, and TIMEDELIM are used.

COMPRESS
For a load operation or an unload operation, whether the data file data is compressed:

GZIP
The data file data is compressed by using the GZIP compression algorithm.

NO
The data file data is not compressed. This is the default.

LZ4
The data file data is compressed by using the LZ4 compression algorithm.

The COMPRESS option cannot be specified if the value of the REMOTESOURCE option is GZIP or LZ4.

CRINSTRING
How to interpret an unescaped carriage-return (CR) or carriage-return line-feed (CRLF) character:

TRUE or ON
An unescaped CR character is interpreted as data, not as a record delimiter.

FALSE or OFF
An unescaped CR is interpreted as a record delimiter. This is the default.

Use fixed-length format for **CRINSTRING** only if the value of the **CtrlChars** option is set to OFF.

CTRLCHARS
Whether to allow an ASCII value 1 - 31 in a CHAR or VARCHAR field. Any NULL, CR, or LF characters must be escaped. Allowed values are:

TRUE or ON
An ASCII value 1 - 31 in a CHAR or VARCHAR field is allowed.

If fixed-length format is enabled, all unescaped characters are allowed.

FALSE or OFF
An ASCII value 1 - 31 in a CHAR or VARCHAR field is not allowed. This is the default.

If fixed-length format is enabled, unescaped characters cause an error.

Exceptions for fixed-length format:

- \t, \n
- \r if the CRinString option is set to ON

DATAOBJECT or FILE_NAME
The fully-qualified name of the file (or any medium that can be treated as a file) that is to contain the external table to be created. This option is mandatory when the name of the file is not specified immediately after the table name; otherwise, it is not allowed.

When both the REMOTESOURCE option is set to LOCAL (this is its default value) and the **extbl_strict_io** configuration parameter is set to NO, the path to the external table file is an absolute path and must be one of the paths specified by the **extbl_location** configuration parameter. Otherwise, the path to the external table file is relative to the path that is specified

by the **extbl_location** configuration parameter followed by the authorization ID of the table definer. For example, if **extbl_location** is set to `/home/xyz` and the authorization ID of the table definer is `user1`, the path to the external table file is relative to `/home/xyz/user1/`.

The file name must be a valid UTF-8 string.
For a load operation, the following conditions apply:

- The file must already exist.
- Required permissions:
  - If the external table is a named external table, the owner must have read permission for the file and write permission for the LOGDIR directory.
  - If the external table is a transient external table, the authorization ID of the statement must have read permission for the file and write permission for the LOGDIR directory.

For an unload operation, the following conditions apply:

- If the file exists, it is overwritten.
- Required permissions:
  - If the external table is a named external table, the owner must have read and write permission for the directory of this file.
  - If the external table is transient, the authorization ID of the statement must have read and write permission for the directory of this file.

DATEDELIM
The delimiter character that separates the components of a date, according to the format specified by the DATESTYLE option. If you specify an empty string, there is no delimiter between the date components, and days and months must be specified as two-digit numbers. When DATESTYLE is set to MONDY or MONDY2, the default DATEDELIM value is a space. The TIMESTAMP_FORMAT and DATEDELIM options are mutually exclusive.

DATESTYLE
How to interpret the date format. For days or months in the range 1 - 9, use 1 digit, 2 digits, or a space followed by a single digit. When the DATEDELIM option is a space, you can specify a comma after the day. An error occurs if you:

- Specify zero for a day, month, or year
- Specify a nonexistent date (for example, 32 August or 30 February)

The DATESTYLE option and the DATE_FORMAT or TIMESTAMP_FORMAT option are mutually exclusive.

Table 131. Possible values for the DateStyle option. The example shows how the date 21 March 2014 would be represented when DATEDELIM is set to '-'.

Value	Description	Example
YMD	4-digit year, 2-digit month, 2-digit day. This is the default.	2014-03-21
DMY	2-digit day, 2-digit month, 4-digit year.	21-03-2014
MDY	2-digit month, 2-digit day, 4-digit year.	03-21-2014
MONDY	3-character month, 2-digit day, 4-digit year.	Mar 21 2014
DMONY	2-digit day, 3-character month, 4-digit year.	21-Mar-2014
Y2MD	2-digit year, 2-digit month, 2-digit day. Not supported for unloads.	14-03-21
DMY2	2-digit day, 2-digit month, 2-digit year. Not supported for unloads.	21-03-14

Table 131. Possible values for the DateStyle option. The example shows how the date 21 March 2014 would be represented when DATEDELIM is set to '-'. *(continued)*

Value	Description	Example
MDY2	2-digit month, 2-digit day, 2-digit year. Not supported for unloads.	03-21-14
MONDY2	3-character month, 2-digit day, 2-digit year. Not supported for unloads.	Mar 21 14
DMONY2	2-digit day, 3-character month, 2-digit year. Not supported for unloads.	21-Mar-14

DATETIMEDELIM
A single-byte character that separates the date component and time component of the timestamp data type.

The default delimiter is a space (' ').

Between the date component and the time component, a delimiter is not required. For example, both of the following values are valid:

```
2010-10-10 10:10:10
2010-10-1010:10:10
```

DATE_FORMAT
The format of the date field in the data file. The value can be any of the date format strings that are accepted by the "TIMESTAMP_FORMAT " on page 527. The default is YYYY-MM-DD. The DATE_FORMAT option and the DATEDELIM or DATESTYLE option are mutually exclusive.

DECIMALDELIM or DECIMAL_CHARACTER
The decimal delimiter for the data types FLOAT, DOUBLE, TIME, and TIMESTAMP. Allowed values are ' , ' and ' . '.

DECPLUSBLANK
Specifies how the positive decimal value is represented during the unload operation.

You can specify one of the following values for this option:

NONE
This is the default.

This value represents a positive decimal value without a sign.

PLUS
Specifies that a positive decimal value is represented by a '+' sign.

BLANK
Specifies that a positive decimal value is represented by a blank sign instead of a '+' sign.

If you specify the DECPLUSBLANK option for the load operation, the output is not affected.

Examples for a table test with ddl (decimal (6,2)) and all the available values for the DECPLUSBLANK option:

```
1234
-4563
```

- Create external table '/tmp/unload.txt' using (DECPLUSBLANK NONE) as select * from test:

```
unload.txt
1234.00
-4563.00
```

- Create external table '/tmp/unload.txt' using (DECPLUSBLANK PLUS) as select * from test:

```
unload.txt
+1234.00
-4563.00
```

- Create external table '/tmp/unload.txt' using (DECPLUSBLANK BLANK) as select * from test:

```
unload.txt
 1234.00
-4563.00
```

DELIMITER or COLUMN_DELIMITER
The character that is used to delimit the fields of an input or output record. The default is a vertical bar (`'|'`).

You can specify a character in the 7-bit ASCII range (decimal 1 - 127) in any of the following ways:

- As a single character (for example `DELIMITER ';'`)
- By specifying its corresponding ASCII decimal value (for example, `DELIMITER 59` or `DELIMITER '59'`)
- By specifying its corresponding ASCII hex value (for example, `DELIMITER x'3B'`)

The decimal range 128 - 255 is supported only with the ISO character set input file by specifying its corresponding ASCII decimal value or hex value. If the input file is in the UTF8 character set, this delimiter value range is not supported.

ENCODING
The type of data in the file:

UTF8
The file uses UTF8 encoding for all character data.

LATIN9
The file uses LATIN9 encoding for all character data.

INTERNAL
This is the default option.

The file uses a mixture of both UTF8 and LATIN9 encoding.

Files are encoded in Netezza internal format and therefore should be used only for files that are extracted from Netezza by using ENCODING (INTERNAL).

When the target column is CODEUINTS32 (NCHAR/VARCHAR), the input data is validated to be valid UTF-8 characters.

This option is supported only in a Unicode database.

DBCS_GRAPHIC
This value is allowed only for a load operation, not an unload operation. If this value is specified, the CCSID option must also be specified. During the load operation, fields of type GRAPHIC or VARGRAPHIC are encoded using the double-byte character set of the specified CCSID; fields of all other types are encoded using the mixed-byte character set of the specified CCSID.

Note: ENCODING cannot be set to DBCS_GRAPHIC for a DEL file that was created by the EXPORT utility, because such DEL files are encoded using a single character set.

The CCSID and ENCODING options are mutually exclusive when the value of the ENCODING option is UTF8, LATIN9, or INTERNAL.

ESCAPECHAR or ESCAPE_CHARACTER
Which character is to be regarded as an escape character. An escape character indicates that the character that follows it, which would otherwise be treated as a field-delimiter character or end-of-row sequence character, is instead treated as part of the value in the field. The escape character is ignored for graphic-string data. There is no default.

FILLRECORD
For a load operation, the field of a record are loaded into the columns of a target table from left to right. This option specifies whether an input record can contain fewer fields than there are columns defined for the target table:

TRUE or ON
An input line can contain fewer fields, provided that all columns for which a value is missing are nullable. Missing values are set to NULL. If one or more columns for which a value is missing is not nullable, the record is rejected.

FALSE or OFF
An input line that contains fewer columns is rejected. This is the default.

FORMAT or FILE_FORMAT
The data format of the source file:

TEXT
The data to be loaded or unloaded is in text-delimited format. This is the default.

INTERNAL
The data is in an internal format used by Netezza Platform Software (NPS). This value is valid only when loading data from a file to the database, not when unloading data to a file. If this value is specified for the FORMAT option, the following options, and only these options, must also be specified:

- DATAOBJECT or FILE_NAME.
- REMOTESOURCE, SWIFT or S3. If the REMOTESOURCE option is specified, it must have the value LOCAL or YES.

  Attention: In the container-only release of Db2 Version 11.5 Mod Pack 1 or later releases, AZURE is compatible, in addition to SWIFT and S3.
- COMPRESS. This must be set to GZIP.

BINARY
The data is in an internal format that is used by Db2.

FIXED
The data is in fixed-length format.

Fixed-length format is supported only for load operations.

Files in fixed-length format use ordinal positions, which are offsets, to identify where fields are within the record.

Note:

- The following external table options are not supported for the fixed-length format:
  - Delimiter
  - Encoding
  - EscapeChar
  - FillRecord
  - IgnoreZero
  - IncludeZeroSeconds
  - Lfinstring
  - QuotedValue
  - RequireQuotes
  - TimeExtraZeros
  - TruncString
- There are no field delimiters.

- An end-of-record delimiter is required even for the last record.
- Usually, data in fixed-length format files does not have decimal delimiters or time delimiters because delimiters are not necessary and use space.
- The locations of delimiters are fixed and specified in the layout definition because the fields are fixed in size. This definition comes with the fixed-length format data file.
- To load fixed-format data into the database, you must define the target data type for the fields and the locations within the record.
- You do not have to load all fields in a fixed-length format file. You can skip them by using the *filler* specification.
- The order of fields in the data file must match the order in the target table. Alternatively, you must create an external table definition that specifies the order of the fields as database columns.
- You can change the field order by using an external table definition in combination with an insert-select statement.
- Typically, unknown values or null values are represented by known data patterns that are classified as representing null.

The following parameters apply when the FORMAT option of the external table is set to FIXED:

LAYOUT

Mandatory.

A layout is an ordered collection of zone or field definitions. It defines the location of the fields of the input record.

Specify comma-separated zone definitions within braces { }.

Each zone definition is made up of mutually exclusive, non-overlapping clauses.

No default value.

The clauses must be in the following order, even if some of them are optional and can be empty:

USE TYPE

Optional.

Indicates whether a zone is a normal data zone, a reference zone, or a filler zone.

For data zones, this value is omitted.

A reference zone is specified as **REF**. This specification implies that the zone is referred by another zone for zone length or null values.

A filler zone is specified as **FILLER**. Filler zones specify that the bytes or characters are treated as fillers in a data file.

NAME

Optional.

The name of the zone.

Currently, this definition is not used. Typically, it is provided to identify the field.

TYPE

Optional.

Defines the type of the zone.

If you do not specify the type, it gets the default value of the corresponding type of a table column.

Valid values are as follows:

- CHAR
- VARCHAR
- NCHAR

- NVARCHAR
- SMALLINT
- BIGINT
- BINARY
- VARBINARY
- GRAPHIC
- VARGRAPHIC
- FLOAT
- DOUBLE
- DEC, NUM, or NUMERIC
- DECFLOAT
- BOOLEAN
- DATE
- TIME
- TIMESTAMP

STYLE

Optional.

Defines the zone representation.

The default representation is based on zone type and format option.

All other styles are valid only for their corresponding non-textual zone types.

Valid values are as follows:

- INTERNAL

  Valid only for textual zones, that is, char, varchar, nchar, and nvarchar.
- DECIMAL

  Valid for integer and numeric zone types.
- DECIMALDELIM <'decimal-delim'>

  Valid for numeric, float, double, and time style (time and timestamp) zone types.
- FLOATING

  Valid for float or double zone types.
- EXPONENTIAL

  Valid for float or double zone types.
- YMD <'date-delim'>

  Valid for date zones, including other date styles that are supported for the DateStyle and DateDelim external table options.
- 12Hour <'time-delim'>

  Valid for time zones, including other time styles that are supported for the TimeStyle and TimeDelim external table options.
- 24Hour <'time-delim'>

  Valid for time zones, including other time styles that are supported for the TimeStyle and TimeDelim external table options.
- YMD <'date-delim'> 24Hour <'time-delim'>

Valid for timestamp zones, including other combinations of date and time styles that are supported for the DateStyle, DateDelim, TimeStyle, and TimeDelim external table options.

- TRUE_FALSE, Y_N, 1_0

  Valid for boolean zones, including other boolean styles that are supported for the BoolStyle external table option. The style must be in accordance with the format.

LENGTH

Optional.

Specified as bytes or characters followed by the number or the internal reference to the reference zone.

Number of bytes or characters as provided or as referenced by the reference zone.

For reference zones or filler zones, you cannot use internal references. For reference zones, the number of bytes specifies how the data is read from the data file to get the referred value.

You can use plus signs and minus signs as follows:

```
BYTES @2 + 10
BYTES @2 - 10
```

NULLIF

Optional.

Definition of the zone NULLESS attribute.

Specifies a known data pattern within the field that, when it is present, signifies that the field is null.

The length is equal to or less than the column width. The maximum length is 39 bytes.

You can use the following types of references:

@

Internal reference to numeric zones.

Exact match of the numeric value.

&

External reference.

Exact match of the specified value.

&&

Isolated reference.

Leading spaces and trailing spaces are to be skipped with the exact string match.

Nulls are detailed in the following examples:

Table 132. Layout example

Use type	Name	Type	Style	Length	Nullif
NA	f1	int4	DECIMAL	Bytes 10	Nullif & = 0
NA	f2	date	YMD	Bytes 10	Nullif &= '2000-10-10'
NA	f3	char(20)	INTERNAL	Chars 10	Nullif &&='ab'
Filler	f4	char(10)	NA	Bytes 10	NA

Remember:

- The referred zone in a length clause must be of type integer.

- You must not specify the NULLIF option for reference zones or filler zones.
- Reference zones and filler zones cannot have variable lengths.
- Variable length zones cannot refer themselves.
- Define the referred zone in a length clause as REF.
- Length-clause references can use only the INTERNAL (@) reference. External or isolated references are not supported.
- Between the referred zone of a length clause and the zone itself, reference zones are not allowed.
- If the reference type is INTERNAL (@), the NULLIF clause cannot refer to itself.
- If the column is non-nullable, it may not have the NULLIF clause.
- Variable length is allowed only for the string type of zones.
- The NULLIF clause can refer only to REF zones or the zones themselves.
- Between the zone that is referred by the NULLIF clause and the zone itself, other referred zones are not allowed, except for the zone that is referred in the length clause.
- The record length can point to zone 1 only for reference.
- A REF must have a zone that refers it.
- The NULLIF clause can have external references only if the REF zone is non-integer.

Recordlength
Specifies the length of the entire record, where null-indicator bytes are included if they exist, and the record delimiter is excluded if it exists.

The value is a constant integer.

The value can also be an internal reference to the reference zone in the layout definition.

There is no default value.

You can use plus signs and minus signs for an internal reference as follows:

```
RECORDLENGTH @1 + 10
RECORDLENGTH @1 - 10
```

IGNOREZERO or TRIM_NULLS
Specifies whether the binary value zero in CHAR fields and VARCHAR fields is to be discarded.

TRUE or ON
The byte value zero is ignored.

FALSE or OFF
The byte value zero is not ignored. This is the default.

KEEP
The binary value zero is accepted and allowed as part of the input field.

INCLUDEHEADER or COLUMN_NAMES
For an unload operation, whether the table column names are to be included as headers in the external-table file:

TRUE or ON
The table column names are to be included as headers.

FALSE or OFF
The table column names are not to be included as headers. This is the default.

INCLUDEZEROSECONDS
For an unload operation, whether to specify **00** as the value for seconds when no value for seconds is available:

TRUE or ON
Specify **00** as the value for seconds.

FALSE or OFF
Do not specify a value for seconds. This is the default.

INCLUDEHIDDEN
For a load operation, specify whether hidden column values are present in a data file.

The `INCLUDEHIDDEN` option works when you are creating an external table by using the LIKE or SAMEAS clause, and base table has hidden columns.

TRUE
A data file contains values against hidden column.

FALSE
A data file does not contain values against hidden column. This is the default. You can change the default value by using the registry variable `DB2_EXTBL_INCLUDE_HIDDEN_COLS`.

LFINSTRING
Specifies how to interpret an unescaped line-feed (sometimes called an LF or newline) character within string data:

TRUE or ON
An unescaped LF character is interpreted as a record delimiter only if it is in the last field of a record; otherwise, it is treated as data. To cause an LF character that is in the last field of a record to be treated as data, enclose the value of that field in single or double quotation marks.

FALSE or OFF
An unescaped LF character is interpreted as a record delimiter regardless of its position. This is the default.

This option is not supported for unload operations.

Attention: This SQL compatibility enhancement is only available in Db2 Version 11.5 Mod Pack 2 and later versions.

LOGDIR or ERROR_LOG
The directory to which the following files are written:

<database>.<schema>.<external-table-name>.<file-name>.<application-handle>.<id>.bad
A file containing rejected records (that is, records that could not be processed).

<database>.<schema>.<external-table-name>.<file-name>.<application-handle>.<id>.log
A log file.

The default is the directory to which the external-table file is written. If the length of the name that is constructed for a .bad or .log file would exceed the allowed maximum, the name of the file that contains the external table (indicated by `<file-name>`) is truncated so that the maximum is not exceeded.

If a .log or .bad file is generated while carrying out an operation on a partition, the name of the generated file is suffixed with a period followed by the 3-digit partition number.

MAXERRORS or MAX_ERRORS
For a load operation, the threshold for the number of rejected records at which the system stops processing and immediately rolls back the load. The default is 1 (that is, a single rejected record results in a rollback).

For fixed-length format, the following conditions apply:

- The parser reports errors for each field or zone rather than one error for the row.
- Multiple errors can be reported for the same row.
- When the parser detects an error in a field or zone, it recovers by using the field length or zone length. It then continues from the next field or zone until the end of record is reached, or an unrecoverable error occurs, or the MaxErrors limit is reached.
- Unrecoverable errors include the following errors:
  - RecordLength mismatch.

- RecordDelimiter is not found.
- The RecordLength value is not valid, that is, the value is a negative value or zero.
- The zone length is not valid, that is, the value is a negative value.
- The UTF-8 initial byte is not valid.
- The UTF-8 continuation bytes are not valid.

MULTIPARTSIZEMB

When the DB2_ENABLE_COS_SDK registry variable is set to ON, Db2 remote storage communication with cloud object storage is facilitated through an embedded vendor COS SDK which allows Db2 to stream objects/files to cloud object storage in multiple parts (aka 'multipart upload'). This parameter specifies the part size for multipart upload, in megabytes (MB), for the file being unloaded, and overrides the value specified in the MULTIPARTSIZEMB dbm config parameter. This option is available starting in Version 11.5 Modification Pack 7, in Linux (x86) environments only.

MAXROWS or MAX_ROWS

If set to a positive integer, this specifies the maximum number of records (rows) in the external table that are to be processed. If set to 0 (the default), there is no limit and all rows are processed. During a load operation, if MAXROWS is set to a positive value, after that number of rows are processed, regardless of whether some of the rows were rejected or skipped, the system ends the load operation and commits all inserted records.

MERIDIANDELIM

A single-byte character that separates the seconds component from the AM token or PM token in the 12-hour delimited and undelimited formats of a time value.

The default delimiter is a space (`' '`).

Between the seconds component and the AM token or PM token, a delimiter is not required. For example, both of the following values are valid:

```
1:02:46.12345 AM
1:02:46.12345AM
```

NOLOG

Specifies whether the `.log` file for the external table is created.

This option does not apply to `.bad` files.

Possible values are:

TRUE
No `.log` file is created.

FALSE
The `.log` file is created.

This is the default.

NULLVALUE or NULL_VALUE

The UTF-8 string of at most 4 bytes that is to be used to indicate a null value. The default is `'NULL'`.

PARTITION

If the Database Partitioning Feature (DPF) is enabled for the database, an external table can be partitioned into several files. The name of each of the data files that comprise an external table are suffixed with a period followed by a 3-digit number from 000 to 999 that indicates the number of the partition. For example, if an external table with the name **dataFile.txt** is divided into three partitions, the files that comprise it have the names **dataFile.txt.000**, **dataFile.txt.001**, and **dataFile.txt.002**. These files must be accessible from all members.

For a partitioned external table, the PARTITION option specifies to which partition or partitions the statement applies:

PARTITION ALL
The statement applies to all of the partitions that comprise the external table. For an unload operation, this is the only value that is allowed.

PARTITION (*n* TO *n*)
The statement applies to all of the partitions in the specified range, for example, **PARTITION (54 TO 62)**.

PARTITION (*n*,*n*,...)
The statement applies only to the specified partition or partitions, for example, **PARTITION (53)** or **PARTITION (51,57,58)**. If more than one partition number is specified, they must be in ascending order (sqlcode SQL0263N with SQLSTATE=42615) and there can be no duplicates (sqlcode SQL0265N with SQLSTATE=42615).

If a .log or .bad file is generated while carrying out an operation on a partitioned external table, the name of the generated file is suffixed with a period followed by the 3-digit partition number.

If the DPF is enabled and the PARTITION option is not specified, the external table is treated as single-partitioned table on the coordinator member. The names of the external table file and the .log and .bad files are not suffixed with a partition number.

If the DPF is not enabled, the PARTITION option can be specified, but only with the value ALL, (0 to 0), or (0) (SQL0644N). It will have no effect.

The REMOTESOURCE and PARTITION options are mutually exclusive.

QUOTEDNULL
For a load operation, how to interpret a value that is enclosed in single or double quotation marks and that matches the null value specified by the NULLVALUE or NULL_VALUE option (for example, "NULL" or 'NULL'):

TRUE or ON
The value is interpreted as a null value. This is the default.

FALSE or OFF
The value is interpreted as a character string.

QUOTEDVALUE or STRING_DELIMITER
Whether data values are enclosed in quotation marks:

SINGLE or YES
Data values are enclosed in single quotation marks (').

DOUBLE
Data values are enclosed in double quotation marks (").

NO
Data values are not enclosed in quotation marks. This is the default.

RECORDDELIM or RECORD_DELIMITER
The string literal that is to be interpreted as a row (record) delimiter. The default is `'\n'`.

When CRINSTRING is set to TRUE, RECORDDELIM cannot contain a CR ('\r') character - with the sole exception of a CRLF ('\r\n') delimiter allowed with CRINSTRING for text format only.

REMOTESOURCE
Where the external-table file resides and, if it resides on a remote system, whether the file data is to be compressed:

LOCAL
The file resides on the local server, that is, the system that hosts the database. This is the default.

YES
The file resides on a system other than the local server. For example, specify YES if a client system is connected to the database and the file resides on that system. File data is not compressed before it is transferred.

GZIP
Similar to YES, except that the file data is compressed using the GZIP compression algorithm before the data is transferred, and is decompressed after it is received. This improves overall performance when a large amount of compressible data is being transferred.

LZ4
Similar to YES, except that the file data is compressed using the LZ4 compression algorithm before the data is transferred, and is decompressed after it is received. This improves overall performance when a large amount of compressible data is being transferred.

The REMOTESOURCE, SWIFT, and S3 options are mutually exclusive.

Attention: In the container-only release of Db2 Version 11.5 Mod Pack 1 or later versions, AZURE is compatible and its options are mutually exclusive with REMOTESOURCE, SWIFT, and S3.

The REMOTESOURCE and PARTITION options are mutually exclusive. The COMPRESS option cannot be specified if the value of the REMOTESOURCE option is GZIP or LZ4.

REQUIREQUOTES
Whether quotation marks are mandatory:

TRUE or ON
Quotation marks are mandatory. The QUOTEDVALUE option must be set to YES, SINGLE, or DOUBLE.

FALSE or OFF
Quotation marks are not mandatory. This is the default.

SKIPROWS or SKIP_ROWS
For a load operation, the number of rows to skip before beginning to load the data. The default is 0. Because skipped rows are processed before they are skipped, a skipped row is still capable of causing a processing error.

SOCKETBUFSIZE
The size, in bytes, of the chunks of data that are read from the source file. Valid values range from 64 KB - 800 MB. If you specify a value outside this range, the value is set to the nearest valid value. The default is 8 MB.

STRICTNUMERIC
For a load operation, how to treat a value that is to be inserted into a DECIMAL field when its scale exceeds that defined for the field:

TRUE or ON
The row containing the value to be inserted is rejected. For example, if any of the following values were to be loaded into a DECIMAL(5,3) field, the row containing that value would be rejected:

```
12.666666666
-98.34496862785
0.00089
```

FALSE or OFF
The row containing the value to be inserted is accepted, and the portion of the decimal fraction that exceeds the scale defined for the field is truncated. This is the default. For example, the values in the previous example would be converted to:

```
12.666
-98.344
0.000
```

SWIFT
Specifies that the source data file is located in a Swift object store. The REMOTESOURCE, SWIFT, and S3 options are mutually exclusive. Use the DATAOBJECT option to specify the file name.

Attention: In the container-only release of Db2 Version 11.5 Mod Pack 1 or later versions, AZURE is compatible and its options are mutually exclusive with REMOTESOURCE, SWIFT, and S3.

Syntax:

```
SWIFT (endpoint, authKey1, authKey2, bucket)
```

where:

endpoint
A character string that specifies the URL of the SWIFT web service.

authKey1
A character string that specifies the access ID or username of the Swift open stack account used to validate the user.

authKey2
A character string that specifies the password of the Swift open stack account used to validate the user.

bucket
The name of the Swift open stack container (bucket) in which the file resides.

Example:

```
CREATE EXTERNAL TABLE exttab1(a int) using
  (dataobject 'datafile1.dat'
   swift('https://dal05.objectstorage.softlayer.net/auth/v1.0/',
    'XXXOS123456-2:xxx123456',
    'b207c6e974020737d92174esdf6d5be9382aa4c335945a14eaa9172c70f8df16',
    'my_dev'
   )
  )
```

S3

Specifies that the source data file is located in an S3 compatible object store. The REMOTESOURCE, SWIFT, and S3 options are mutually exclusive. Use the DATAOBJECT option to specify the file name.

Attention: In the container-only release of Db2 Version 11.5 Mod Pack 1 or later versions, AZURE is compatible and its options are mutually exclusive with REMOTESOURCE, SWIFT, and S3.

Syntax:

```
S3 (endpoint, authKey1, authKey2, bucket)
```

where:

endpoint
A character string that specifies the URL of the S3 compatible web service.

authKey1
A character string that specifies the S3 access key ID of the access keys used to validate the user and all user actions. For IBM Cloud Object Storage, this is the access key ID from the HMAC credentials.

authKey2
A character string that specifies the S3 secret key of the access keys that are used to validate the user and all user actions. For IBM Cloud Object Storage, this is the secret access key from the HMAC credentials.

bucket
The name of the S3 bucket in which the file resides.

Note: For IBM Cloud Object Storage, to create HMAC credentials, when creating new service credentials, specify `{"HMAC:true}` in the **Add Inline Configuration Parameters** field.

Example using AWS S3:

```
CREATE EXTERNAL TABLE exttab2(a int) using
  (dataobject 'datafile2.dat'
   s3('s3.amazonaws.com',
    'XXXOS123456-2:xxx123456',
    'bs07c6e974040737d92174e5e96d5be9382aa4c33xxx5a14eaa9172c70f8df16',
    'my_dev'
   )
  )
```

Example using IBM Cloud Object Storage:

```
CREATE EXTERNAL TABLE exttab2(a int) using
  (dataobject 'datafile2.dat'
   s3('s3-api.us-geo.objectstorage.softlayer.net',
    '1a2bkXXXsaddntLo0xX0',
    'XXxxiEPjJ7T7WBUz74E6abcdABCDE8Q7RgU4gYY9',
    'my_dev'
   )
  )
```

AZURE

Attention: This feature is available in the container-only release of Db2 Version 11.5 Mod Pack 1 or later versions.

Specifies that the source data file is located in Microsoft Azure Blob Storage. The REMOTESOURCE, SWIFT, S3, and AZURE options are mutually exclusive. Use the DATAOBJECT option to specify the file name. Syntax:

Syntax:

```
AZURE (endpoint, authKey1, authKey2, bucket)
```

where:

endpoint
A character string that specifies the URL of the AZURE web service.

authKey1
A character string that specifies the access ID or username of the Azure Blob Storage account used to validate the user.

authKey2
A character string that specifies the access key of the Azure Blob Storage account used to validate the user.

bucket
The name of the Azure Blob Storage container (bucket) in which the file resides.

Example:

```
CREATE EXTERNAL TABLE exttab1(a int) using
  (dataobject 'datafile1.dat'
   azure('https://my_account.blob.core.windows.net',
    'my_account',
    'lW+oHjmZecPS++IKgThAHlMUOaFUA5C6Z2RlFmc9JPpK34RO/ZIOywzILxJnzGPHz6d/
yDrcQDAwH5wySbOZMQ==',
    'my_bucket'
   )
  )
```

Example using IBM Cloud Object Storage:

```
CREATE EXTERNAL TABLE exttab2(a int) using
  (dataobject 'datafile2.dat'
   s3('s3-api.us-geo.objectstorage.softlayer.net',
    '1a2bkXXXsaddntLo0xX0',
    'XXxxiEPjJ7T7WBUz74E6abcdABCDE8Q7RgU4gYY9',
    'my_dev'
```

```
    )
  )
```

TIMEDELIM
The single-byte character that is to separate time components (hours, minutes, and seconds). The default is ' : '. If TIMEDELIM is set to an empty string, hours, minutes, and seconds must all be specified as two-digit numbers. The TIMESTAMP_FORMAT and TIMEDELIM options are mutually exclusive.

TIMEROUNDNANOS or TIMEEXTRAZEROS

Note: This option applies only to TIMESTAMP columns.

Specifies whether records that contain time values whose non-zero precision exceeds six decimal places are to be accepted (and rounded to the nearest microsecond) or rejected:

TRUE
All records are accepted. Their time values are rounded to the nearest microsecond.

FALSE
Only those records that can be stored without a loss of precision (for example, '08.15.32.123' or '08.15.32.12345600000', but not '08.15.32.1234567') are accepted. All other records are rejected. This is the default.

TIMESTYLE
The time format that is to be used in the data file:

24HOUR
24-hour format, for example 23:55. This is the default.

12HOUR
12-hour format, for example 11:55 PM. An AM or PM token can be preceded by a single space and is not case-sensitive.

The TIMESTYLE option and the TIME_FORMAT or TIMESTAMP_FORMAT option are mutually exclusive.

TIMESTAMP_FORMAT
The format of the timestamp field in the data file. The value can be any of the format strings that are accepted by the “TIMESTAMP_FORMAT ” on page 527. The default is **'YYYY-MM-DD HH.MI.SS'**. The TIMESTAMP_FORMAT option and the TIMEDELIM, DATEDELIM, TIMESTYLE, or DATESTYLE option are mutually exclusive.

TIME_FORMAT
The format of the time field in the data file. The value can be any of the time format strings that are accepted by the “TIMESTAMP_FORMAT ” on page 527. The default is **HH.MI.SS**. The TIME_FORMAT option and a TIMEDELIM or TIMESTYLE option are mutually exclusive.

TRIMBLANKS
How an external table is to treat leading or trailing blanks (that is, leading or trailing space characters) in a string:

LEADING
All leading blanks (that is, blanks that precede the first non-blank character) are removed.

TRAILING
All trailing blanks (that is, blanks that follow the last non-blank character) are removed.

BOTH
All leading and trailing blanks are removed.

NONE
No blanks are removed. This is the default.

When reading data from a file and loading it into an external table:

- If QUOTEDVALUE or STRING_DELIMITER is specified with the values SINGLE, YES, or DOUBLE, leading and trailing blanks within quotation marks are not removed.

- For CHAR and NCHAR data, the values TRAILING or BOTH will not have any effect on trailing blanks, because the string will automatically be re-padded with trailing blanks.

TRUNCSTRING or TRUNCATE_STRING
How the system processes a CHAR or VARCHAR string that exceeds its declared storage size:

TRUE
The system truncates a string value that exceeds its declared storage size.

FALSE
The system returns an error when a string value exceeds its declared storage size. This is the default.

Y2BASE
The year that is the beginning of the 100-year range. Years that are specified as 2 digits are counted from this year. The default is 2000. This option must be specified when DATESTYLE is set to Y2MD, MDY2, DMY2, MONDY2 or DMONY2.

Table 133. Options

Option	Default	Applies to Load	Applies to Unload
Azure **Attention:** This option only applies to the container-only release of Db2 Version 11.5 Mod Pack 1 or later versions.	(no default)	Y	Y
BOOLSTYLE or BOOLEAN_STYLE	1_0	Y	Y
CARDINALITY	(no default)	Y	Y
CCSID	(no default)	Y	Y
COMPRESS	NO	Y	Y
CRINSTRING	FALSE	Y	Y
CTRLCHARS	FALSE	Y	N
DATAOBJECT or FILE_NAME	(no default)	Y	Y
DATEDELIM	'-'	Y	Y
DATETIMEDELIM	A space (' ')	Y	Y
DATESTYLE	YMD	Y	Y
DATE_FORMAT	YYYY-MM-DD	Y	Y
DECIMALDELIM or DECIMAL_CHARACTER	'.'	Y	Y
DELIMITER	'|'	Y	Y
ENCODING	INTERNAL	Y	Y[1]
ESCAPECHAR or ESCAPE_CHARACTER	(no default)	Y	Y
FILLRECORD	FALSE	Y	N
FORMAT or FILE_FORMAT	TEXT	Y	Y
IGNOREZERO or TRIM_NULLS	FALSE	Y	N
INCLUDEHEADER or COLUMN_NAMES	FALSE	N	Y

Table 133. Options (continued)

Option	Default	Applies to Load	Applies to Unload
INCLUDEZEROSECONDS	FALSE	Y	Y
INCLUDEHIDDEN	FALSE	Y	N
LFINSTRING	FALSE	Y	N
LOGDIR or ERROR_LOG	target directory of external-table file	Y	N
MULTIPARTSIZEMB	value specified by the MULTIPARTSIZEMB dbm config parameter.	Y	N
MAXERRORS or MAX_ERRORS	1	Y	N
MAXROWS or MAX_ROWS	0	Y	N
MERIDIANDELIM	A space (' ')	Y	Y
NOLOG	FALSE	Y	Y
NULLVALUE or NULL_VALUE	'NULL'	Y	Y
PARTITION	(no default)	Y	Y
QUOTEDNULL	TRUE	Y	N
QUOTEDVALUE	NO	Y	N
RECORDDELIM or RECORD_DELIMITER	'\n'	Y	N
REMOTESOURCE	LOCAL	Y	Y
REQUIREQUOTES	FALSE	Y	N
SKIPROWS or SKIP_ROWS	0	Y	N
SOCKETBUFSIZE	8 MB	Y	Y
STRICTNUMERIC	FALSE	Y	N
SWIFT	(no default)	Y	Y
S3	(no default)	Y	Y
TIMEDELIM	':'	Y	Y
TIMEROUNDNANOS or TIMEEXTRAZEROS	FALSE	Y	N
TIMESTAMP_FORMAT	'YYYY-MM-DD HH.MI.SS'	Y	Y
TIMESTYLE	24HOUR	Y	Y
TIME_FORMAT	HH.MI.SS	Y	Y
TRIMBLANKS	NONE	Y	Y
TRUNCSTRING or TRUNCATE_STRING	FALSE	Y	N
Y2BASE	2000	Y	N

[1] Only for the values INTERNAL, UTF8, and LATIN9.

AS SELECT STATEMENT
Specifies that, for each column in the derived result table of the fullselect, a corresponding column is to be defined for the table and populated with the results of the query. Each defined column adopts the following attributes from its corresponding column of the result table (if applicable to the data type):

- Column name
- Column description
- Data type, length, precision, and scale
- Nullability

Notes

- Records that cannot be processed (if any) are written to a file with a name of the form:

```
<database>.<schema>.<external-table-name>.<file-name>.<application-handle>.<id>.bad
```

  Errors are logged in a file with a name of the form:

```
<database>.<schema>.<external-table-name>.<file-name>.<application-handle>.<id>.log
```

  These files are located in the directory specified by the LOGDIR or ERROR_LOG option.

  For an operation on a partition, the name of the generated .bad or .log file is suffixed with a period followed by the 3-digit partition number.
- To create, insert into, or drop a named external table, issue a CREATE, INSERT, or DROP statement. You cannot insert into or drop a transient external table.
- Dropping an external table deletes the table definition but does not delete the data file that is associated with the table.

Restrictions

- Remote external table restrictions:
  - It is not allowed within routines
  - It is not allowed with the use of LOAD CURSOR
  - For remote external tables (that is, for external tables are not located in a Swift or S3 object store and for which the REMOTESOURCE option is set to a value other than LOCAL):

    Attention: In the container-only release of Db2 Version 11.5 Mod Pack 1 or later versions, AZURE is compatible, with the same remote external table restriction.

    Note: A single query or subquery cannot select from more than one external table at a time, and cannot reference the same external table more than once. If necessary, combine data from several external tables into a single table and use that table in the query.

    In addition, a union operation cannot involve more than one external table.
- External tables can be queried only by a user ID defined within the operating system.
- External tables cannot be used by a Db2 instance running on a Windows system.
- Data being loaded must be properly formatted.
- You cannot delete, truncate, or update an external table.

Syntax alternatives

The following alternatives are non-standard. They are supported for compatibility with earlier product versions or with other database products.

- SAMEAS can be used in place of LIKE.

- For the REMOTESOURCE option, the values ODBC, JDBC, or OLE-DB can be specified in place of YES.
- If the FORMAT option is set to INTERNAL, the value YES can be specified in place of GZIP for the COMPRESS option.

Examples

- Unload data to an external table:

```
CREATE EXTERNAL TABLE 'order.tbl' USING (DELIMITER '|') AS SELECT * from orders;
```

```
CREATE EXTERNAL TABLE 'export.csv' USING (DELIMITER ',') AS SELECT foo.x, bar.y, bar.dt FROM
foo, bar WHERE foo.x = bar.x;
```

- Load data from an external table:

```
INSERT INTO target SELECT * FROM EXTERNAL 'data.txt' USING (DELIMITER '|');
```

```
INSERT INTO orders SELECT * FROM EXTERNAL 'order.tbl'( order_num INT, order_dt TIMESTAMP)
USING (DELIMITER '|');
```

- Select data from an external table:

```
SELECT * FROM EXTERNAL 'order.tbl' (order_num INT, order_dt TIMESTAMP) USING (DELIMITER '|');
```

```
SELECT * FROM EXTERNAL 'test.txt' LIKE test_table USING (DELIMITER ',');
```

```
SELECT x, y AS dt FROM EXTERNAL 'test.txt' ( x integer, y decimal(18,4) ) USING (DELIMITER
',');
```

CREATE FUNCTION

The CREATE FUNCTION statement is used to register or define a user-defined function or function template at the current server.

There are five different types of functions that can be created using this statement. Each of these is described separately.

- External Scalar. The function is written in a programming language and returns a scalar value. The external executable is registered in the database, along with various attributes of the function.
- External Table. The function is written in a programming language and returns a complete table. The external executable is registered in the database along with various attributes of the function.
- OLE DB External Table. A user-defined OLE DB external table function is registered in the database to access data from an OLE DB provider.
- Sourced or Template. A source function is implemented by invoking another function (either built-in, external, SQL, or source) that is already registered in the database.

  It is possible to create a partial function, called a *function template*, which defines what types of values are to be returned, but which contains no executable code. The user maps it to a data source function within a federated system, so that the data source function can be invoked from a federated database. A function template can be registered only with an application server that is designated as a federated server.
- SQL Scalar, Table or Row. The function body is written in SQL and defined together with the registration in the database. It returns a scalar value, a table, or a single row.
- Aggregate interface. A aggregate interface function is implemented by invoking several external procedures and an external function which are referred to as component routines.

The CREATE FUNCTION statement can be submitted in obfuscated form. In an obfuscated statement, only the function name and its parameters are readable. The rest of the statement is encoded in such

a way that is not readable but can be decoded by the database server. Obfuscated statements can be produced by calling the DBMS_DDL.WRAP function.

CREATE FUNCTION (aggregate interface)

The CREATE FUNCTION (aggregate interface) statement is used to register a user-defined aggregate function at the current server.

An aggregate function returns a single value that is the result of an evaluation of a set of like values, such as those in a column within a set of rows.

Invocation

This statement can be embedded in an application program or issued in dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges that are held by the authorization ID of the statement must include at least one of the following authorities:

- IMPLICIT_SCHEMA authority on the database, if the schema name of the function does not refer to an existing schema
- CREATEIN privilege on the schema, if the schema name of the function refers to an existing schema
- DBADM authority

The privileges that are held by the authorization ID of the statement must also include EXECUTE privilege on the following routines, if the authorization ID of the statement does not have DATAACCESS authority:

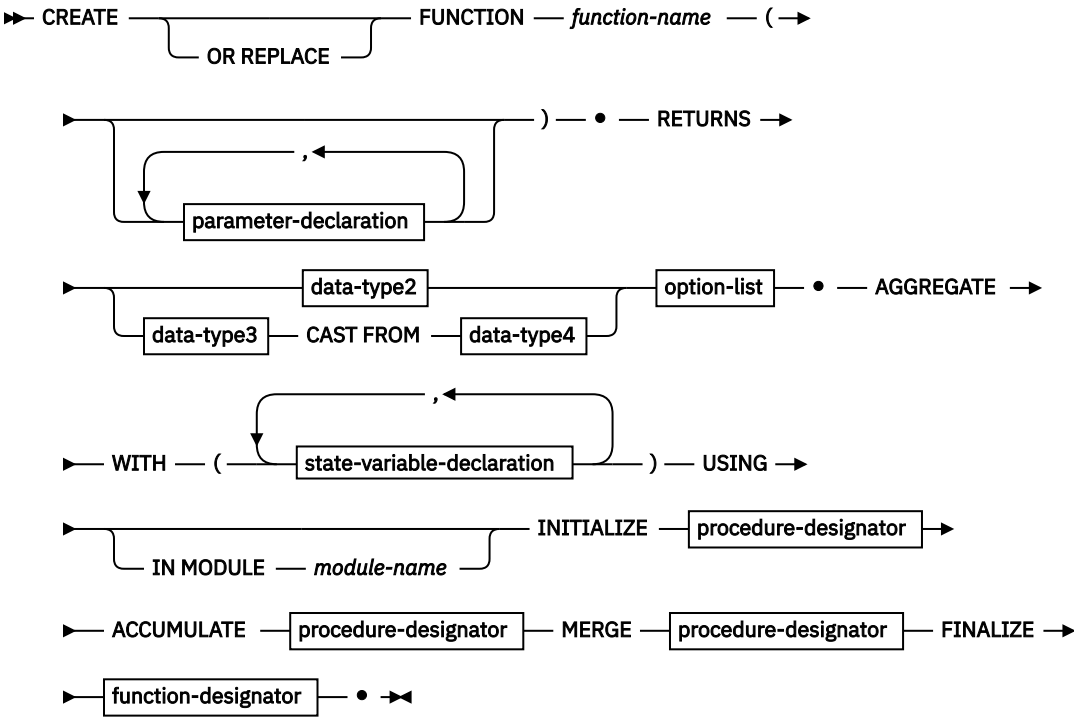
- INITIATE
- ACCUMULATE
- MERGE
- FINALIZE

Group privileges, except for PUBLIC, are not considered on any dependent object specified in the CREATE FUNCTION statement.

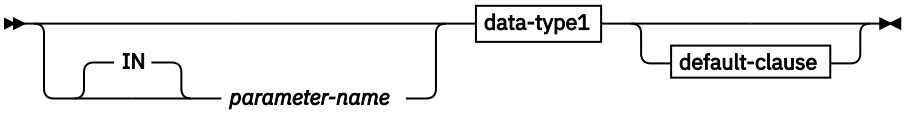
To replace an existing function, the authorization ID of the statement must be the owner of the existing function (SQLSTATE 42501).

If the SECURED option is specified, the authorization ID of the statement must include SECADM or CREATE_SECURE_OBJECT authority (SQLSTATE 42501).

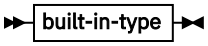
Syntax



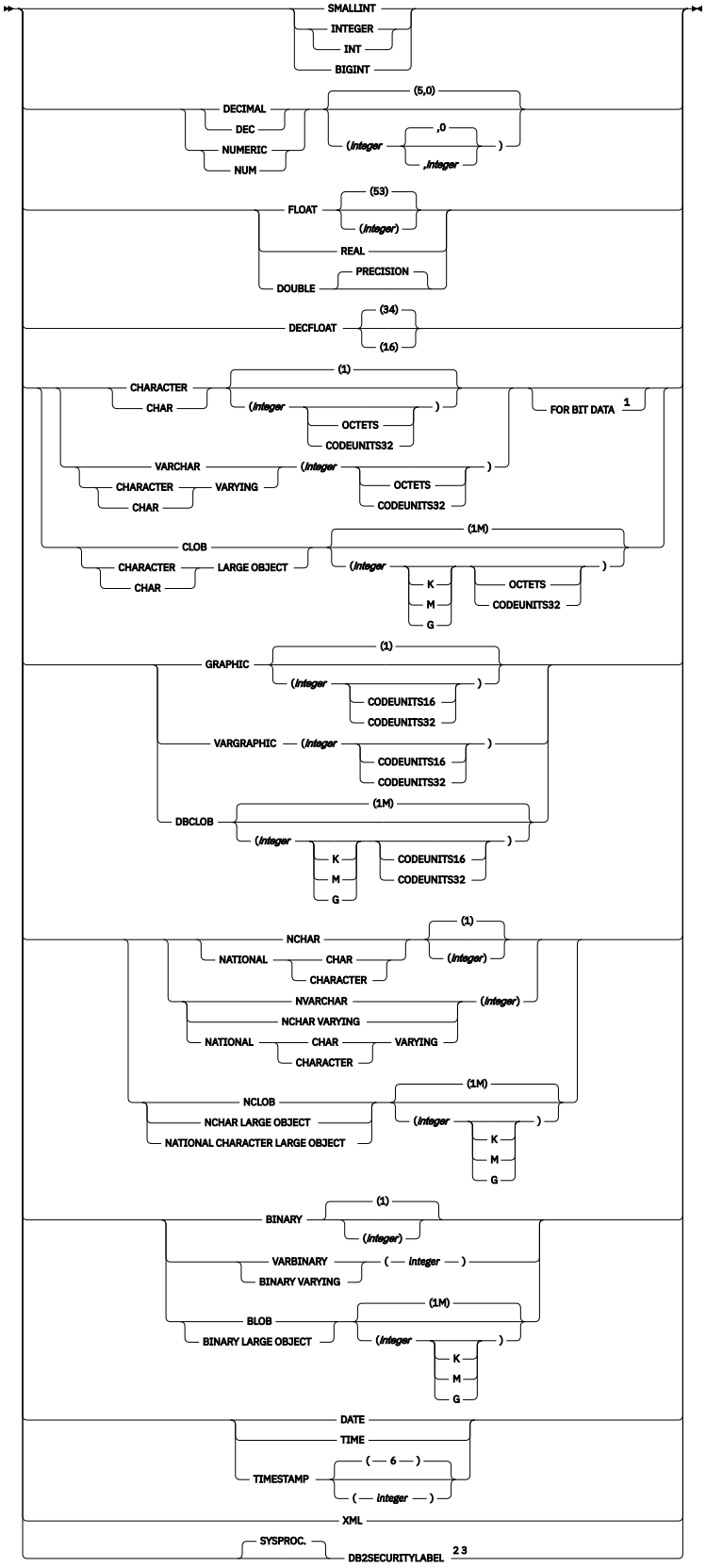
parameter-declaration



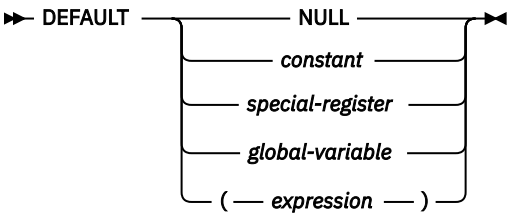
data-type1, data-type2, data-type3, data-type4, data-type5



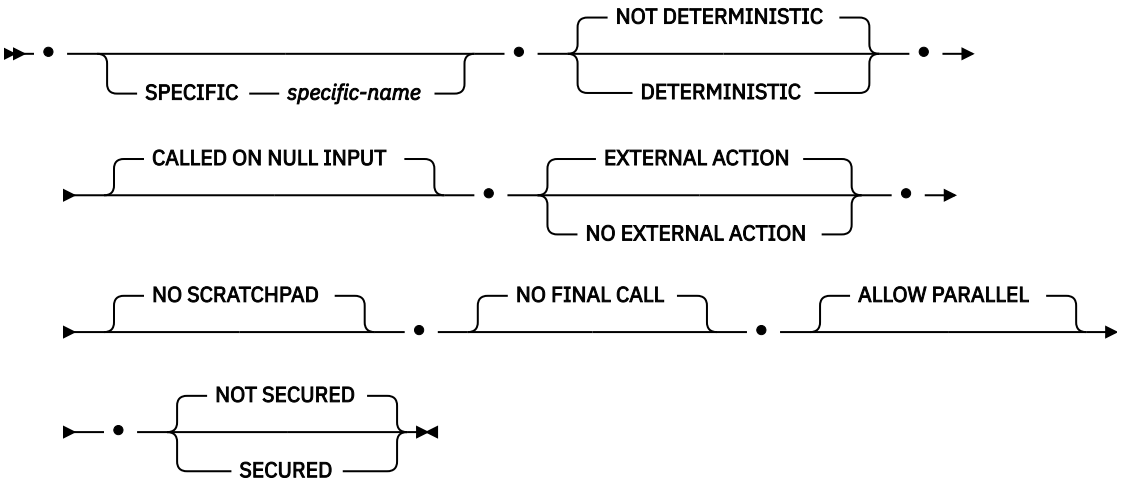
built-in-type



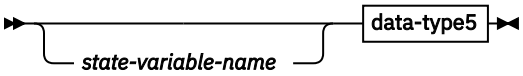
default-clause



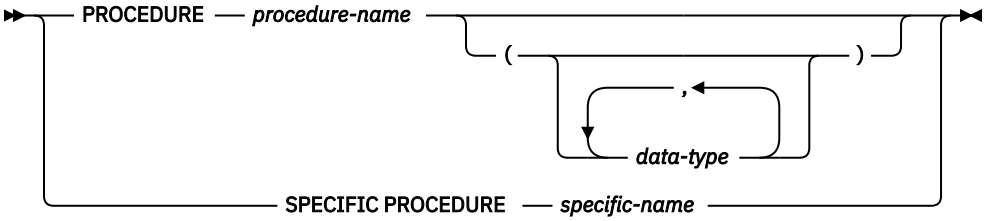
option-list



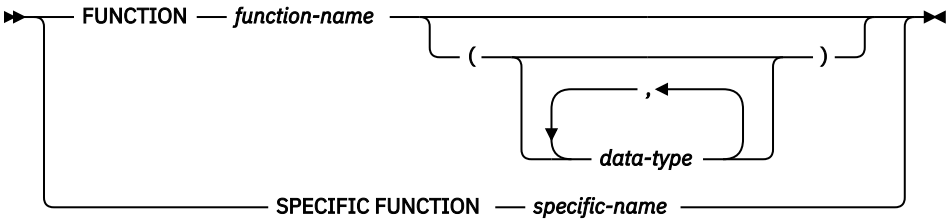
state-variable-declaration



procedure-designator



function-designator



Notes:

[1] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

[2] DB2SECURITYLABEL is the built-in distinct type that must be used to define the row security label column of a protected table.

[3] For a column of type DB2SECURITYLABEL, NOT NULL WITH DEFAULT is implicit and cannot be explicitly specified (SQLSTATE 42842). The default value for a column of type DB2SECURITYLABEL is the session authorization ID's security label for write access.

Description

OR REPLACE

Specifies to replace the definition for the function if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog, with the exception that privileges that were granted on the function are not affected. This option can be specified only by the owner of the object. This option is ignored if a definition for the function does not exist at the current server. An existing function can be replaced if either of the following conditions apply:

- The specific name and function name of the new definition must be the same as the specific name and function name of the old definition
- The signature of the new definition must match the signature of the old definition

Otherwise, a new function is created.

If the function is referenced in the definition of a row permission or a column mask, the function cannot be replaced (SQLSTATE 42893).

function-name

Names the function that is being defined. It is a qualified or unqualified name that designates a function. The unqualified form of *function-name* is an SQL identifier. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a *schema-name* followed by a period and an SQL identifier. The qualified name must not be the same as the data type of the first parameter, if that first parameter is a structured type.

The function signature must not identify a function or method described in the catalog (SQLSTATE 42723). When the name is being assessed, the number of parameters and the data type of each parameter (without regard for any length, precision or scale attributes of the data type) is considered. The unqualified name, together with the number and data types of the parameters, must be unique within its schema. However, the name does not need to be unique across schemas.

If a two-part name is specified, the *schema-name* cannot begin with "SYS" (SQLSTATE 42939).

A number of names that are used as keywords in predicates are reserved for system use, and cannot be used as a *function-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators.

In general, the same name can be used for more than one function if the function signatures are different.

Although it is not prohibited, do not give an aggregate interface function the same name as a built-in function, unless it is an intentional override. Examples of such functions are MAX, MIN, and AVG. Creating a user-defined function that has different behavior, yet the same name, and consistent arguments as a built-in scalar or aggregate function, can lead to problems. Examples include:

- Problems in dynamic SQL statements
- Static SQL applications can fail when they are rebound
- Applications might appear to run successfully but provide a different result

(*parameter-declaration*,...)

Identifies the number of input parameters of the function, and specifies the mode, name, data type, and optional default value of each parameter. One entry in the list must be specified for each parameter that the function expects to receive. Up to 90 parameters can be specified (SQLSTATE 54023).

You can register a function that has no parameters; the parentheses must still be coded, with no intervening data types. For example:

```
CREATE FUNCTION WOOFER() ...
```

Two functions with identical names in the same schema cannot have the same type for all corresponding parameters. Lengths, precisions, and scales are not considered in this type comparison. Therefore, CHAR(8) and CHAR(35) are considered to be the same type, as are DECIMAL(11,2) and DECIMAL (4,3). A weakly typed distinct type that is specified for a parameter is considered to be the same data type as the source type of the distinct type. For a Unicode database, CHAR(13) and GRAPHIC(8) are considered to be the same type. Further bundling of types causes them to be treated as the same type for this purpose, such as DECIMAL and NUMERIC. A duplicate signature returns an error (SQLSTATE 42723).

IN
Identifies the parameter as an input parameter to the function. Any changes that are made to the parameter within the function are not available to the invoking context when control is returned.

parameter-name
Specifies an optional name for the parameter. The name cannot be the same as any other *parameter-name* in the parameter list (SQLSTATE 42734).

data-type1
Specifies the data type of the parameter. The data type can be a built-in data type. For a complete description of each built-in data type, see "CREATE TABLE " on page 1351. The data type must not be XML, CLOB, DBCLOB, or BLOB. (SQLSTATE 42815). The data type must not be a distinct type (SQLSTATE 42611).

DEFAULT
Specifies a default value for the parameter. The default can be a constant, a special register, a global variable, an expression, or the keyword NULL. The special registers that can be specified as the default are the same as those special registers that can be specified for a column default (see *default-clause* in the "CREATE TABLE " on page 1351). Other special registers can be specified as the default by using an expression.

The expression can be any expression of the type described in "Expressions" on page 132. If a default value is not specified, the parameter has no default and the corresponding argument cannot be omitted on invocation of the procedure. The maximum size of the expression is 64 KB.

The default expression must not modify SQL data (SQLSTATE 428FL or SQLSTATE 429BL). The expression must be assignment compatible to the parameter data type (SQLSTATE 42821).

RETURNS
This mandatory clause identifies the output of the function.

data-type2
Specifies the data type of the output.

In this case, the same considerations apply as described in *data-type1* for function parameters.

The data type must not be a distinct type (SQLSTATE 42611).

data-type3* CAST FROM *data-type4
Specifies the data type of the output.

This form of the RETURNS clause returns a different data type to the invoking statement than the data type that was returned by the function code of the FINALIZE function. Example:

```
CREATE FUNCTION GET_HIRE_DATE(CHAR(6))
  RETURNS DATE CAST FROM CHAR(10)
  ...
```

In the preceding code, the function code returns a CHAR(10) value to the database manager. The database manager then converts it to a DATE and passes that value to the invoking statement. The *data-type4* must be castable to the *data-type3* parameter. If it is not castable, an error (SQLSTATE 42880) is returned.

The length, precision, or scale for *data-type3* can be inferred from *data-type4*. Although you can specify the length, precision, or scale for parameterized types for *data-type3*, it is not necessary. Instead, empty parentheses can be used. For example, VARCHAR() can be used).

FLOAT() cannot be used (SQLSTATE 42601) since parameter value indicates different data types (REAL or DOUBLE).

Distinct types are not valid as the type specified in *data-type3* or *data-type4* (SQLSTATE 42815).

The cast operation is also subject to runtime checks that might result in conversion errors.

built-in-type
See "CREATE TABLE " on page 1351 for the description of built-in data types.

option-list

SPECIFIC *specific-name*
Provides a unique name for the instance of the function that is being defined. This specific name can be used when sourcing on this function, dropping the function, or commenting on the function. It can never be used to invoke the function. The unqualified form of *specific-name* is an SQL identifier. The qualified form is a *schema-name* followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another function instance or method specification that exists at the application server; otherwise, an error (SQLSTATE 42710) is returned.

The *specific-name* can be the same as an existing *function-name*.

If no qualifier is specified, the qualifier that was used for *function-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *function-name* or an error (SQLSTATE 42882) is returned.

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmssxxx.

DETERMINISTIC or NOT DETERMINISTIC
This optional clause specifies whether the function always returns the same results for given argument values (DETERMINISTIC) or whether the function depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC function must always return the same result from successive invocations with identical inputs. Optimizations taking advantage of the fact that identical inputs always produce the same results are prevented by specifying NOT DETERMINISTIC. An example of a NOT DETERMINISTIC function would be a random-number generator. An example of a DETERMINISTIC function would be a function that determines the square root of the input. If DETERMINISTIC is specified, then the INITIALIZE, ACCUMULATE, MERGE, and FINALIZE routines that are identified must also be DETERMINISTIC (SQLSTATE 428IA).

CALLED ON NULL INPUT
CALLED ON NULL INPUT always applies to aggregate interface functions. In other words, the function is called regardless of whether any argument's set of values are all null. The INITIALIZE, ACCUMULATE, and MERGE procedures that are identified are also always CALLED ON NULL INPUT since they are procedures. The FINALIZE function that is identified must also be CALLED ON NULL INPUT (SQLSTATE 428IA). Any parameter of the component routines can return a null value or a normal (non-null) value. Responsibility for testing for null argument values lies with the component routine.

EXTERNAL ACTION or NO EXTERNAL ACTION
Specifies whether the function has actions that change the state of an object that the database manager does not manage. An example of an external action is sending a message or writing a record to a file. The default is EXTERNAL ACTION.

EXTERNAL ACTION
Specifies that the function has actions that change the state of an object that the database manager does not manage.

A function with external actions might return incorrect results if the function is executed by parallel tasks. For example, if the function sends a note for each initial call to it, one note is sent for each parallel task instead of once for the function.

NO EXTERNAL ACTION
Specifies that the function does not have actions that change the state of an object that the database manager does not manage. The database manager uses this information during optimization of SQL statements. If NO EXTERNAL ACTION is specified, then the INITIALIZE, ACCUMULATE, MERGE, and FINALIZE routines that are identified must also be NO EXTERNAL ACTION (SQLSTATE 428IA).

NO SCRATCHPAD
This optional clause can be used to specify whether a scratchpad is to be provided for an external function. NO SCRATCHPAD is allowed for an aggregate interface function. The INITIALIZE, ACCUMULATE, and MERGE procedures identified always have NO SCRATCHPAD since they are procedures. The FINALIZE function that is identified must also be NO SCRATCHPAD (SQLSTATE 428IA).

NO FINAL CALL
This optional clause specifies whether a final call is to be made to an external function. NO FINAL CALL is allowed for an aggregate interface function. The INITIALIZE, ACCUMULATE, and MERGE procedures that are identified always are NO FINAL CALL since they are procedures. The FINALIZE function that is identified must also be NO FINAL CALL (SQLSTATE 428IA).

ALLOW PARALLEL
This optional clause specifies whether, for a single reference to the function, the invocation of the function can be parallelized. For aggregate interface functions, only ALLOW PARALLEL is supported.

NOT SECURED or SECURED
Specifies whether the function is considered secure for row and column access control. The default is NOT SECURED.

NOT SECURED
Indicates that the function is not considered secure. When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled and column level access control is activated for its table (SQLSTATE 428HA). This rule applies to the non-secure user-defined functions that are invoked anywhere in the statement.

SECURED
Indicates that the function is considered secure.

AGGREGATE
This mandatory clause indicates that the CREATE FUNCTION statement is being used to register a user-defined aggregate function.

WITH
This clause is used to specify the state variables that are available between the stages of aggregate function processing. There must be at least one state variable defined.

state-variable-name
Specifies an optional name for the state variable. The name cannot be the same as any other *state-variable-name* in the list of state variables for this function definition (SQLSTATE 42734).

data-type5
Specifies the data type of a state variable.

The considerations that apply for the parameters of functions (as described in *data-type1*) also apply for function parameters. However, XML, CLOB, DBCLOB, and BLOB data types are not allowed as state variable data types (SQLSTATE 42611). The data type for *data-type5* must not be a distinct type (SQLSTATE 42611).

The total byte count of the state variables cannot be bigger than 32677 using the table row size approach to counting bytes (SQLSTATE 42611).

USING
This mandatory clause specifies how the aggregate function processing is implemented. The USING clause specifies a set of user-defined procedures and a user-defined scalar function that implement the phases of aggregate function processing.

IN MODULE *module-name*
This optional clause specifies that the three procedures and one function that are specified in the INITIALIZE, ACCUMULATE, MERGE, and FINALIZE clauses are in module *module-name*. If this clause is specified, the *procedure-name*, *function-name*, or *specific-name* specified in *procedure-designator* and *function-designator* must be unqualified names (SQLSTATE 42601).

INITIALIZE *procedure-designator*
Uniquely identifies a single procedure that implements the initialization phase of the aggregation.

The procedure that is selected must have output parameters only. The number of output parameters must be the same as the number of state variables specified in the AGGREGATE WITH clause (SQLSTATE 428IA). The data type of each output parameter in the procedure that is selected must have the exact same type as the corresponding data type specified in the AGGREGATE WITH clause (SQLSTATE 428IA).

Specify the following combinations of options for the selected procedure (SQLSTATE 428IA):

- LANGUAGE C and NO SQL
- LANGUAGE JAVA and NO SQL
- LANGUAGE SQL and CONTAINS SQL

The procedure must exist when this statement is run, unless the AUTO_REVAL database configuration parameter is set to DEFERRED_FORCE.

ACCUMULATE *procedure-designator*
Uniquely identifies a single procedure that implements the accumulate phase of the aggregation.

The procedure that is selected must meet the following criteria (SQLSTATE 428IA):

- The procedure that is selected must first have the same number of input-only parameters as the number of the parameters specified in the aggregation function.
- The procedure that is selected must then have the same number of INOUT parameters as the number of the state variables specified in the AGGREGATE WITH clause.

The data type of each input-only parameter in the procedure that is selected must have the exact same type as the corresponding data type specified in *parameter-declaration* (SQLSTATE 428IA). The data type of each INOUT parameter in the procedure that is selected must have the exact same type as the corresponding data type specified in the AGGREGATE WITH clause (SQLSTATE 428IA).

Specify the following combinations of options for the selected procedure (SQLSTATE 428IA):

- LANGUAGE C and NO SQL
- LANGUAGE JAVA and NO SQL
- LANGUAGE SQL and CONTAINS SQL

The procedure must exist when this statement is run, unless the AUTO_REVAL database configuration parameter is set to DEFERRED_FORCE.

MERGE *procedure-designator*
Uniquely identifies a single procedure that implements the merge phase of the aggregation.

The procedure that is selected must meet the following criteria (SQLSTATE 428IA):

- The procedure that is selected must first have the same number of input-only parameters as the number of the state variables specified in the AGGREGATE WITH clause.
- The procedure that is selected must then have the same number of INOUT parameters as the number of the state variables specified in the AGGREGATE WITH clause.

The data type of each input-only parameter in the procedure that is selected must have the exact same data type as the corresponding data type specified in the AGGREGATE WITH clause (SQLSTATE 428IA). The data type of each INOUT parameter in the procedure that is selected must have the exact same type as the corresponding data type specified in the AGGREGATE WITH clause (SQLSTATE 428IA).

Specify the following combinations of options for the selected procedure (SQLSTATE 428IA):

- LANGUAGE C and NO SQL
- LANGUAGE JAVA and NO SQL
- LANGUAGE SQL and CONTAINS SQL

The procedure must exist when this statement is run, unless the AUTO_REVAL database configuration parameter is set to DEFERRED_FORCE.

FINALIZE *function-designator*
Uniquely identifies a single user-defined scalar function that implements the final result phase of the aggregation.

The function that is selected must have the same number of input-only parameters as the number of the state variables specified in the AGGREGATE WITH clause (SQLSTATE 428IA). The data type of each input-only parameter in the function that is selected must have the exact same data type as the corresponding data type specified in the AGGREGATE WITH clause (SQLSTATE 428IA). The output data type of the function that is selected must have the exact same type as the output data type specified in the RETURNS clause (SQLSTATE 428IA).

Specify the following combinations of options for the selected procedure (SQLSTATE 428IA):

- LANGUAGE C and NO SQL
- LANGUAGE JAVA and NO SQL
- LANGUAGE SQL and CONTAINS SQL

The function must exist when this statement is run, unless the AUTO_REVAL database configuration parameter is set to DEFERRED_FORCE.

In the descriptions for the INITIALIZE, ACCUMULATE, MERGE, and FINALIZE routines, exact same data type means that lengths, precisions, scales, string units, and CCSIDs are considered in this type comparison. Therefore, the following data types are considered different:

- CHAR(8) and CHAR(35)
- VARCHAR(10 OCTETS) and VARCHAR(10 CODEUNIT32)
- DECIMAL(11,2) and DECIMAL (4,3)

A weakly typed distinct type is considered to be a different data type as the source type of the distinct type. CHAR(13) and GRAPHIC(13) are considered to be different types, even in a Unicode database.

procedure-designator

PROCEDURE *procedure-name*
Identifies a particular procedure, and is valid only if exactly one procedure instance with the name *procedure-name* exists in the schema. The identified procedure can have any number of parameters defined for it. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. If no procedure by this name exists in the named or implied schema, an error (SQLSTATE 42704) is returned. If there is more than one instance of the procedure in the named or implied schema, an error (SQLSTATE 42725) is returned. If a procedure by this name exists and the authorization ID of the statement does not have EXECUTE privilege on this procedure, an error (SQLSTATE 42501) is returned.

PROCEDURE *procedure-name (data-type,...)*
Provides the procedure signature, which uniquely identifies the procedure. The procedure resolution algorithm is not used.

procedure-name
Specifies the name of the procedure. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

(data-type,...)

Values must match the data types that were specified (in the corresponding position) on the CREATE PROCEDURE statement. The number of data types, and the logical concatenation of the data types, is used to identify the specific procedure instance.

If a data type is unqualified, the type name is resolved by searching the schemas on the SQL path.

It is not necessary to specify the length, precision, or scale for the parameterized data types. Instead, an empty set of parentheses can be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601) because the parameter value indicates different data types (REAL or DOUBLE).

If length, precision, or scale is coded, the value must exactly match that specified in the CREATE PROCEDURE statement. When length is specified for character and graphic string data types, the string unit of the length attribute must exactly match that specified in the CREATE PROCEDURE statement.

A type of FLOAT(*n*) does not need to match the defined value for *n* because $0 < n < 25$ means REAL, and $24 < n < 54$ means DOUBLE. Matching occurs based on whether the type is REAL or DOUBLE.

If no procedure with the specified signature exists in the named or implied schema, an error (SQLSTATE 42883) is returned.

If a procedure by this procedure signature exists and the authorization ID of the statement does not have EXECUTE privilege on this procedure, an error (SQLSTATE 42501) is returned.

SPECIFIC PROCEDURE *specific-name*

Identifies a particular procedure, by using the name that is specified or defaulted to at procedure creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific procedure instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is returned. If a procedure by this specific-name exists, and the authorization ID of the statement does not have EXECUTE privilege on this procedure, an error (SQLSTATE 42501) is returned.

FUNCTION *function-name*

Identifies a particular function, and is valid only if exactly one function instance with the name *function-name* exists in the schema. The identified function can have any number of parameters defined for it. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. If no function by this name exists in the named or implied schema, an error (SQLSTATE 42704) is returned. If there is more than one instance of the function in the named or implied schema, an error (SQLSTATE 42725) is returned. If a function by this name exists and the authorization ID of the statement does not have EXECUTE privilege on this function, an error (SQLSTATE 42501) is returned.

function-designator

FUNCTION *function-name (data-type,...)*

Provides the function signature, which uniquely identifies the function. The function resolution algorithm is not used.

function-name

Specifies the name of the function. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

(data-type,...)
Values must match the data types that were specified (in the corresponding position) on the CREATE FUNCTION statement. The number of data types, and the logical concatenation of the data types, is used to identify the specific function instance.

If a data type is unqualified, the type name is resolved by searching the schemas on the SQL path.

It is not necessary to specify the length, precision, or scale for the parameterized data types. Instead, an empty set of parentheses can be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601) because the parameter value indicates different data types (REAL or DOUBLE).

If length, precision, or scale is coded, the value must exactly match that specified in the CREATE FUNCTION statement. When length is specified for character and graphic string data types, the string unit of the length attribute must exactly match that specified in the CREATE FUNCTION statement.

A type of FLOAT(*n*) does not need to match the defined value for *n* because 0 < *n* < 25 means REAL, and 24 < *n* < 54 means DOUBLE. Matching occurs based on whether the type is REAL or DOUBLE.

If no function with the specified signature exists in the named or implied schema, an error (SQLSTATE 42883) is returned.

If a function by this function signature exists and the authorization ID of the statement does not have EXECUTE privilege on this function, an error (SQLSTATE 42501) is returned.

SPECIFIC FUNCTION *specific-name*
Identifies a particular user-defined function, by using the name that is specified or defaulted to at function creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific function instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is returned. If a function by this *specific-name* exists, and the authorization ID of the statement does not have EXECUTE privilege on this function, an error (SQLSTATE 42501) is returned.

Notes

- ***Privileges:*** The definer of a function always receives the EXECUTE privilege on the function. The definer of a function also receives the right to drop the function. The definer of the function is also given the WITH GRANT OPTION if the definer of the function has EXECUTE WITH GRANT OPTION on all of the component routines.

Examples

1. Define an aggregate function that returns the average of a set of numeric values, by using Java routines.

```
CREATE OR REPLACE PROCEDURE myavg_initialize(OUT sum DOUBLE, OUT count INT)
   LANGUAGE JAVA PARAMETER STYLE JAVA
   FENCED NO SQL
   EXTERNAL NAME 'myclass!myavg_initialize';

CREATE OR REPLACE PROCEDURE myavg_accumulate(IN input DOUBLE, INOUT sum DOUBLE, INOUT count
INT)
   LANGUAGE JAVA PARAMETER STYLE JAVA
   FENCED NO SQL
   EXTERNAL NAME 'myclass!myavg_accumulate';

CREATE OR REPLACE PROCEDURE myavg_merge(IN sum DOUBLE, IN count INT,
      INOUT mergesum DOUBLE, INOUT mergecount INT)
   LANGUAGE JAVA PARAMETER STYLE JAVA
```

```
   FENCED NO SQL
   EXTERNAL NAME 'myclass!myavg_merge';

CREATE OR REPLACE FUNCTION myavg_finalize(sum DOUBLE, count INT)
   RETURNS DECFLOAT(34)
   LANGUAGE JAVA PARAMETER STYLE JAVA
   FENCED NO SQL
   EXTERNAL NAME 'myclass!myavg_finalize';

CREATE OR REPLACE FUNCTION myavg(DOUBLE)
   RETURNS DECFLOAT(34)
   AGGREGATE WITH (sum DOUBLE, count INT)
   USING
   INITIALIZE PROCEDURE myavg_initialize
   ACCUMULATE PROCEDURE myavg_accumulate
   MERGE PROCEDURE myavg_merge
   FINALIZE FUNCTION myavg_finalize;
```

2. Define an aggregate function with procedure and function names that are unqualified. Define the aggregate function under the schema FOO. Invoke the aggregate function under schema BAR.

```
SET SCHEMA FOO;

CREATE OR REPLACE FUNCTION myavg(DOUBLE)
   RETURNS DOUBLE
   AGGREGATE WITH (sum DOUBLE, count INT)
   USING
   INITIALIZE PROCEDURE myavg_initialize
   ACCUMULATE PROCEDURE myavg_accumulate
   MERGE PROCEDURE myavg_merge
   FINALIZE FUNCTION myavg_finalize;

SET SCHEMA BAR;

SELECT FOO.myavg(c1) FROM t1;
```

   The database manager looks for procedures with the names of *FOO.myavg_initialize*, *FOO.myavg_accumulate*, *FOO.myavg_merge*, and function with the name of *FOO.myavg_finalize* for the invocation of *FOO.myavg*.

3. Define an aggregate function with specific procedure and function names that are unqualified. Define the aggregate function under the schema FOO. Invoke the aggregate function under schema BAR.

```
SET SCHEMA FOO;

CREATE OR REPLACE FUNCTION myavg(DOUBLE)
   RETURNS DOUBLE
   AGGREGATE WITH (sum DOUBLE, count INT)
   USING
   INITIALIZE SPECIFIC PROCEDURE myavg_initialize1
   ACCUMULATE SPECIFIC PROCEDURE myavg_accumulate1
   MERGE SPECIFIC PROCEDURE myavg_merge1
   FINALIZE SPECIFIC FUNCTION myavg_finalize1;

SET SCHEMA BAR;

SELECT FOO.myavg(c1) FROM t1;
```

   The database manager looks for procedures with the specific names of *FOO.myavg_initialize1*, *FOO.myavg_accumulate1*, *FOO.myavg_merge1*, and function with the specific name of *FOO.myavg_finalize1* for the invocation of *FOO.myavg*.

4. Define an aggregate function without some of the component routines. The aggregate function will be created as invalid and revalidation will be invoked in the next access.

```
UPDATE DB CFG USING AUTO_REVAL DEFERRED_FORCE;

CREATE OR REPLACE PROCEDURE myavg_initialize(OUT sum DOUBLE, OUT count INT)
   LANGUAGE JAVA PARAMETER STYLE JAVA
   FENCED NO SQL
   EXTERNAL NAME 'myclass!myavg_initialize';

CREATE OR REPLACE PROCEDURE myavg_accumulate(IN input DOUBLE, INOUT sum DOUBLE, INOUT count
INT)
   LANGUAGE JAVA PARAMETER STYLE JAVA
   FENCED NO SQL
```

```
   EXTERNAL NAME 'myclass!myavg_accumulate';

-- component routine merge and finalize are missing, the creation is successful and myavg is
invalid:
CREATE OR REPLACE FUNCTION myavg(DOUBLE)
   RETURNS DOUBLE
   AGGREGATE WITH (sum DOUBLE, count INT)
   USING
   INITIALIZE  PROCEDURE myavg_initialize
   ACCUMULATE PROCEDURE myavg_accumulate
   MERGE PROCEDURE myavg_merge
   FINALIZE FUNCTION myavg_finalize;

CREATE OR REPLACE PROCEDURE myavg_merge(IN sum DOUBLE, IN count INT,
      INOUT mergesum DOUBLE, INOUT mergecount INT)
   LANGUAGE JAVA PARAMETER STYLE JAVA
   FENCED NO SQL
   EXTERNAL NAME 'myclass!myavg_merge';

CREATE OR REPLACE FUNCTION myavg_finalize(sum DOUBLE, count INT)
   RETURNS DOUBLE
   LANGUAGE JAVA PARAMETER STYLE JAVA
   FENCED NO SQL
   EXTERNAL NAME 'myclass!myavg_finalize';

-- revalidation of myavg will be invoked and it will be successful
SELECT myavg(c1) FROM t1;
```

5. Define an aggregate function by using a global variable as a default for its parameter. Dropping the global variable invalidates the function.

```
CREATE VARIABLE gv1 DOUBLE;

-- create all 4 component routines (myavg_initialize, myavg_accumulate, myavg_merge,
myavg_finalize) like Example 1
...
...
...

CREATE OR REPLACE FUNCTION myavg(p1 DOUBLE DEFAULT gv1)
   RETURNS DOUBLE
   AGGREGATE WITH (sum DOUBLE, count INT)
   USING
   INITIALIZE SPECIFIC PROCEDURE myavg_initialize
   ACCUMULATE SPECIFIC PROCEDURE myavg_accumulate
   MERGE SPECIFIC PROCEDURE myavg_merge
   FINALIZE SPECIFIC FUNCTION myavg_finalize;

-- the following statement invalidates the function 'myavg'
DROP VARIABLE gv1;
```

6. Define an aggregate function whose component routines use a global variable as a default for its parameter. Dropping the global variable invalidates both the component routine and the aggregate function.

```
CREATE VARIABLE gv1 INT;

CREATE OR REPLACE PROCEDURE myavg_initialize(OUT sum DOUBLE, OUT count INT DEFAULT gv1)
   LANGUAGE JAVA PARAMETER STYLE JAVA
   FENCED NO SQL
   EXTERNAL NAME 'myclass!myavg_initialize';

-- create the remaining 3 component routines (myavg_accumulate, myavg_merge, myavg_finalize)
like Example 1
...
...
...

CREATE OR REPLACE FUNCTION myavg(DOUBLE)
   RETURNS DOUBLE
   AGGREGATE WITH (sum DOUBLE, count INT)
   USING
   INITIALIZE SPECIFIC PROCEDURE myavg_initialize
   ACCUMULATE SPECIFIC PROCEDURE myavg_accumulate
   MERGE SPECIFIC PROCEDURE myavg_merge
   FINALIZE SPECIFIC FUNCTION myavg_finalize;

-- the following statement invalidates both the routine 'myavg_initialize' and the function
```

```
'myavg'
DROP VARIABLE gv1;
```

7. Define an aggregation function, then create a procedure that calls the aggregation function. Next, drop or replace one of the component routines. Either action invalidates both the aggregation function and the procedure that calls it.

```
CREATE OR REPLACE PROCEDURE myavg_initialize(OUT sum DOUBLE, OUT count INT)
   LANGUAGE JAVA PARAMETER STYLE JAVA
   FENCED NO SQL
   EXTERNAL NAME 'myclass!myavg_initialize';

-- create the remaining 3 component routines (myavg_accumulate, myavg_merge, myavg_finalize)
like Example 1
...
...
...

CREATE OR REPLACE FUNCTION myavg(DOUBLE)
   RETURNS DOUBLE
   AGGREGATE WITH (sum DOUBLE, count INT)
   USING
   INITIALIZE SPECIFIC PROCEDURE myavg_initialize
   ACCUMULATE SPECIFIC PROCEDURE myavg_accumulate
   MERGE SPECIFIC PROCEDURE myavg_merge
   FINALIZE SPECIFIC FUNCTION myavg_finalize;

CREATE OR REPLACE PROCEDURE myproc (OUT p1 DOUBLE)
BEGIN
   SET p1 = (SELECT myavg(c1) FROM t1);
END;

-- drop the component routine
-- this action invalidates both the 'myavg' aggregation function and the 'myproc' procedure
that calls it:
DROP PROCEDURE myavg_initialize;

-- re-create the component routine
-- like the DROP statement, this action invalidates both the 'myavg' aggregation function
and the 'myproc' procedure that calls it:
CREATE OR REPLACE PROCEDURE myavg_initialize(OUT sum DOUBLE, OUT count INT)
   LANGUAGE C PARAMETER STYLE C
   FENCED NO SQL
   EXTERNAL NAME 'myclass!myavg_initialize_C_version';

-- revaliation is invoked the next time 'myproc' is accessed
-- both the 'myavg' aggregation function and the 'myproc' procedure are revalidated
CALL myproc(?);
```

8. Use the AUTO_REVAL database configuration parameter to control the invalidation and revalidation semantics.

```
UPDATE DB CFG USING AUTO_REVAL DEFERRED_FORCE;

-- global variable 'gv1' does not exist; the 'myavg_initialize' procedure can be created,
but it is invalid
CREATE OR REPLACE PROCEDURE myavg_initialize(OUT sum DOUBLE, OUT count INT default gv1)
   LANGUAGE JAVA PARAMETER STYLE JAVA
   FENCED NO SQL
   EXTERNAL NAME 'myclass!myavg_initialize';

-- create the remaining 3 component routines (myavg_accumulate, myavg_merge, myavg_finalize)
like Example 1
...
...
...

-- the 'myavg' function can be created, but it is invalid
CREATE OR REPLACE FUNCTION myavg(DOUBLE)
   RETURNS mydouble
   AGGREGATE WITH (sum DOUBLE, count INT)
   USING
   INITIALIZE SPECIFIC PROCEDURE myavg_initialize
   ACCUMULATE SPECIFIC PROCEDURE myavg_accumulate
   MERGE SPECIFIC PROCEDURE myavg_merge
   FINALIZE SPECIFIC FUNCTION myavg_finalize;

-- create the global variable 'gv1'
CREATE VARIABLE gv1 DOUBLE;
```

```
-- revalidation of 'myavg' function and 'myavg_initialize' procedure is invoked;
revalidation is successful
SELECT myavg(c1) FROM t1;

-- change the setting of the AUTO_REVAL database configuration parameter to IMMEDAITE
UPDATE DB CFG USING AUTO_REVAL IMMEDIATE;

-- the CREATE OR REPLACE VARIABLE statement invokes the revalidation for both 'myavg'
function and 'myavg_initialize' procedure
CREATE OR REPLACE VARIABLE gv1 DOUBLE DEFAULT 1.0;
```

9. Create an aggregate function that uses SQL routines to calculate and return the average of a set of numeric values.

```
CREATE OR REPLACE PROCEDURE myavg_initialize(OUT sum DECFLOAT, OUT count INT)
LANGUAGE SQL
CONTAINS SQL
BEGIN

  SET sum = 0;
  SET count = 0;

END @

CREATE OR REPLACE PROCEDURE myavg_accumulate(IN input DECFLOAT, INOUT sum DECFLOAT, INOUT
count INT)
LANGUAGE SQL
CONTAINS SQL
BEGIN

  SET sum = sum + input;
  SET count = count + 1;

END @

CREATE OR REPLACE PROCEDURE myavg_merge(IN sum DECFLOAT, IN count INT,
INOUT mergesum DECFLOAT, INOUT mergecount INT)
LANGUAGE SQL
CONTAINS SQL
BEGIN

  SET mergesum = sum + mergesum;
  SET mergecount = count + mergecount;

END @

CREATE OR REPLACE FUNCTION myavg_finalize(sum DECFLOAT, count INT)
LANGUAGE SQL
CONTAINS SQL
RETURNS DECFLOAT(34)
BEGIN

  RETURN (sum /  count);

END @

CREATE OR REPLACE FUNCTION myavg(DECFLOAT)
RETURNS DECFLOAT(34)
AGGREGATE WITH (sum DECFLOAT, count INT)
USING
INITIALIZE PROCEDURE myavg_initialize
ACCUMULATE PROCEDURE myavg_accumulate
MERGE PROCEDURE myavg_merge
FINALIZE FUNCTION myavg_finalize
@
```

CREATE FUNCTION (external scalar)

The CREATE FUNCTION (External Scalar) statement is used to register a user-defined external scalar function at the current server. A *scalar function* returns a single value each time it is invoked, and is in general valid wherever an SQL expression is valid.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- CREATE_EXTERNAL_ROUTINE authority on the database and at least one of the following authorities:
  - IMPLICIT_SCHEMA authority on the database, if the schema name of the function does not refer to an existing schema
  - CREATEIN privilege on the schema, if the schema name of the function refers to an existing schema
  - SCHEMAADM authority on the schema, if the schema name of the function refers to an existing schema
- DBADM authority

Group privileges are not considered for any table or view specified in the CREATE FUNCTION statement.

To create a not-fenced function, the privileges held by the authorization ID of the statement must also include at least one of the following authorities:

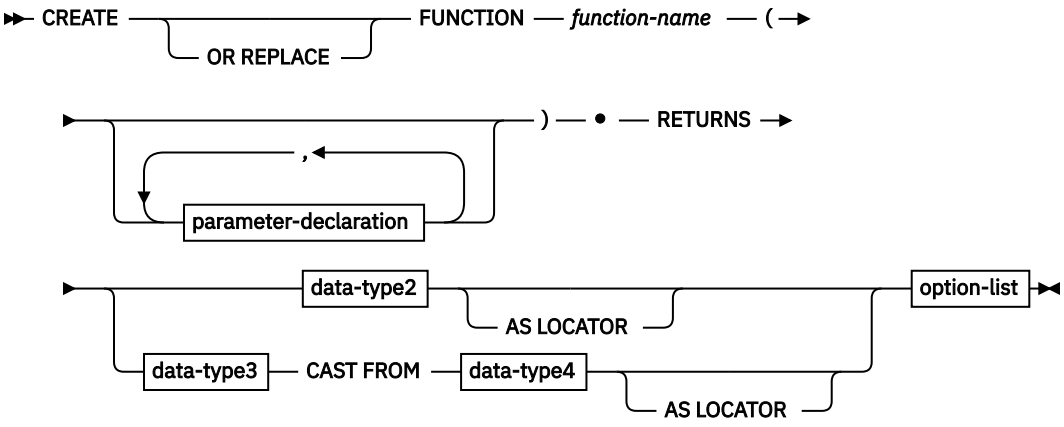
- CREATE_NOT_FENCED_ROUTINE authority on the database
- DBADM authority

To create a fenced function, no additional authorities or privileges are required.

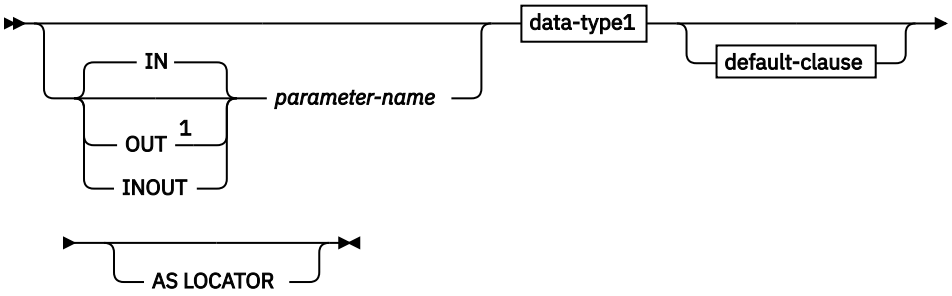
To replace an existing function, the authorization ID of the statement must be the owner of the existing function (SQLSTATE 42501).

If the SECURED option is specified, the authorization ID of the statement must include SECADM or CREATE_SECURE_OBJECT authority (SQLSTATE 42501).

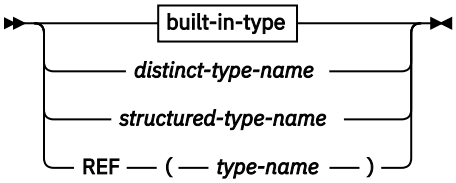
Syntax



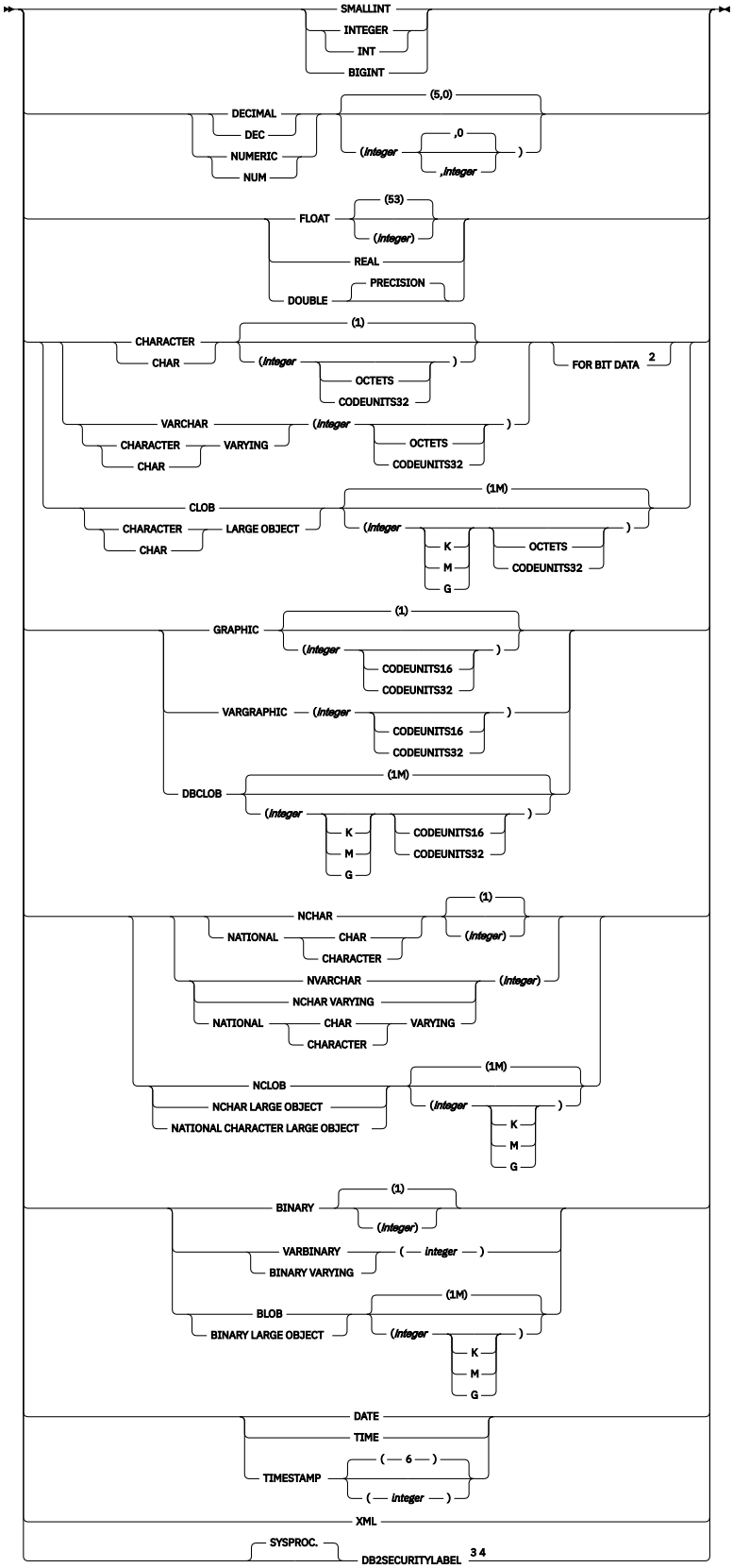
parameter-declaration



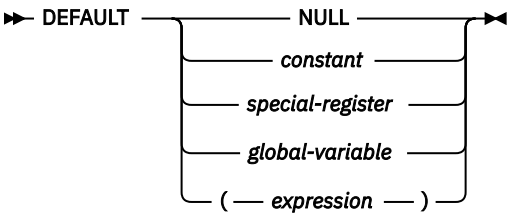
data-type1, data-type2, data-type3, data-type4



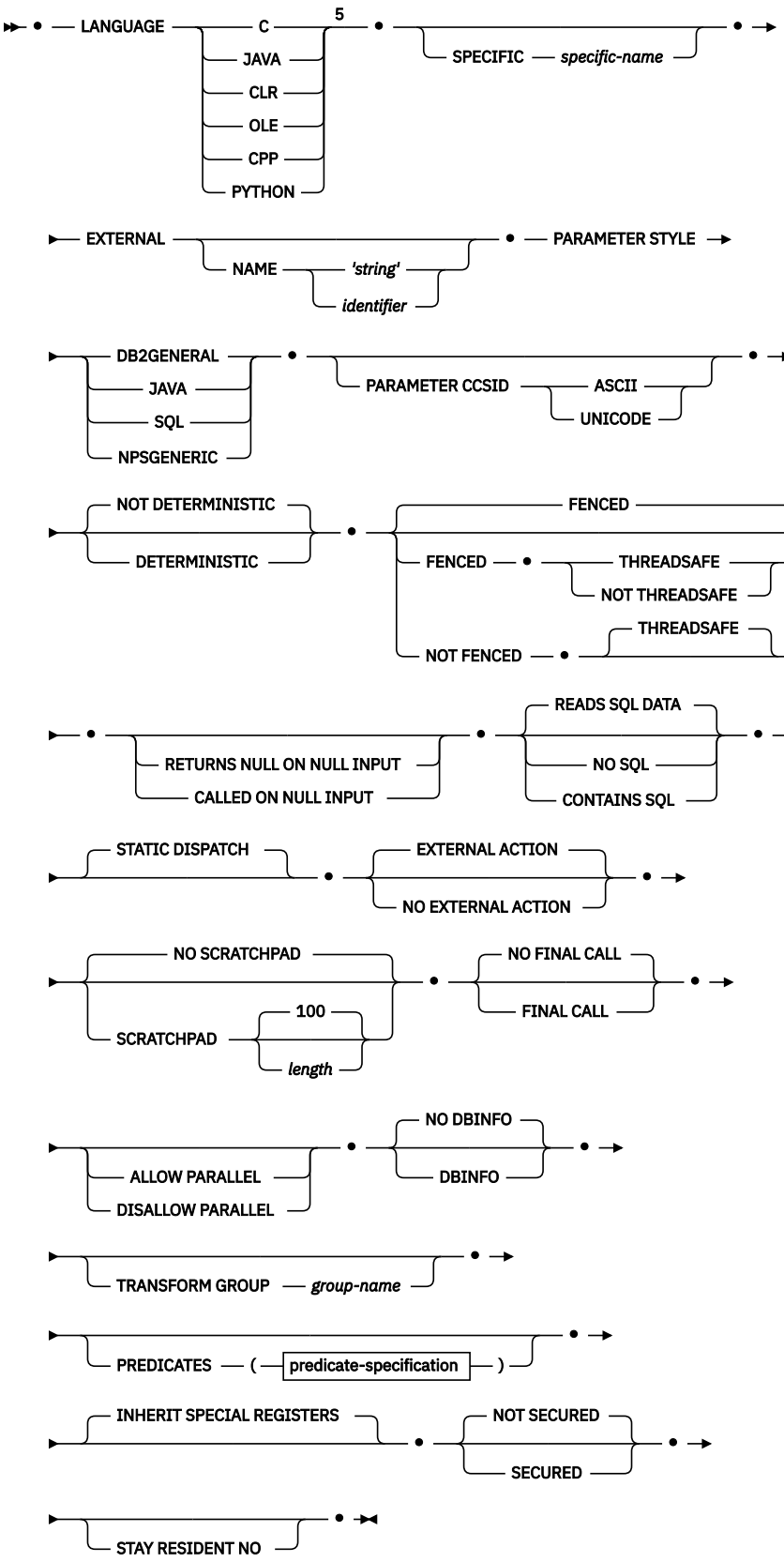
built-in-type



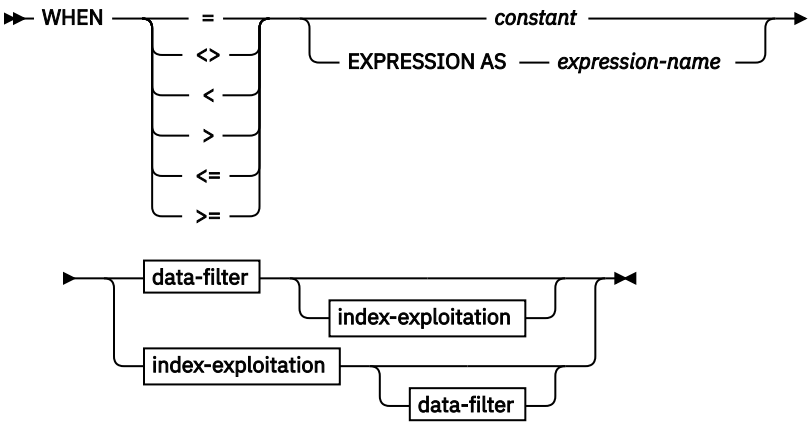
default-clause


DEFAULT
NULL
constant
special-register
global-variable
(
expression
)

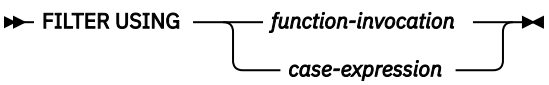
option-list



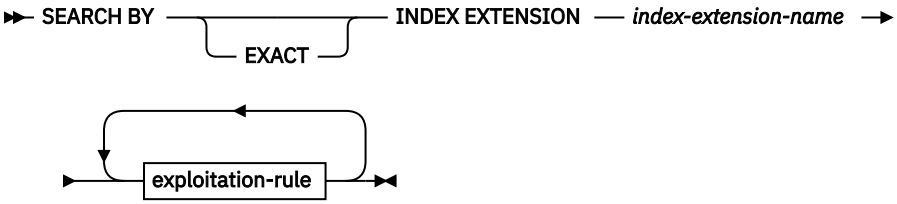
predicate-specification



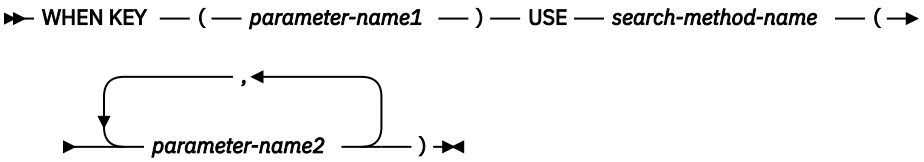
data-filter



index-exploitation



exploitation-rule



Notes:

[1] OUT and INOUT are valid only if the function has LANGUAGE C.

[2] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

[3] DB2SECURITYLABEL is the built-in distinct type that must be used to define the row security label column of a protected table.

[4] For a column of type DB2SECURITYLABEL, NOT NULL WITH DEFAULT is implicit and cannot be explicitly specified (SQLSTATE 42842). The default value for a column of type DB2SECURITYLABEL is the session authorization ID's security label for write access.

[5] LANGUAGE SQL is also supported.

Description

OR REPLACE

Specifies to replace the definition for the function if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog, with the exception that privileges that were granted on the function are not affected. This option can be specified only by the owner of the object. This option is ignored if a definition for the function does not exist at the current server. To replace an existing function, the specific name and function name of the new definition must be the same as the specific name and function name of the old definition, or the signature of the new definition must match the signature of the old definition. Otherwise, a new function is created.

If the function is referenced in the definition of a row permission or a column mask, the function cannot be replaced (SQLSTATE 42893).

function-name
Names the function being defined. It is a qualified or unqualified name that designates a function. The unqualified form of *function-name* is an SQL identifier. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a *schema-name* followed by a period and an SQL identifier. The qualified name must not be the same as the data type of the first parameter, if that first parameter is a structured type.

The name, including the implicit or explicit qualifiers, together with the number of parameters and the data type of each parameter (without regard for any length, precision or scale attributes of the data type) must not identify a function or method described in the catalog (SQLSTATE 42723). The unqualified name, together with the number and data types of the parameters, while of course unique within its schema, need not be unique across schemas.

If a two-part name is specified, the *schema-name* cannot begin with "SYS";. Otherwise, an error (SQLSTATE 42939) is raised.

A number of names used as keywords in predicates are reserved for system use, and cannot be used as a *function-name*. The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators. Failure to observe this rule will lead to an error (SQLSTATE 42939).

In general, the same name can be used for more than one function if there is some difference in the signature of the functions.

Although there is no prohibition against it, an external user-defined function should not be given the same name as a built-in function, unless it is an intentional override. To give a function having a different meaning the same name (for example, LENGTH, VALUE, MAX), with consistent arguments, as a built-in scalar or aggregate function, is to invite trouble for dynamic SQL statements, or when static SQL applications are rebound; the application may fail, or perhaps worse, may appear to run successfully while providing a different result.

(parameter-declaration,...)
Identifies the number of input parameters of the function, and specifies the mode, name, data type, and optional default value of each parameter. One entry in the list must be specified for each parameter that the function expects to receive. Up to 90 parameters can be specified (SQLSTATE 54023).

You can register a function that has no parameters; the parentheses must still be coded, with no intervening data types. For example:

```
CREATE FUNCTION WOOFER() ...
```

No two identically-named functions within a schema are permitted to have exactly the same type for all corresponding parameters. Lengths, precisions, and scales are not considered in this type comparison. Therefore, CHAR(8) and CHAR(35) are considered to be the same type, as are DECIMAL(11,2) and DECIMAL (4,3). A weakly typed distinct type specified for a parameter is considered to be the same data type as the source type of the distinct type. For a Unicode database, CHAR(13) and GRAPHIC(8) are considered to be the same type. There is some further bundling of types that causes them to be treated as the same type for this purpose, such as DECIMAL and NUMERIC. A duplicate signature returns an error (SQLSTATE 42723).

IN | OUT | INOUT
Specifies the mode of the parameter. If an error is returned by the function, OUT parameters are undefined and INOUT parameters are unchanged. The default is IN.

IN
Identifies the parameter as an input parameter to the function. Any changes made to the parameter within the function are not available to the invoking context when control is returned.

OUT
Identifies the parameter as an output parameter for the function.

The function must be defined with LANGUAGE C (SQLSTATE 42613).

The function can be referenced only on the right side of an assignment statement that is in a compound SQL (compiled) statement, and the function reference cannot be part of an expression (SQLSTATE 42887).

INOUT
Identifies the parameter as both an input and output parameter for the function.

The function must be defined with LANGUAGE C (SQLSTATE 42613).

The function can be referenced only on the right side of an assignment statement that is in a compound SQL (compiled) statement, and the function reference cannot be part of an expression (SQLSTATE 42887).

parameter-name
Specifies an optional name for the parameter. Parameter names are required to reference the parameters of a function in the *index-exploitation* clause of a predicate specification. The name cannot be the same as any other *parameter-name* in the parameter list (SQLSTATE 42734).

data-type1
Specifies the data type of the parameter. The data type can be a built-in data type, a distinct type, a structured type, or a reference type. For a more complete description of each built-in data type, see "CREATE TABLE". Some data types are not supported in all languages. For details on the mapping between SQL data types and host language data types, see "Data types that map to SQL data types in embedded SQL applications".

- A datetime type parameter is passed as a character data type, and the data is passed in the ISO format.
- DECIMAL (and NUMERIC) are invalid with LANGUAGE C and OLE (SQLSTATE 42815).
- DECFLOAT is invalid with LANGUAGE C, COBOL, CLR, JAVA, and OLE (SQLSTATE 42815).
- XML is invalid with LANGUAGE OLE.
- Because the XML value that is seen inside a function is a serialized version of the XML value that is passed as a parameter in the function call, parameters of type XML must be declared using the syntax `XML AS CLOB(`*n*`)`.
- CLR does not support DECIMAL scale greater than 28 (SQLSTATE 42613).
- Array types cannot be specified (SQLSTATE 42815).
- BINARY and VARBINARY data types are invalid with LANGUAGE CLR and OLE (SQLSTATE 42815).

For a user-defined distinct type, the length, precision, or scale attributes for the parameter are those of the source type of the distinct type (those specified on CREATE TYPE). A distinct type parameter is passed as the source type of the distinct type. If the name of the distinct type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

For a user-defined structured type, the appropriate transform functions must exist in the associated transform group.

For a reference type, the parameter can be specified as REF(*type-name*) if the parameter is unscoped.

DEFAULT
Specifies a default value for the parameter. The default can be a constant, a special register, a global variable, an expression, or the keyword NULL. The special registers that can be specified as

the default are that same as those that can be specified for a column default (see *default-clause* in the CREATE TABLE statement). Other special registers can be specified as the default by using an expression.

The expression can be any expression of the type described in "Expressions". If a default value is not specified, the parameter has no default and the corresponding argument cannot be omitted on invocation of the procedure. The maximum size of the *expression* is 64K bytes.

The default expression must not modify SQL data (SQLSTATE 428FL or SQLSTATE 429BL). The expression must be assignment compatible to the parameter data type (SQLSTATE 42821).

A default cannot be specified in the following situations:

- For INOUT or OUT parameters (SQLSTATE 42601)
- For a parameter of type ARRAY, ROW, or CURSOR (SQLSTATE 429BB)
- For a parameter to a function definition that also specified a PREDICATES clause (SQLSTATE 42613)

AS LOCATOR
Specifies that a locator to the value of the parameter is passed to the function instead of the actual value. Specify AS LOCATOR only for parameters with a LOB data type or a distinct type based on a LOB data type (SQLSTATE 42601). Passing locators instead of values can result in fewer bytes being passed to the function, especially when the value of the parameter is very large.

The AS LOCATOR clause has no effect on determining whether data types can be promoted, nor does it affect the function signature, which is used in function resolution.

If the function is FENCED and has the NO SQL option, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

RETURNS
This mandatory clause identifies the output of the function.

data-type2
Specifies the data type of the output.

In this case, exactly the same considerations apply as for the parameters of external functions described previously in *data-type1* for function parameters.

AS LOCATOR
For LOB types or distinct types which are based on LOB types, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed from the UDF instead of the actual value.

data-type3* CAST FROM *data-type4
Specifies the data type of the output.

This form of the RETURNS clause is used to return a different data type to the invoking statement from the data type that was returned by the function code. For example, in

```
CREATE FUNCTION GET_HIRE_DATE(CHAR(6))
  RETURNS DATE CAST FROM CHAR(10)
  ...
```

the function code returns a CHAR(10) value to the database manager, which, in turn, converts it to a DATE and passes that value to the invoking statement. The *data-type4* must be castable to the *data-type3* parameter. If it is not castable, an error (SQLSTATE 42880) is raised.

Since the length, precision or scale for *data-type3* can be inferred from *data-type4,* it not necessary (but still permitted) to specify the length, precision, or scale for parameterized types specified for *data-type3*. Instead empty parentheses may be used (for example VARCHAR() may be used). FLOAT() cannot be used (SQLSTATE 42601) since parameter value indicates different data types (REAL or DOUBLE).

Distinct types, array types, and structured types are not valid as the type specified in *data-type4* (SQLSTATE 42815).

The cast operation is also subject to runtime checks that might result in conversion errors being raised.

AS LOCATOR
For *data-type4* specifications that are LOB types or distinct types which are based on LOB types, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed back from the UDF instead of the actual value.

built-in-type
See "CREATE TABLE" for the description of built-in data types.

SPECIFIC *specific-name*
Provides a unique name for the instance of the function that is being defined. This specific name can be used when sourcing on this function, dropping the function, or commenting on the function. It can never be used to invoke the function. The unqualified form of *specific-name* is an SQL identifier. The qualified form is a *schema-name* followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another function instance or method specification that exists at the application server; otherwise an error (SQLSTATE 42710) is raised.

The *specific-name* may be the same as an existing *function-name*.

If no qualifier is specified, the qualifier that was used for *function-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *function-name* or an error (SQLSTATE 42882) is raised.

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmssxxx.

EXTERNAL
This clause indicates that the CREATE FUNCTION statement is being used to register a new function based on code written in an external programming language and adhering to the documented linkage conventions and interface.

If NAME clause is not specified "NAME *function-name*" is assumed.

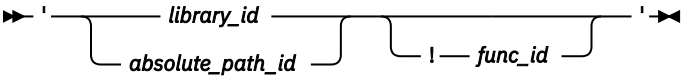
NAME *'string'*
This clause identifies the name of the user-written code which implements the function being defined.

The *'string'* option is a string constant with a maximum of 254 bytes. The format used for the string is dependent on the LANGUAGE specified.

- For LANGUAGE C:

  The *string* specified is the library name and function within the library, which the database manager invokes to execute the user-defined function being created. The library (and the function within the library) do not need to exist when the CREATE FUNCTION statement is executed. However, when the function is used in an SQL statement, the library and function within the library must exist and be accessible from the database server machine; otherwise, an error is returned (SQLSTATE 42724).

  The *string* can be specified as follows:



  Extraneous blanks are not permitted within the single quotation marks.

  library_id
  Identifies the library name containing the function. The database manager will look for the library as follows:

Operating system	Library name location
Linux AIX	If `myfunc` was given as the *library_id*, and the database manager is being run from `/u/production`, the database manager will look for the function in library `/u/production/sqllib/function/myfunc`
Windows	The database manager will look for the function in a directory path that is specified by the LIBPATH or PATH environment variable

absolute_path_id
Identifies the full path name of the file containing the function. The format depends on the operating system, as illustrated in the following table:

Operating system	Full path name example
Linux AIX	A value of '/u/jchui/mylib/myfunc' would cause the database manager to look in `/u/jchui/mylib` for the `myfunc` shared library.
Windows	A value of 'd:\mylib\myfunc.dll' would cause the database manager to load the dynamic link library, `myfunc.dll`, from the `d:\mylib` directory. If an absolute path ID is being used to identify the routine body, be sure to append the `.dll` extension.

! func_id
Identifies the entry point name of the function to be invoked. The ! serves as a delimiter between the library ID and the function ID. The format depends on the operating system, as illustrated in the following table:

Operating system	Entry point name of the function
Linux AIX	A value of 'mymod!func8' would direct the database manager to look for the library `$inst_home_dir/sqllib/function/mymod` and to use entry point `func8` within that library.
Windows	A value of 'mymod!func8' would direct the database manager to load the `mymod.dll` file and to call the `func8()` function in the dynamic link library (DLL).

If the string is not properly formed, an error is returned (SQLSTATE 42878).

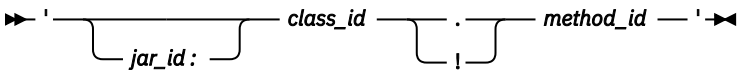
The body of every external function should be in a directory that is available on every database partition.

- For LANGUAGE JAVA:

  The *string* specified contains the optional jar file identifier, class identifier and method identifier, which the database manager invokes to execute the user-defined function being created. The class identifier and method identifier do not need to exist when the CREATE FUNCTION statement is executed. If a *jar_id* is specified, it must exist when the CREATE FUNCTION

statement is executed. However, when the function is used in an SQL statement, the method identifier must exist and be accessible from the database server machine; otherwise, an error is returned (SQLSTATE 42724).

The *string* can be specified as follows:



Extraneous blanks are not permitted within the single quotation marks.

jar_id
Identifies the jar identifier given to the jar collection when it was installed in the database. It can be either a simple identifier, or a schema qualified identifier. Examples are 'myJar' and 'mySchema.myJar'.

class_id
Identifies the class identifier of the Java object. If the class is part of a package, the class identifier part must include the complete package prefix, for example, 'myPacks.UserFuncs'. The directory the Java virtual machine will look in for the classes depends on the operating system, as illustrated in the following table:

Operating system	**Directory the Java virtual machine will look in for the classes**
Linux AIX	`'.../myPacks/UserFuncs/'`
Windows	`'...\myPacks\UserFuncs\'`

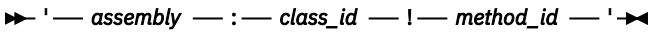
method_id
Identifies the method name of the Java object to be invoked.

- For LANGUAGE CLR:

The *string* specified represents the .NET assembly (library or executable), the class within that assembly, and the method within the class that the database manager invokes to execute the function being created. The module, class, and method do not need to exist when the CREATE FUNCTION statement is executed. However, when the function is used in an SQL statement, the module, class, and method must exist and be accessible from the database server machine; otherwise, an error is returned (SQLSTATE 42724).

C++ routines that are compiled with the '/clr' compiler option to indicate that they include managed code extensions must be cataloged as 'LANGUAGE CLR' and not 'LANGUAGE C'. The database server needs to know that the .NET infrastructure is being utilized in a user-defined function in order to make necessary runtime decisions. All user-defined functions using the .NET infrastructure must be cataloged as 'LANGUAGE CLR'.

The *string* can be specified as follows:



The name must be enclosed by single quotation marks. Extraneous blanks are not permitted.

assembly
Identifies the DLL or other assembly file in which the class resides. Any file extensions (such as .dll) must be specified. If the full path name is not given, the file must reside in the function directory of the database product's installation path

For example, `c:\sqllib\function`.

If the file resides in a subdirectory of the installation function directory, the subdirectory can be given before the file name rather than specifying the full path.

For example, if your install directory is `c:\sqllib` and your assembly file is `c:\sqllib\function\myprocs\mydotnet.dll`, it is only necessary to specify 'myprocs\mydotnet.dll' for the assembly.

The case sensitivity of this parameter is the same as the case sensitivity of the file system.

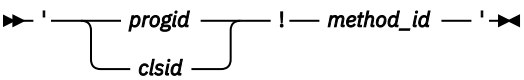
class_id
Specifies the name of the class within the given assembly in which the method that is to be invoked resides. If the class resides within a namespace, the full namespace must be given in addition to the class. For example, if the class EmployeeClass is in namespace MyCompany.ProcedureClasses, then MyCompany.ProcedureClasses.EmployeeClass must be specified for the class. Note that the compilers for some .NET languages will add the project name as a namespace for the class, and the behavior may differ depending on whether the command line compiler or the GUI compiler is used. This parameter is case sensitive.

method_id
Specifies the method within the given class that is to be invoked. This parameter is case sensitive.

- For LANGUAGE OLE:

The *string* specified is the OLE programmatic identifier (progid) or class identifier (clsid), and method identifier, which the database manager invokes to execute the user-defined function being created. The programmatic identifier or class identifier, and method identifier do not need to exist when the CREATE FUNCTION statement is executed. However, when the function is used in an SQL statement, the method identifier must exist and be accessible from the database server machine; otherwise, an error is returned (SQLSTATE 42724).

The *string* can be specified as follows:



Extraneous blanks are not permitted within the single quotation marks.

progid
Identifies the programmatic identifier of the OLE object.

progid is not interpreted by the database manager but only forwarded to the OLE APIs at run time. The specified OLE object must be creatable and support late binding (also called IDispatch-based binding).

clsid
Identifies the class identifier of the OLE object to create. It can be used as an alternative for specifying a *progid* in the case that an OLE object is not registered with a progid. The *clsid* has the form:

```
{nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn}
```

where 'n' is an alphanumeric character. *clsid* is not interpreted by the database manager but only forwarded to the OLE APIs at run time.

method_id
Identifies the method name of the OLE object to be invoked.

- For LANGUAGE CPP:

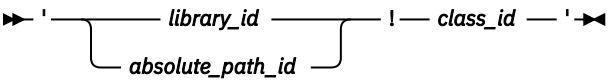
The *string* specified is the library identifier and class identifier within the library, which contains the evaluate method the database manager invokes to execute the user-defined function that is being created. If the string is not properly formed, an error is returned (SQLSTATE 42878).

It is not necessary that the library (or the class within the library) exist when the CREATE FUNCTION statement is executed. However, when the function is used in an SQL statement,

the library and class within the library must exist and be accessible from the database server machine; otherwise, an error is returned (SQLSTATE 42724).

The body of every external function should be in a directory that is available on every database partition.

The *string* can be specified as follows:



Extraneous blanks are not permitted within the single quotation marks.

library_id
The name of the library that contains the function:

- On a UNIX system, if the specified library ID is `myfunc`, and if the database manager is being run from `/u/production`, the database manager looks for the function in the following library:

```
/u/production/sqllib/function/myfunc
```

- On a Windows operating system, the database manager looks for the function in the directory path specified by the LIBPATH or PATH environment variable.

absolute_path_id
The full path of the file that contains the function. For example:

- On a UNIX system, the following specification causes the database manager to look in `/u/jchui/mylib` for the myfunc shared library:

```
'/u/jchui/mylib/myfunc'
```

- On a Windows operating system, the following specification causes the database manager to load the dynamic link library myfunc.dll from the `d:\mylib` directory:

```
'd:\mylib\myfunc.dll'
```

  If an absolute path ID is being used to identify the routine body, be sure to append the .dll extension.

class_id
The name of the class that contains the methods that are to be invoked.

For example, if you specify `'mymod!myclass'`:

- On a UNIX system, the database manager looks for the library `$inst_home_dir/sqllib/function/mymod` and invokes the evaluate method of the myclass class in that library.
- On a Windows operating system, the database manager loads the mymod.dll file and calls the evaluate method of the myclass class in the dynamic link library (DLL).

NAME *identifier*
This *identifier* specified is an SQL identifier. The SQL identifier is used as the *library-id* in the string. Unless it is a delimited identifier, the identifier is folded to upper case. If the identifier is qualified with a schema name, the schema name portion is ignored. This form of NAME can only be used with LANGUAGE C.

LANGUAGE
This mandatory clause specifies the language interface convention to which the body of the user-defined function is written.

C
The database manager calls the user-defined function as if it were a C function. The user-defined function must conform to the C language calling and linkage convention as defined by the standard ANSI C prototype.

JAVA
The database manager calls the user-defined function as a method in a Java class.

CLR
The database manager calls the user-defined function as a method in a .NET class. LANGUAGE CLR is supported only for user-defined functions running on Windows operating systems. NOT FENCED cannot be specified for a CLR routine (SQLSTATE 42601).

OLE
The database manager calls the user-defined function as if it were a method exposed by an OLE automation object. The user-defined function must conform with the OLE automation data types and invocation mechanism, as described in the *OLE Automation Programmer's Reference*.

LANGUAGE OLE is supported for user-defined functions for this database product only in Windows operating systems. THREADSAFE may not be specified for UDFs defined with LANGUAGE OLE (SQLSTATE 42613).

CPP
The database manager calls the user-defined function by invoking the evaluate method of a C++ class.

PYTHON
The database manager calls the user-defined function as a method in a Python class.

PARAMETER STYLE
This clause is used to specify the conventions used for passing parameters to and returning the value from functions.

DB2GENERAL
Used to specify the conventions for passing parameters to and returning the value from external functions that are defined as a method in a Java class. This can only specified when LANGUAGE JAVA is used.

The value DB2GENRL may be used as a synonym for DB2GENERAL.

JAVA
This means that the function will use a parameter passing convention that conforms to the Java language and SQLJ Routines specification. This can only be specified when LANGUAGE JAVA is used, no structured data types are specified as parameters, and no CLOB, BLOB, or DBCLOB data types are specified as return types (SQLSTATE 429B8). PARAMETER STYLE JAVA functions do not support the FINAL CALL, SCRATCHPAD, or DBINFO clause.

SQL
Used to specify the conventions for passing parameters to and returning the value from external functions that conform to C language calling and linkage conventions, methods exposed by OLE automation objects, or public static methods of a .NET object. This must be specified when LANGUAGE C, LANGUAGE CLR, or LANGUAGE OLE is used.

NPSGENERIC

Used to specify the conventions for passing parameters to and returning the value from external functions that are defined as a method in a C++ class. This can specified only when the LANGUAGE option is set to CPP or PYTHON.

When NPSGENERIC is specified as the parameter style, the UDF is written in C++ as a derived class of the nz.udx_ver2.Udf class. The class must implement the following two methods in addition to its constructor and destructor:

static Udf* Udf::instantiate(UdxInit *pInit)
Static member method instantiate(), which must instantiate a new instance of the UDF derived class and return a pointer to the new instance as a class Udf pointer. The engine uses this method to create an instance of a UDF object.

virtual ReturnValue Udf::evaluate()
Member method evaluate(), which is called by the engine to evaluate the user function and return a value to the caller.

PARAMETER CCSID

Specifies the encoding scheme to use for all string data passed into and out of the function. If the PARAMETER CCSID clause is not specified, the default is PARAMETER CCSID UNICODE for Unicode databases, and PARAMETER CCSID ASCII for all other databases.

ASCII

Specifies that string data is encoded in the database code page. If the database is a Unicode database, PARAMETER CCSID ASCII cannot be specified (SQLSTATE 56031). When the function is invoked, the application code page for the function is the database code page.

UNICODE

Specifies that string data is encoded in Unicode. If the database is a Unicode database, character data is in UTF-8, and graphic data is in UCS-2. If the database is not a Unicode database, character data is in UTF-8. In either case, when the function is invoked, the application code page for the function is 1208.

If the database is not a Unicode database, and a function with PARAMETER CCSID UNICODE is created, the function cannot have any graphic types, the XML type, or user-defined types (SQLSTATE 560C1).

If the database is not a Unicode database, and the alternate collating sequence has been specified in the database configuration, functions can be created with either PARAMETER CCSID ASCII or PARAMETER CCSID UNICODE. All string data passed into and out of the function will be converted to the appropriate code page.

This clause cannot be specified with LANGUAGE CPP, LANGUAGE OLE, LANGUAGE JAVA, or LANGUAGE CLR (SQLSTATE 42613).

DETERMINISTIC or NOT DETERMINISTIC

This optional clause specifies whether the function always returns the same results for given argument values (DETERMINISTIC) or whether the function depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC function must always return the same result from successive invocations with identical inputs. Optimizations taking advantage of the fact that identical inputs always produce the same results are prevented by specifying NOT DETERMINISTIC. An example of a NOT DETERMINISTIC function would be a random-number generator. An example of a DETERMINISTIC function would be a function that determines the square root of the input.

FENCED or NOT FENCED

This clause specifies whether or not the function is considered "safe" to run in the database manager operating environment's process or address space.

If a function is registered as FENCED, the database manager protects its internal resources (for example, data buffers) from access by the function. Most functions will have the option of running as FENCED or NOT FENCED. In general, a function running as FENCED will not perform as well as a similar one running as NOT FENCED.

CAUTION: Use of NOT FENCED for functions not adequately coded, reviewed, and tested can compromise the integrity of your database. This database product safeguards against many of the common types of inadvertent failures that might occur, but cannot guarantee complete integrity when NOT FENCED user-defined functions are used.

Only FENCED can be specified for a function with LANGUAGE OLE or NOT THREADSAFE (SQLSTATE 42613).

If the function is FENCED and has the NO SQL option, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

Either SYSADM authority, DBADM authority, or a special authority (CREATE_NOT_FENCED_ROUTINE) is required to register a user-defined function as NOT FENCED.

LANGUAGE CLR user-defined functions cannot be created when specifying the NOT FENCED clause (SQLSTATE 42601).

THREADSAFE or NOT THREADSAFE

Specifies whether the function is considered safe to run in the same process as other routines (THREADSAFE), or not (NOT THREADSAFE).

If the function is defined with LANGUAGE other than OLE:

- If the function is defined as THREADSAFE, the database manager can invoke the function in the same process as other routines. In general, to be threadsafe, a function should not use any global or static data areas. Most programming references include a discussion of writing threadsafe routines. Both FENCED and NOT FENCED functions can be THREADSAFE.
- If the function is defined as NOT THREADSAFE, the database manager will never simultaneously invoke the function in the same process as another routine.

For FENCED functions, THREADSAFE is the default if the LANGUAGE is JAVA or CLR. For all other languages, NOT THREADSAFE is the default. If the function is defined with LANGUAGE OLE, THREADSAFE may not be specified (SQLSTATE 42613).

For NOT FENCED functions, THREADSAFE is the default. NOT THREADSAFE cannot be specified (SQLSTATE 42613).

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT

This optional clause can be used to avoid a call to the external function if any of the arguments is null. If the user-defined function is defined to have no parameters, then this null argument condition cannot arise, and it does not matter how this specification is coded. If this clause is not specified, the default is RETURNS NULL ON NULL INPUT, except when PARAMETER STYLE JAVA is specified, in which case the default is CALLED ON NULL INPUT.

If RETURNS NULL ON NULL INPUT is specified, and if, at execution time, any one of the function's arguments is null, then the user-defined function is not called and the result is the null value.

If CALLED ON NULL INPUT is specified, then regardless of whether any arguments are null, the user-defined function is called. It can return a null value or a normal (non-null) value. But responsibility for testing for null argument values lies with the UDF.

The value NULL CALL may be used as a synonym for CALLED ON NULL INPUT for backwards and family compatibility. Similarly, NOT NULL CALL may be used as a synonym for RETURNS NULL ON NULL INPUT.

READS SQL DATA, NO SQL, or CONTAINS SQL

Specifies the classification of SQL statements that the function can run. The database manager verifies that the SQL statements that the function issues are consistent with this specification.

For the classification of each statement, see "SQL statements that can be executed in routines and triggers" in Developing User-defined Routines (SQL and External).

The default is READS SQL DATA.

READS SQL DATA

Specifies that the function can run statements with a data access classification of READS SQL DATA, CONTAINS SQL, or NO SQL (SQLSTATE 38002 or 42985). The function cannot run SQL statements that modify data. (SQLSTATE 38003 or 42985).

NO SQL

Specifies that the function can run only SQL statements with a data access classification of NO SQL (SQLSTATE 38001).

CONTAINS SQL

Specifies that the function can run only SQL statements with a data access classification of CONTAINS SQL or NO SQL (SQLSTATE 38004 or 42985). The function cannot run any SQL statements that read or modify data (SQLSTATE 38003 or 42985).

STATIC DISPATCH

This optional clause indicates that at function resolution time, a function is chosen by the database server based on the static types (declared types) of the parameters of the function.

EXTERNAL ACTION or NO EXTERNAL ACTION

Specifies whether the function takes an action that changes the state of an object that the database manager does not manage. An example of an external action is sending a message or writing a record to a file. The default is EXTERNAL ACTION.

EXTERNAL ACTION

Specifies that the function takes an action that changes the state of an object that the database manager does not manage.

A function with external actions might return incorrect results if the function is executed by parallel tasks. For example, if the function sends a note for each initial call to it, one note is sent for each parallel task instead of once for the function. Specify the DISALLOW PARALLEL clause for functions that do not work correctly with parallelism.

NO EXTERNAL ACTION

Specifies that the function does not take any action that changes the state of an object that the database manager does not manage. The database manager uses this information during optimization of SQL statements.

NO SCRATCHPAD or SCRATCHPAD *length*

This optional clause may be used to specify whether a scratchpad is to be provided for an external function. (It is strongly recommended that user-defined functions be re-entrant, so a scratchpad provides a means for the function to "save state" from one call to the next.)

- If SCRATCHPAD is specified, then at first invocation of the user-defined function, memory is allocated for a scratchpad to be used by the external function. On each invocation of the user-defined function, an additional argument is passed to the external function which addresses the scratchpad. This scratchpad has the following characteristics:
  - *length*, if specified, sets the size of the scratchpad in bytes; this value must be between 1 and 32 767 (SQLSTATE 42820). The default size is 100 bytes.
  - It is initialized to all X'00''s.
  - Its scope is the SQL statement. There is one scratchpad per reference to the external function in the SQL statement. So if the UDFX function in the following statement is defined with the SCRATCHPAD keyword, three scratchpads would be assigned.

    ```
    SELECT A, UDFX(A) FROM TABLEB
      WHERE UDFX(A) > 103 OR UDFX(A) < 19
    ```

    If ALLOW PARALLEL is specified or defaulted to, then the scope is different from the one shown previously. If the function is executed in multiple database partitions, a scratchpad would be assigned in each database partition where the function is processed, for each reference to the function in the SQL statement. Similarly, if the query is executed with intrapartition parallelism enabled, more than three scratchpads may be assigned.
  - It is persistent. Its content is preserved from one external function call to the next. Any changes made to the scratchpad by the external function on one call will be there on the next call. The database manager initializes scratchpads at the beginning of execution of each SQL statement. The database manager may reset scratchpads at the beginning of execution of each subquery. The system issues a final call before resetting a scratchpad if the FINAL CALL option is specified.
  - It can be used as a central point for system resources (for example, memory) which the external function might acquire. The function could acquire the memory on the first call, keep its address in the scratchpad, and refer to it in subsequent calls.

    (In such a case where system resource is acquired, the FINAL CALL keyword should also be specified; this causes a special call to be made at end-of-statement to allow the external function to free any system resources acquired.)
- If NO SCRATCHPAD is specified then no scratchpad is allocated or passed to the external function.

SCRATCHPAD cannot be specified in combination with a PARAMETER STYLE JAVA function.

FINAL CALL or NO FINAL CALL

This optional clause specifies whether a final call is to be made to an external function. The purpose of such a final call is to enable the external function to free any system resources it has acquired. It can be useful in conjunction with the SCRATCHPAD keyword in situations where the external function acquires system resources such as memory and anchors them in the scratchpad.

- If FINAL CALL is specified, then at execution time an additional argument is passed to the external function which specifies the type of call. The types of calls are:

  Normal call
  SQL arguments are passed and a result is expected to be returned.

  First call
  The first call to the external function for this reference to the user-defined function in this SQL statement. The first call is a normal call.

  Final call
  A final call to the external function to enable the function to free up resources. The final call is not a normal call. This final call occurs at the following times:

  End-of-statement
  This case occurs when the cursor is closed for cursor-oriented statements, or when the statement is through executing otherwise.

  End-of-parallel-task
  This case occurs when the function is executed by parallel tasks.

  End-of-transaction or interrupt
  This case occurs when the normal end-of-statement does not occur. For example, the logic of an application may for some reason bypass the close of the cursor. During this type of final call, no SQL statements may be issued except for CLOSE cursor (SQLSTATE 38505). This type of final call is indicated with a special value in the "call type" argument.

  If a commit operation occurs while a cursor defined as WITH HOLD is open, a final call is made at the subsequent close of the cursor or at the end of the application.
- If NO FINAL CALL is specified, no "call type" argument is passed to the external function, and no final call is made.

FINAL CALL cannot be specified in combination with the following parameter settings:

- PARAMETER STYLE JAVA
- LANGUAGE CPP

ALLOW PARALLEL or DISALLOW PARALLEL

This optional clause specifies whether, for a single reference to the function, the invocation of the function can be parallelized. In general, the invocations of most scalar functions should be parallelizable, but there may be functions (such as those depending on a single copy of a scratchpad) that cannot. If either ALLOW PARALLEL or DISALLOW PARALLEL are specified for a scalar function, then this specification is accepted. The following questions should be considered in determining which keyword is appropriate for the function.

- Are all the UDF invocations completely independent of each other? If YES, then specify ALLOW PARALLEL.
- Does each UDF invocation update the scratchpad, providing value(s) that are of interest to the next invocation? (For example, the incrementing of a counter.) If YES, then specify DISALLOW PARALLEL or accept the default.
- Is there some external action performed by the UDF which should happen only on one database partition? If YES, then specify DISALLOW PARALLEL or accept the default.
- Is the scratchpad used, but only so that some expensive initialization processing can be performed a minimal number of times? If YES, then specify ALLOW PARALLEL.
- Is the function going to be invoked in a query that accesses a column-organized table? If YES, then specifying ALLOW PARALLEL might improve performance.

In any case, the body of every external function should be in a directory that is available on every database partition.

The default value is ALLOW PARALLEL, except if one or more of the following options is specified in the statement.

- NOT DETERMINISTIC
- EXTERNAL ACTION
- SCRATCHPAD
- FINAL CALL

If any of these options is specified or implied, the default value is DISALLOW PARALLEL.

INHERIT SPECIAL REGISTERS
This optional clause specifies that updatable special registers in the function will inherit their initial values from the environment of the invoking statement. For a function invoked in the select-statement of a cursor, the initial values are inherited from the environment when the cursor is opened. For a routine invoked in a nested object (for example a trigger or view), the initial values are inherited from the runtime environment (not inherited from the object definition).

No changes to the special registers are passed back to the invoker of the function.

Non-updatable special registers, such as the datetime special registers, reflect a property of the statement currently executing, and are therefore set to their default values.

NO DBINFO or DBINFO
This optional clause specifies whether certain specific information known by the database server will be passed to the UDF as an additional invocation-time argument (DBINFO) or not (NO DBINFO). NO DBINFO is the default. DBINFO is not supported for the following clauses (SQLSTATE 42613):

- LANGUAGE OLE
- PARAMETER STYLE JAVA

If DBINFO is specified, then a structure is passed to the UDF which contains the following information:

- Data base name - the name of the currently connected database.
- Application ID - unique application ID which is established for each connection to the database.
- Application Authorization ID - the application runtime authorization ID, regardless of the nested UDFs in between this UDF and the application.
- Code page - identifies the database code page.
- Schema name - under the exact same conditions as for Table name, contains the name of the schema; otherwise blank.
- Table name - if and only if the UDF reference is either the right side of a SET clause in an UPDATE statement or an item in the VALUES list of an INSERT statement, contains the unqualified name of the table being updated or inserted; otherwise blank.
- Column name - under the exact same conditions as for Table name, contains the name of the column being updated or inserted; otherwise blank.
- Database version/release - identifies the version, release and modification level of the database server invoking the UDF.
- Platform - contains the server's platform type.
- Table function result column numbers - not applicable to external scalar functions.

TRANSFORM GROUP *group-name*
Indicates the transform group to be used for user-defined structured type transformations when invoking the function. A transform is required if the function definition includes a user-defined structured type as either a parameter or returns data type. If this clause is not specified, the default group name DB2_FUNCTION is used. If the specified (or default) *group-name* is not defined for a referenced structured type, an error is raised (SQLSTATE 42741). If a required FROM SQL or TO SQL

transform function is not defined for the given group-name and structured type, an error is raised (SQLSTATE 42744).

The transform functions, both FROM SQL and TO SQL, whether designated or implied, must be SQL functions which properly transform between the structured type and its built in type attributes.

PREDICATES

Defines the filtering or index extension exploitation performed when this function is used in a predicate. A predicate-specification allows the optional SELECTIVITY clause of a search-condition to be specified. If the PREDICATES clause is specified, the function must be defined as DETERMINISTIC with NO EXTERNAL ACTION (SQLSTATE 42613). If the PREDICATES clause is specified, and the database is not a Unicode database, PARAMETER CCSID UNICODE must not be specified (SQLSTATE 42613).

WHEN *comparison-operator*

Introduces a specific use of the function in a predicate with a comparison operator ("=", "<", ">", ">=", "<=", "<>").

constant

Specifies a constant value with a data type comparable to the RETURNS type of the function (SQLSTATE 42818). When a predicate uses this function with the same comparison operator and this constant, the specified filtering and index exploitation will be considered by the optimizer.

EXPRESSION AS *expression-name*

Provides a name for an expression. When a predicate uses this function with the same comparison operator and an expression, filtering and index exploitation may be used. The expression is assigned an expression name so that it can be used as a search function argument. The *expression-name* cannot be the same as any *parameter-name* of the function being created (SQLSTATE 42711). When an expression is specified, the type of the expression is identified.

FILTER USING

Allows specification of an external function or a case expression to be used for additional filtering of the result table.

function-invocation

Specifies a filter function that can be used to perform additional filtering of the result table. This is a version of the defined function (used in the predicate) that reduces the number of rows on which the user-defined predicate must be executed, to determine if rows qualify. If the results produced by the index are close to the results expected for the user-defined predicate, applying the filtering function may be redundant. If not specified, data filtering is not performed.

This function can use any *parameter-name*, the *expression-name*, or constants as arguments (SQLSTATE 42703), and returns an integer (SQLSTATE 428E4). A return value of 1 means the row is kept, otherwise it is discarded.

This function must also:

- Not be defined with LANGUAGE SQL (SQLSTATE 429B4)
- Not be defined with NOT DETERMINISTIC or EXTERNAL ACTION (SQLSTATE 42845)
- Not have a structured data type as the data type of any of the parameters (SQLSTATE 428E3)
- Not include a subquery (SQLSTATE 428E4)
- Not include an XMLQUERY or XMLEXISTS expression (SQLSTATE 428E4)

If an argument invokes another function or method, these rules are also enforced for this nested function or method. However, system-generated observer methods are allowed as arguments to the filter function (or any function or method used as an argument), as long as the argument evaluates to a built-in data type.

The definer of the function must have EXECUTE privilege on the specified filter function.

The *function-invocation* clause must not exceed 65 536 bytes in length in the database code page (SQLSTATE 22001).

case-expression
Specifies a case expression for additional filtering of the result table. The *searched-when-clause* and *simple-when-clause* can use *parameter-name*, *expression-name*, or a constant (SQLSTATE 42703). An external function with the rules specified in FILTER USING *function-invocation* may be used as a result-expression. Any function or method referenced in the *case-expression* must also conform to the four rules listed under *function-invocation*.

Subqueries and XMLQUERY or XMLEXISTS expressions cannot be used anywhere in the *case-expression* (SQLSTATE 428E4).

The case expression must return an integer (SQLSTATE 428E4). A return value of 1 in the result-expression means that the row is kept; otherwise it is discarded.

The *case-invocation* clause must not exceed 65 536 bytes in length in the database code page (SQLSTATE 22001).

index-exploitation
Defines a set of rules in terms of the search method of an index extension that can be used to exploit the index.

SEARCH BY INDEX EXTENSION *index-extension-name*
Identifies the index extension. The *index-extension-name* must identify an existing index extension.

EXACT
Indicates that the index lookup is exact in terms of the predicate evaluation. Use EXACT indicate that neither the original user-defined predicate function or the filter need to be applied after the index lookup. The EXACT predicate is useful when the index lookup returns the same results as the predicate.

If EXACT is not specified, then the original user-defined predicate is applied after index lookup. If the index is expected to provide only an approximation of the predicate, do not specify the EXACT option.

If the index lookup is not used, then the filter function and the original predicate have to be applied.

exploitation-rule
Describes the search targets and search arguments and how they can be used to perform the index search through a search method defined in the index extension.

WHEN KEY (*parameter-name1*)
This defines the search target. Only one search target can be specified for a key. The *parameter-name1* value identifies parameter names of the defined function (SQLSTATE 42703 or 428E8).

The data type of *parameter-name1* must match that of the source key specified in the index extension (SQLSTATE 428EY). The match must be exact for built-in and distinct data types and within the same structured type hierarchy for structured types.

This clause is true when the values of the named parameter are columns that are covered by an index based on the index extension specified.

USE *search-method-name*(*parameter-name2,...*)
This defines the search argument. It identifies which search method to use from those defined in the index extension. The *search-method-name* must match a search method defined in the index extension (SQLSTATE 42743). The *parameter-name2* values identify parameter names of the defined function or the *expression-name* in the EXPRESSION AS clause (SQLSTATE 42703). It must be different from any parameter name specified in the search target (SQLSTATE 428E9). The number of parameters and the data type of each *parameter-name2* must match the parameters defined for the search method in the index extension (SQLSTATE

42816). The match must be exact for built-in and distinct data types and within the same structured type hierarchy for structured types.

NOT SECURED or SECURED
Specifies whether the function is considered secure for row and column access control. The default is NOT SECURED.

NOT SECURED
Indicates that the function is not considered secure. When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled and column level access control is activated for its table (SQLSTATE 428HA). This rule applies to the non secure user-defined functions that are invoked anywhere in the statement.

SECURED
Indicates that the function is considered secure. The function must be secure when it is referenced in a row permission or a column mask (SQLSTATE 428H8).

STAY RESIDENT NO
Specifies that the library that is loaded for the function is not to remain resident in memory after the function ends. This clause is ignored when:

- The NOT FENCED clause is specified.
- The LANGUAGE option is set to JAVA or CLR.

Notes

- Determining whether one data type is castable to another data type does not consider length or precision and scale for parameterized data types such as CHAR and DECIMAL. Therefore, errors may occur when using a function as a result of attempting to cast a value of the source data type to a value of the target data type. For example, VARCHAR is castable to DATE but if the source type is actually defined as VARCHAR(5), an error will occur when using the function.
- When choosing the data types for the parameters of a user-defined function, consider the rules for promotion that will affect its input values (see "Promotion of data types"). For example, a constant which may be used as an input value could have a built-in data type different from the one expected and, more significantly, may not be promoted to the data type expected. Based on the rules for promotion, it is generally recommended to use the following data types for parameters:
  - INTEGER instead of SMALLINT
  - DOUBLE instead of REAL
  - VARCHAR instead of CHAR
  - VARGRAPHIC instead of GRAPHIC
- For portability of UDFs across platforms the following data types should not be used:
  - FLOAT- use DOUBLE or REAL instead.
  - NUMERIC- use DECIMAL instead.
  - LONG VARCHAR- use CLOB (or BLOB) instead.
- A function and a method may not be in an overriding relationship (SQLSTATE 42745). For more information about overriding, see "CREATE TYPE (Structured)".
- A function may not have the same signature as a method (comparing the first *parameter-type* of the function with the *subject-type* of the method) (SQLSTATE 42723).
- Creating a function with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- In a partitioned database environment, the use of SQL in external user-defined functions or methods is not supported (SQLSTATE 42997).
- Only routines defined as NO SQL can be used to define an index extension (SQLSTATE 428F8).

- If the function allows SQL, the external program must not attempt to access any federated objects (SQLSTATE 55047).
- A Java routine defined as NOT FENCED will be invoked as if it had been defined as FENCED THREADSAFE.
- XML parameters are only supported in LANGUAGE JAVA external functions when the PARAMETER STYLE DB2GENERAL clause is specified.
- ***Table access restrictions***

  If a function is defined as READS SQL DATA, no statement in the function can access a table that is being modified by the statement which invoked the function (SQLSTATE 57053). For example, suppose the user-defined function BONUS() is defined as READS SQL DATA. If the statement UPDATE EMPLOYEE SET SALARY = SALARY + BONUS(EMPNO) is invoked, no SQL statement in the BONUS function can read from the EMPLOYEE table.
- ***Setting of the default value***: Parameters of a function that are defined with a default value are set to their default value when the functions is invoked, but only if a value is not supplied for the corresponding argument, or is specified as DEFAULT, when the function is invoked.
- ***Privileges***: The definer of a function always receives the EXECUTE privilege WITH GRANT OPTION on the function, as well as the right to drop the function.

  When the function is used in an SQL statement, the function definer must have the EXECUTE privilege on any packages used by the function or EXECUTEIN privilege or DATAACCESS authority on the schema containing the packages.
- ***EXTERNAL ACTION functions***: If an EXTERNAL ACTION function is invoked in other than the outermost select list, the results are unpredictable since the number of times the function is invoked will vary depending on the access plan used.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of this database product and with other database products. These alternatives are non-standard and should not be used.
  - PARAMETER STYLE DB2SQL can be specified in place of PARAMETER STYLE SQL
  - NOT VARIANT can be specified in place of DETERMINISTIC, and VARIANT can be specified in place of NOT DETERMINISTIC
  - NULL CALL can be specified in place of CALLED ON NULL INPUT, and NOT NULL CALL can be specified in place of RETURNS NULL ON NULL INPUT

  The following syntax is accepted as the default behavior:
  - ASUTIME NO LIMIT
  - NO COLLID
  - PROGRAM TYPE SUB
  - STAY RESIDENT NO
  - CCSID UNICODE in a Unicode database
  - CCSID ASCII in a non-Unicode database if PARAMETER CCSID UNICODE is not specified
- ***Creating a secure function***: Normally users with SECADM authority do not have privileges to create database objects such as triggers and functions. Typically; they will examine the data accessed by the function, ensure it is secure, then grant the CREATE_SECURE_OBJECT authority to someone who currently has required privileges to create a secure user-defined function. After the function is created, they will revoke the CREATE_SECURE_OBJECT authority from the function owner.

  The SECURED attribute is considered to be an assertion that declares the user has established a change control audit procedure for all changes to the user-defined function. The database manager assumes that such a control audit procedure is in place for all subsequent ALTER FUNCTION statements or changes to external packages.
- ***Invoking other user-defined functions in a secure function***: If a secure user-defined function invokes other user-defined functions, the database manager does not validate whether those nested user-

defined functions have the SECURED attribute. If those nested functions can access sensitive data, the user with SECADM authority needs to ensure those functions are allowed to access those data and a change control audit procedure has been established for all changes to those functions.

- ***Replacing an existing function such that the secure attribute is changed (from SECURED to NOT SECURED and vice versa)***: Packages and dynamically cached SQL statements that depend on the function may be invalidated because the secure attribute affects the access path selection for statements involving tables for which row or column level access control is activated.

Examples

1. Pellow is registering the CENTER function in his PELLOW schema. Let those keywords that will default do so, and let the system provide a function specific name:

```
CREATE FUNCTION CENTER (INT,FLOAT)
  RETURNS FLOAT
  EXTERNAL NAME 'mod!middle'
  LANGUAGE C
  PARAMETER STYLE SQL
  DETERMINISTIC
  NO SQL
  NO EXTERNAL ACTION
```

2. Now, McBride (who has DBADM authority) is registering another CENTER function in the PELLOW schema, giving it an explicit specific name for subsequent data definition language use, and explicitly providing all keyword values. Note also that this function uses a scratchpad and presumably is accumulating data there that affects subsequent results. Since DISALLOW PARALLEL is specified, any reference to the function is not parallelized and therefore a single scratchpad is used to perform some one-time only initialization and save the results.

```
CREATE FUNCTION PELLOW.CENTER (FLOAT, FLOAT, FLOAT)
  RETURNS DECIMAL(8,4) CAST FROM FLOAT
  SPECIFIC FOCUS92
  EXTERNAL NAME 'effects!focalpt'
  LANGUAGE C   PARAMETER STYLE SQL
  DETERMINISTIC   FENCED   NOT NULL CALL   NO SQL   NO EXTERNAL ACTION
  SCRATCHPAD   NO FINAL CALL
  DISALLOW PARALLEL
```

3. The following example is the C language user-defined function program written to implement the rule `output = 2 * input - 4` returning NULL if and only if the input is null. It could be written even more simply (that is, without null checking), if the CREATE FUNCTION statement had used NOT NULL CALL. The CREATE FUNCTION statement:

```
CREATE FUNCTION ntest1 (SMALLINT)
  RETURNS SMALLINT
  EXTERNAL NAME 'ntest1!nudft1'
  LANGUAGE C    PARAMETER STYLE SQL
  DETERMINISTIC   NOT FENCED   NULL CALL
  NO SQL   NO EXTERNAL ACTION
```

   The program code:

```
#include "sqlsystm.h"
/* NUDFT1 IS A USER_DEFINED SCALAR FUNCTION */
/* udft1 accepts smallint input
and produces smallint output
implementing the rule:
if (input is null)
set output = null;
else
set output = 2 * input - 4;
*/
void SQL_API_FN nudft1
(short *input,      /* ptr to input arg */
short *output,     /* ptr to where result goes */
short *input_ind,  /* ptr to input indicator var */
short *output_ind, /* ptr to output indicator var */
char sqlstate[6],  /* sqlstate, allows for null-term */
char fname[28],    /* fully qual func name, nul-term */
char finst[19],    /* func specific name,  null-term */
```

```
char msgtext[71])  /* msg text buffer,     null-term */
{
/* first test for null input */
if (*input_ind == -1)
{
/* input is null, likewise output */
*output_ind = -1;
}
else
{
/* input is not null.  set output to 2*input-4 */
*output = 2 * (*input) - 4;
/* and set out null indicator to zero */
*output_ind = 0;
}
/* signal successful completion by leaving sqlstate as is */
/* and exit */
return;
}
/* end of UDF: NUDFT1 */
```

4. The following example registers a Java UDF which returns the position of the first vowel in a string. The UDF is written in Java, is to be run fenced, and is the findvwl method of class javaUDFs.

```
CREATE FUNCTION findv ( CLOB(100K))
  RETURNS INTEGER
  FENCED
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  EXTERNAL NAME 'javaUDFs.findvwl'
  NO EXTERNAL ACTION
  CALLED ON NULL INPUT
  DETERMINISTIC
  NO SQL
```

5. This example outlines a user-defined predicate WITHIN that takes two parameters, g1 and g2, of type SHAPE as input:

```
CREATE FUNCTION within (g1 SHAPE, g2 SHAPE)
RETURNS INTEGER
LANGUAGE C
PARAMETER STYLE SQL
DETERMINISTIC
NOT FENCED
NO SQL
NO EXTERNAL ACTION
EXTERNAL NAME 'db2sefn!SDESpatilRelations'
PREDICATES
WHEN = 1
FILTER USING mbrOverlap(g1..xmin, g1..ymin, g1..xmax, g1..max,
g2..xmin, g2..ymin, g2..xmax, g2..ymax)
SEARCH BY INDEX EXTENSION gridIndex
WHEN KEY(g1) USE withinExplRule(g2)
WHEN KEY(g2) USE withinExplRule(g1)
```

The description of the WITHIN function is similar to that of any user-defined function, but the following additions indicate that this function can be used in a user-defined predicate.

- **PREDICATES WHEN = 1** indicates that when this function appears as

  ```
  within(g1, g2) = 1
  ```

  in the WHERE clause of a DML statement, the predicate is to be treated as a user-defined predicate and the index defined by the index extension *gridIndex* should be used to retrieve rows that satisfy this predicate. If a constant is specified, the constant specified during the DML statement has to match exactly the constant specified in the create index statement. This condition is provided mainly to cover Boolean expression where the result type is either a 1 or a 0. For other cases, the EXPRESSION clause is a better choice.
- **FILTER USING mbrOverlap** refers to a filtering function mbrOverlap, which is a cheaper version of the WITHIN predicate. In this example, the mbrOverlap function takes the minimum bounding rectangles as input and quickly determines if they overlap or not. If the minimum bounding

rectangles of the two input shapes do not overlap, then g1 will not be contained with g2. Therefore the tuple can be safely discarded, avoiding the application of the expensive WITHIN predicate.

- The **SEARCH BY INDEX EXTENSION** clause indicates that combinations of index extension and search target can be used for this user-defined predicate.

6. This example outlines a user-defined predicate DISTANCE that takes two parameters, P1 and P2, of type POINT as input:

```
CREATE FUNCTION distance (P1 POINT, P2 POINT)
  RETURNS INTEGER
  LANGUAGE C
  PARAMETER STYLE SQL
  DETERMINISTIC
  NOT FENCED
  NO SQL
  NO EXTERNAL ACTION
  EXTERNAL NAME 'db2sefn!SDEDistances'
  PREDICATES
  WHEN > EXPRESSION AS distExpr
  SEARCH BY INDEX EXTENSION gridIndex
  WHEN KEY(P1) USE distanceGrRule(P2, distExpr)
  WHEN KEY(P2) USE distanceGrRule(P1, distExpr)
```

The description of the DISTANCE function is similar to that of any user-defined function, but the following additions indicate that when this function is used in a predicate, that predicate is a user-defined predicate.

- **PREDICATES WHEN > EXPRESSION AS distExpr** is another valid predicate specification. When an expression is specified in the WHEN clause, the result type of that expression is used for determining if the predicate is a user-defined predicate in the DML statement. For example:

```
SELECT T1.C1
  FROM T1, T2
  WHERE distance (T1.P1, T2.P1) > T2.C2
```

The predicate specification distance takes two parameters as input and compares the results with T2.C2, which is of type INTEGER. Since only the data type of the right side expression matters, (as opposed to using a specific constant), it is better to choose the EXPRESSION clause in the CREATE FUNCTION DDL for specifying a wildcard as the comparison value.

Alternatively, the following statement is also a valid user-defined predicate:

```
SELECT T1.C1
  FROM T1, T2
  WHERE distance(T1.P1, T2.P1) > distance (T1.P2, T2.P2)
```

There is currently a restriction that only the right side is treated as the expression; the term on the left side is the user-defined function for the user-defined predicate.

- The **SEARCH BY INDEX EXTENSION** clause indicates that combinations of index extension and search target can be used for this user-defined-predicate. In the case of the distance function, the expression identified as distExpr is also one of the search arguments that is passed to the range-producer function (defined as part of the index extension). The expression identifier is used to define a name for the expression so that it is passed to the range-producer function as an argument.

CREATE FUNCTION (external table)

The CREATE FUNCTION (External Table) statement is used to register a user-defined external table function at the current server.

A *table function* can be used in the FROM clause of a SELECT, and returns a table to the SELECT by returning one row at a time.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- CREATE_EXTERNAL_ROUTINE authority on the database and at least one of the following authorities:
  - IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the function does not exist
  - CREATEIN privilege on the schema, if the schema name of the function exists
  - SCHEMAADM authority on the schema, if the schema name of the function exists
- DBADM authority

Group privileges are not considered for any table or view specified in the CREATE FUNCTION statement.

To create a not-fenced function, the privileges held by the authorization ID of the statement must also include at least one of the following authorities:

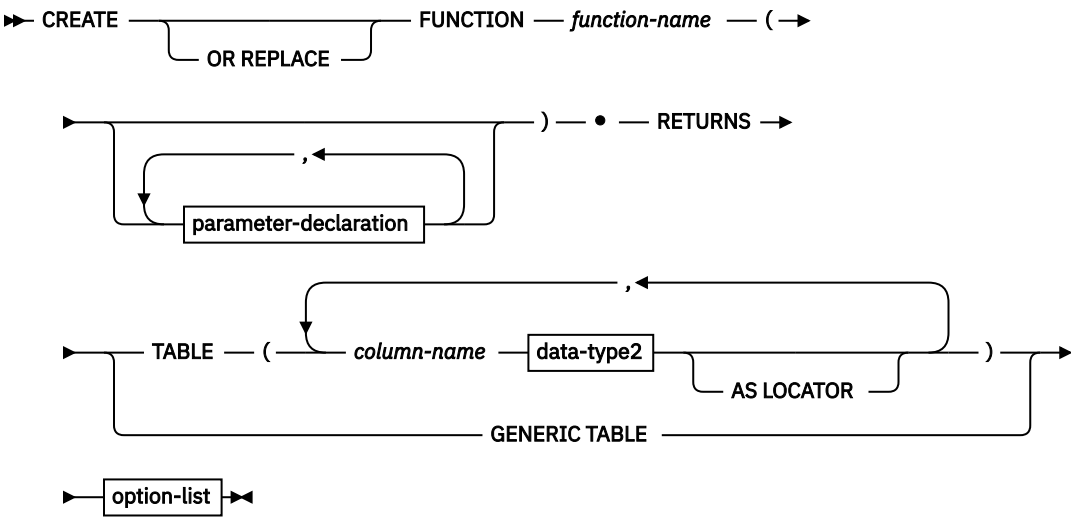
- CREATE_NOT_FENCED_ROUTINE authority on the database
- DBADM authority

To create a fenced function, no additional authorities or privileges are required.

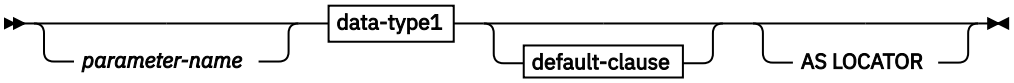
To replace an existing function, the authorization ID of the statement must be the owner of the existing function (SQLSTATE 42501).

If the SECURED option is specified, the authorization ID of the statement must include SECADM or CREATE_SECURE_OBJECT authority (SQLSTATE 42501).

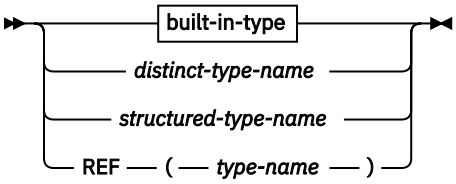
Syntax



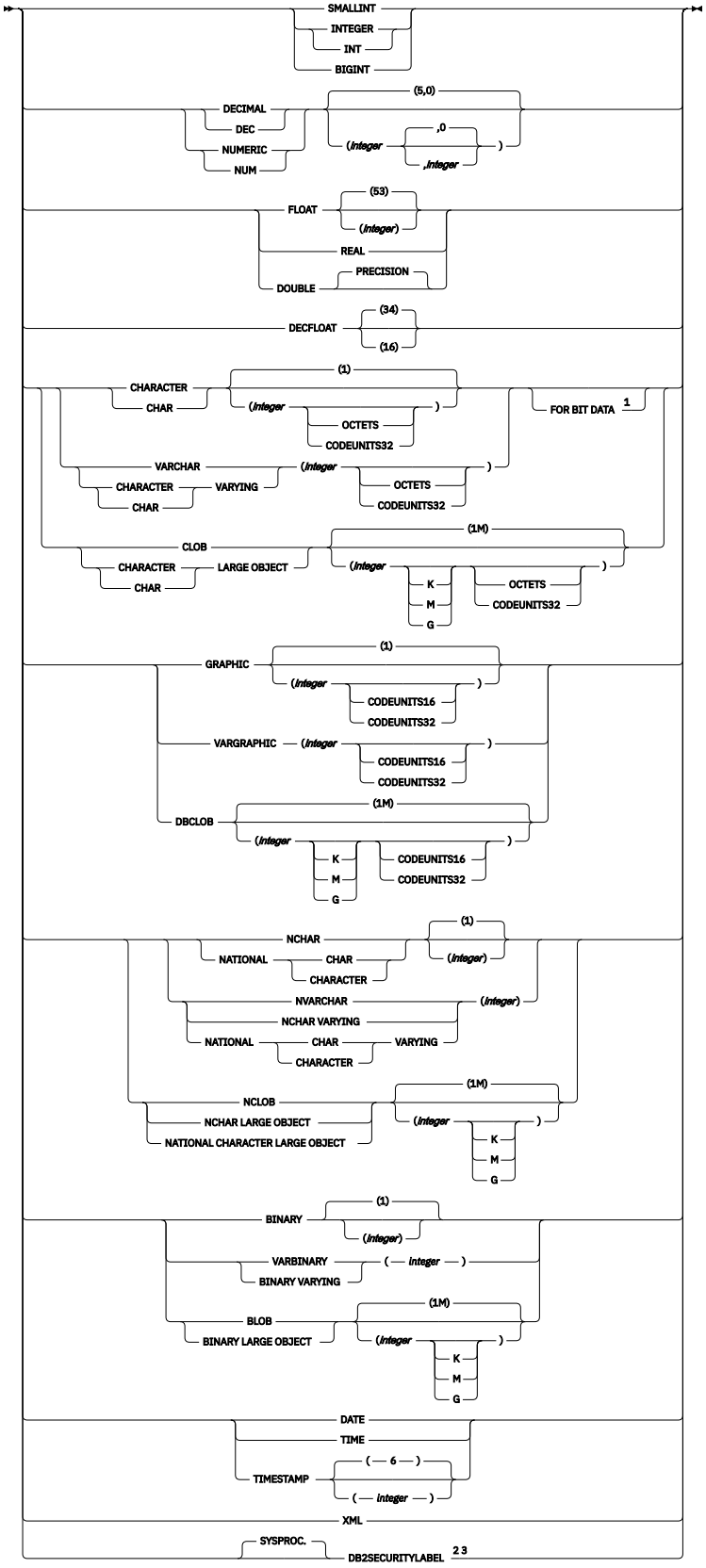
parameter-declaration



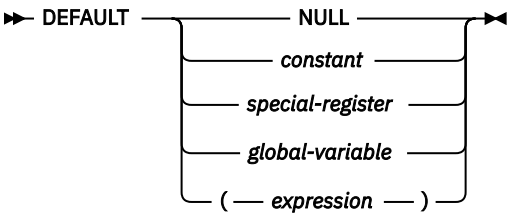
data-type1, data-type2



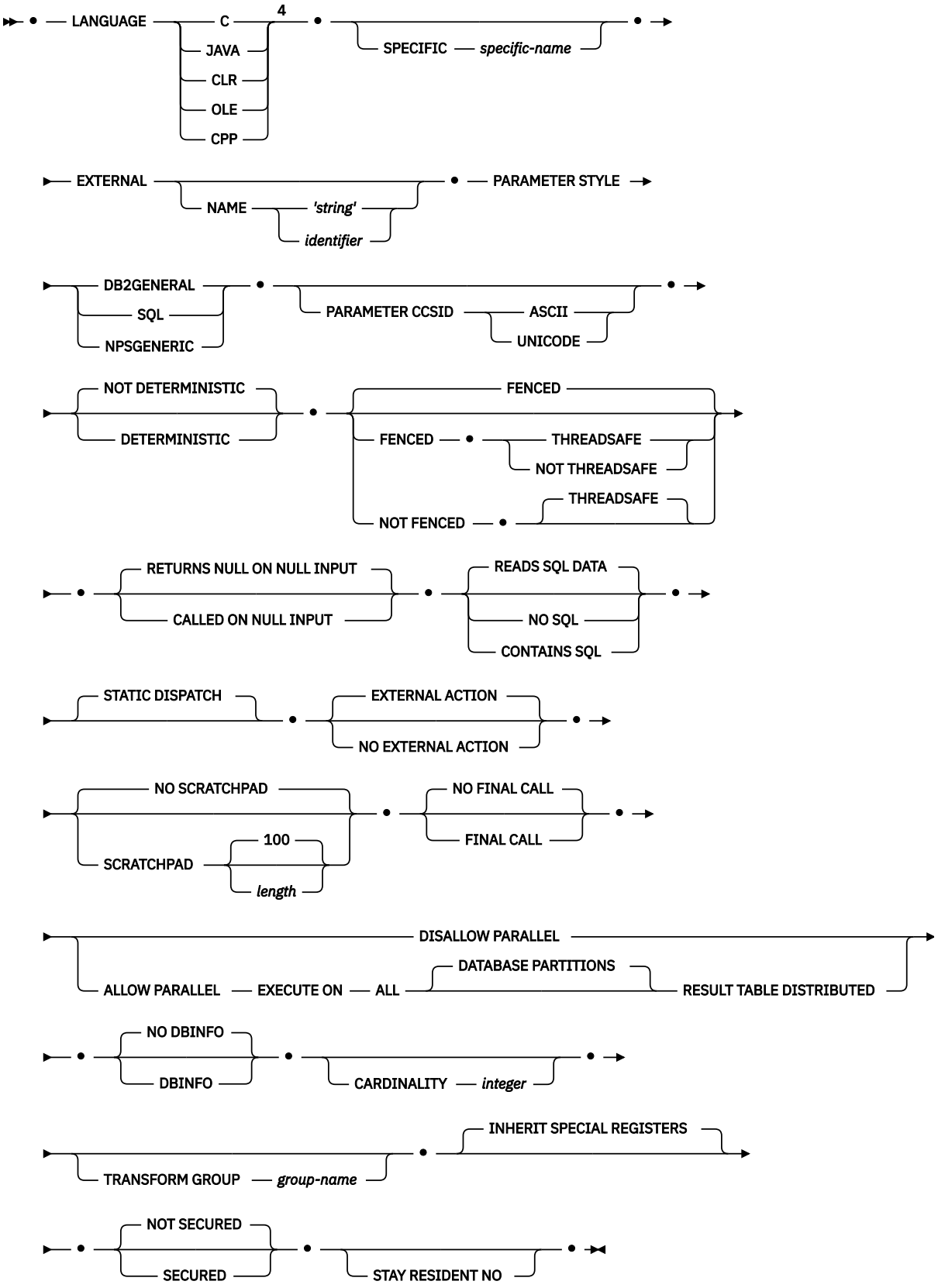
built-in-type



default-clause


DEFAULT
NULL
constant
special-register
global-variable
(
expression
)

option-list



Notes:

[1] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

[2] DB2SECURITYLABEL is the built-in distinct type that must be used to define the row security label column of a protected table.

[3] For a column of type DB2SECURITYLABEL, NOT NULL WITH DEFAULT is implicit and cannot be explicitly specified (SQLSTATE 42842). The default value for a column of type DB2SECURITYLABEL is the session authorization ID's security label for write access.

[4] For information about creating LANGUAGE OLE DB external table functions, see "CREATE FUNCTION (OLE DB External Table)". For information about creating LANGUAGE SQL table functions, see "CREATE FUNCTION (SQL Scalar, Table, or Row)".

Description

OR REPLACE

Specifies to replace the definition for the function if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog, with the exception that privileges that were granted on the function are not affected. This option can be specified only by the owner of the object. This option is ignored if a definition for the function does not exist at the current server. To replace an existing function, the specific name and function name of the new definition must be the same as the specific name and function name of the old definition, or the signature of the new definition must match the signature of the old definition. Otherwise, a new function is created.

If the function is referenced in the definition of a row permission or a column mask, the function cannot be replaced (SQLSTATE 42893).

function-name

Names the function being defined. It is a qualified or unqualified name that designates a function. The unqualified form of *function-name* is an SQL identifier. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a *schema-name* followed by a period and an SQL identifier. The qualified name must not be the same as the data type of the first parameter, if that first parameter is a structured type.

The name, including the implicit or explicit qualifiers, together with the number of parameters and the data type of each parameter (without regard for any length, precision or scale attributes of the data type) must not identify a function described in the catalog (SQLSTATE 42723). The unqualified name, together with the number and data types of the parameters, while of course unique within its schema, need not be unique across schemas.

If a two-part name is specified, the *schema-name* cannot begin with "SYS" (SQLSTATE 42939).

A number of names used as keywords in predicates are reserved for system use, and cannot be used as a *function-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators.

The same name can be used for more than one function if there is some difference in the signature of the functions. Although there is no prohibition against it, an external user-defined table function should not be given the same name as a built-in function.

(*parameter-declaration*,...)

Identifies the number of input parameters of the function, and specifies the data type and optional default value of each parameter. One entry in the list must be specified for each parameter that the function will expect to receive. No more than 90 parameters are allowed (SQLSTATE 54023).

It is possible to register a function that has no parameters. In this case, the parentheses must still be coded, with no intervening data types. For example:

```
CREATE FUNCTION WOOFER() ...
```

No two identically-named functions within a schema are permitted to have exactly the same type for all corresponding parameters. Lengths, precisions, and scales are not considered in this type comparison. Therefore, CHAR(8) and CHAR(35) are considered to be the same type, as are

DECIMAL(11,2) and DECIMAL (4,3). A weakly typed distinct type specified for a parameter is considered to be the same data type as the source type of the distinct type. For a Unicode database, CHAR(13) and GRAPHIC(8) are considered to be the same type. There is some further bundling of types that causes them to be treated as the same type for this purpose, such as DECIMAL and NUMERIC. A duplicate signature returns an error (SQLSTATE 42723).

parameter-name
Specifies an optional name for the input parameter. The name cannot be the same as any other *parameter-name* in the parameter list (SQLSTATE 42734).

data-type1
Specifies the data type of the input parameter. The data type can be a built-in data type, a distinct type, a structured type, or a reference type. For a more complete description of each built-in data type, see "CREATE TABLE". Some data types are not supported in all languages. For details on the mapping between SQL data types and host language data types, see "Data types that map to SQL data types in embedded SQL applications".

- A datetime type parameter is passed as a character data type, and the data is passed in the ISO format.
- DECIMAL (and NUMERIC) are invalid with LANGUAGE C and OLE (SQLSTATE 42815).
- XML is invalid with LANGUAGE OLE.
- Because the XML value that is seen inside a function is a serialized version of the XML value that is passed as a parameter in the function call, parameters of type XML must be declared using the syntax `XML AS CLOB(`*n*`)`.
- CLR does not support DECIMAL scale greater than 28 (SQLSTATE 42613).
- Array types cannot be specified (SQLSTATE 42815).
- BINARY and VARBINARY data types are invalid with LANGUAGE CLR and OLE (SQLSTATE 42815).

For a user-defined distinct type, the length, precision, or scale attributes for the parameter are those of the source type of the distinct type (those specified on CREATE TYPE). A distinct type parameter is passed as the source type of the distinct type. If the name of the distinct type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

For a user-defined structured type, the appropriate transform functions must exist in the associated transform group.

For a reference type, the parameter can be specified as REF(*type-name*) if the parameter is unscoped.

DEFAULT
Specifies a default value for the parameter. The default can be a constant, a special register, a global variable, an expression, or the keyword NULL. The special registers that can be specified as the default are that same as those that can be specified for a column default (see *default-clause* in the CREATE TABLE statement). Other special registers can be specified as the default by using an expression.

The expression can be any expression of the type described in "Expressions". If a default value is not specified, the parameter has no default and the corresponding argument cannot be omitted on invocation of the procedure. The maximum size of the *expression* is 64K bytes.

The default expression must not modify SQL data (SQLSTATE 428FL or SQLSTATE 429BL). The expression must be assignment compatible to the parameter data type (SQLSTATE 42821).

A default cannot be specified for a parameter of type ARRAY, ROW, or CURSOR (SQLSTATE 429BB).

AS LOCATOR
Specifies that a locator to the value of the parameter is passed to the function instead of the actual value. Specify AS LOCATOR only for parameters with a LOB data type or a distinct type

based on a LOB data type (SQLSTATE 42601). Passing locators instead of values can result in fewer bytes being passed to the function, especially when the value of the parameter is very large.

The AS LOCATOR clause has no effect on determining whether data types can be promoted, nor does it affect the function signature, which is used in function resolution.

If the function is FENCED and has the NO SQL option, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

RETURNS
Specifies the output of the function.

TABLE
Specifies that the output of the function is a table. The parentheses that follow this keyword delimit a list of the names and types of the columns of the table. The list style resembles the style of a simple CREATE TABLE statement which has no additional specifications (constraints, for example). No more than 255 columns are allowed (SQLSTATE 54011).

column-name
Specifies the name of this column. The name cannot be qualified and the same name cannot be used for more than one column of the table.

data-type2
Specifies the data type of the column, and can be any data type supported for a parameter of a UDF written in the particular language, except for structured types (SQLSTATE 42997).

AS LOCATOR
When *data-type2* is a LOB type or distinct type based on a LOB type, the use of this option indicates that the function is returning a locator for the LOB value that is instantiated in the result table.

The valid types for use with this clause are discussed in the "CREATE FUNCTION (external scalar)" statement topic.

GENERIC TABLE
Specifies that the output of the function is a generic table. This clause is allowed only if you specify the LANGUAGE JAVA clause and the PARAMETER STYLE DB2GENERAL clause (SQLSTATE 42613).

built-in-type
See "CREATE TABLE" for the description of built-in data types.

SPECIFIC *specific-name*
Provides a unique name for the instance of the function that is being defined. This specific name can be used when sourcing on this function, dropping the function, or commenting on the function. It can never be used to invoke the function. The unqualified form of *specific-name* is an SQL identifier. The qualified form is a *schema-name* followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another function instance that exists at the application server; otherwise an error (SQLSTATE 42710) is raised.

The *specific-name* may be the same as an existing *function-name*.

If no qualifier is specified, the qualifier that was used for *function-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *function-name* or an error (SQLSTATE 42882) is raised.

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmssxxx.

EXTERNAL
This clause indicates that the CREATE FUNCTION statement is being used to register a new function based on code written in an external programming language and adhering to the documented linkage conventions and interface.

If NAME clause is not specified "NAME *function-name*" is assumed.

NAME *'string'*

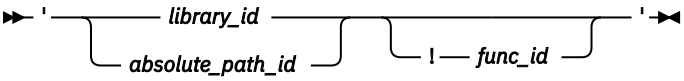
This clause identifies the user-written code that implements the function being defined.

The *'string'* option is a string constant with a maximum of 254 bytes. The format used for the string is dependent on the LANGUAGE specified.

- For LANGUAGE C:

  The *string* specified is the library name and function within the library, which the database manager invokes to execute the user-defined function being created. The library (and the function within the library) do not need to exist when the CREATE FUNCTION statement is executed. However, when the function is used in an SQL statement, the library and function within the library must exist and be accessible from the database server machine.

  The *string* can be specified as follows:



  Extraneous blanks are not permitted within the single quotation marks.

  library_id

  Identifies the library name containing the function. The database manager will look for the library as follows:

Operating system	Library name location
Linux AIX	If `myfunc` was given as the *library_id*, and the database manager is being run from `/u/production`, the database manager will look for the function in library `/u/production/sqllib/function/myfunc`
Windows	The database manager will look for the function in a directory path that is specified by the LIBPATH or PATH environment variable

  absolute_path_id

  Identifies the full path name of the file containing the function. The format depends on the operating system, as illustrated in the following table:

Operating system	Full path name example
Linux AIX	A value of '/u/jchui/mylib/myfunc' would cause the database manager to look in `/u/jchui/mylib` for the `myfunc` shared library.
Windows	A value of 'd:\mylib\myfunc.dll' would cause the database manager to load the dynamic link library, `myfunc.dll`, from the `d:\mylib` directory. If an absolute path ID is being used to identify the routine body, be sure to append the `.dll` extension.

  ! func_id

  Identifies the entry point name of the function to be invoked. The ! serves as a delimiter between the library ID and the function ID. The format depends on the operating system, as illustrated in the following table:

Operating system	Entry point name of the function
Linux AIX	A value of 'mymod!func8' would direct the database manager to look for the library `$inst_home_dir/sqllib/function/mymod` and to use entry point `func8` within that library.
Windows	A value of 'mymod!func8' would direct the database manager to load the `mymod.dll` file and to call the `func8()` function in the dynamic link library (DLL).

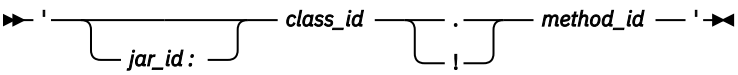
If the string is not properly formed, an error is returned (SQLSTATE 42878).

In any case, the body of every external function should be in a directory that is available on every database partition.

- For LANGUAGE JAVA:

  The *string* specified contains the optional jar file identifier, class identifier and method identifier, which the database manager invokes to execute the user-defined function being created. The class identifier and method identifier do not need to exist when the CREATE FUNCTION statement is executed. If a *jar_id* is specified, it must exist when the CREATE FUNCTION statement is executed. However, when the function is used in an SQL statement, the method identifier must exist and be accessible from the database server machine.

  The *string* can be specified as follows:



  Extraneous blanks are not permitted within the single quotation marks.

  jar_id
  Identifies the jar identifier given to the jar collection when it was installed in the database. It can be either a simple identifier, or a schema qualified identifier. Examples are 'myJar' and 'mySchema.myJar'

  class_id
  Identifies the class identifier of the Java object. If the class is part of a package, the class identifier part must include the complete package prefix, for example, 'myPacks.UserFuncs'. The directory the Java virtual machine will look in for the classes depends on the operating system, as illustrated in the following table:

Operating system	Directory the Java virtual machine will look in for the classes
Linux AIX	`'.../myPacks/UserFuncs/'`
Windows	`'...\myPacks\UserFuncs\'`

  method_id
  Identifies the method name of the Java object to be invoked.

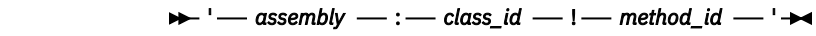
- For LANGUAGE CLR:

  The *string* specified represents the .NET assembly (library or executable), the class within that assembly, and the method within the class that the database manager invokes to execute the

function being created. The module, class, and method do not need to exist when the CREATE FUNCTION statement is executed. However, when the function is used in an SQL statement, the module, class, and method must exist and be accessible from the database server machine; otherwise, an error is returned (SQLSTATE 42724).

C++ routines that are compiled with the '/clr' compiler option to indicate that they include managed code extensions must be cataloged as 'LANGUAGE CLR' and not 'LANGUAGE C'. The database server needs to know that the .NET infrastructure is being utilized in a user-defined function in order to make necessary runtime decisions. All user-defined functions using the .NET infrastructure must be cataloged as 'LANGUAGE CLR'.

The *string* can be specified as follows:



The name must be enclosed by single quotation marks. Extraneous blanks are not permitted.

assembly
Identifies the DLL or other assembly file in which the class resides. Any file extensions (such as `.dll`) must be specified. If the full path name is not given, the file must reside in the function directory of the database product's installation path

For example, `c:\sqllib\function`.

If the file resides in a subdirectory of the install function directory, the subdirectory can be given before the file name rather than specifying the full path.

For example, if your install directory is `c:\sqllib` and your assembly file is `c:\sqllib\function\myprocs\mydotnet.dll`, it is only necessary to specify 'myprocs\mydotnet.dll' for the assembly.

The case sensitivity of this parameter is the same as the case sensitivity of the file system.

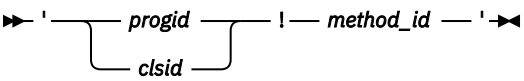
class_id
Specifies the name of the class within the given assembly in which the method that is to be invoked resides. If the class resides within a namespace, the full namespace must be given in addition to the class. For example, if the class EmployeeClass is in namespace MyCompany.ProcedureClasses, then MyCompany.ProcedureClasses.EmployeeClass must be specified for the class. Note that the compilers for some .NET languages will add the project name as a namespace for the class, and the behavior may differ depending on whether the command line compiler or the GUI compiler is used. This parameter is case sensitive.

method_id
Specifies the method within the given class that is to be invoked. This parameter is case sensitive.

- For LANGUAGE OLE:

The *string* specified is the OLE programmatic identifier (progid) or class identifier (clsid), and method identifier, which the database manager invokes to execute the user-defined function being created. The programmatic identifier or class identifier, and method identifier do not need to exist when the CREATE FUNCTION statement is executed. However, when the function is used in an SQL statement, the method identifier must exist and be accessible from the database server machine; otherwise, an error is returned (SQLSTATE 42724).

The *string* can be specified as follows:



Extraneous blanks are not permitted within the single quotation marks.

progid
Identifies the programmatic identifier of the OLE object.

progid is not interpreted by the database manager but only forwarded to the OLE APIs at run time. The specified OLE object must be creatable and support late binding (also called IDispatch-based binding).

clsid
Identifies the class identifier of the OLE object to create. It can be used as an alternative for specifying a *progid* in the case that an OLE object is not registered with a progid. The *clsid* has the form:

{nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn}

where 'n' is an alphanumeric character. *clsid* is not interpreted by the database manager but only forwarded to the OLE APIs at run time.

method_id
Identifies the method name of the OLE object to be invoked.

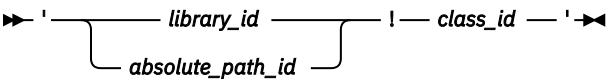
- For LANGUAGE CPP:

The *string* specified is the library identifier and class identifier within the library, which contains the evaluate method the database manager invokes to execute the user-defined function that is being created. If the string is not properly formed, an error is returned (SQLSTATE 42878).

It is not necessary that the library (or the class within the library) exist when the CREATE FUNCTION statement is executed. However, when the function is used in an SQL statement, the library and class within the library must exist and be accessible from the database server machine; otherwise, an error is returned (SQLSTATE 42724).

The body of every external function should be in a directory that is available on every database partition.

The *string* can be specified as follows:



Extraneous blanks are not permitted within the single quotation marks.

library_id
The name of the library that contains the function:

– On a UNIX system, if the specified library ID is `myfunc`, and if the database manager is being run from `/u/production`, the database manager looks for the function in the following library:

```
/u/production/sqllib/function/myfunc
```

– On a Windows operating system, the database manager looks for the function in the directory path specified by the LIBPATH or PATH environment variable.

absolute_path_id
The full path of the file that contains the function. For example:

– On a UNIX system, the following specification causes the database manager to look in `/u/jchui/mylib` for the myfunc shared library:

```
'/u/jchui/mylib/myfunc'
```

– On a Windows operating system, the following specification causes the database manager to load the dynamic link library myfunc.dll from the `d:\mylib` directory:

```
'd:\mylib\myfunc.dll'
```

If an absolute path ID is being used to identify the routine body, be sure to append the .dll extension.

class_id
The name of the class that contains the methods that are to be invoked.

For example, if you specify `'mymod!myclass'`:

- On a UNIX system, the database manager looks for the library `$inst_home_dir/sqllib/function/mymod` and invokes the evaluate method of the myclass class in that library.
- On a Windows operating system, the database manager loads the mymod.dll file and calls the evaluate method of the myclass class in the dynamic link library (DLL).

NAME *identifier*
This clause identifies the name of the user-written code which implements the function being defined. The *identifier* specified is an SQL identifier. The SQL identifier is used as the *library-id* in the string. Unless it is a delimited identifier, the identifier is folded to upper case. If the identifier is qualified with a schema name, the schema name portion is ignored. This form of NAME can only be used with LANGUAGE C.

LANGUAGE
This mandatory clause is used to specify the language interface convention to which the user-defined function body is written.

C
This means the database manager will call the user-defined function as if it were a C function. The user-defined function must conform to the C language calling and linkage convention as defined by the standard ANSI C prototype.

JAVA
This means the database manager will call the user-defined function as a method in a Java class.

CLR
The database manager calls the user-defined function as a method in a .NET class. LANGUAGE CLR is supported only for user-defined functions running on Windows operating systems. NOT FENCED cannot be specified for a CLR routine (SQLSTATE 42601).

OLE
The database manager calls the user-defined function as if it were a method exposed by an OLE automation object. The user-defined function must conform with the OLE automation data types and invocation mechanism, as described in the *OLE Automation Programmer's Reference*.

LANGUAGE OLE is supported for user-defined functions for this database product only in Windows 32-bit operating systems.

For information about creating LANGUAGE OLE DB external table functions, see "CREATE FUNCTION (OLE DB External Table)".

CPP
The database manager calls the user-defined function by invoking the evaluate method of a C++ class.

PARAMETER STYLE
This clause is used to specify the conventions used for passing parameters to and returning the value from functions.

DB2GENERAL
Used to specify the conventions for passing parameters to and returning the value from external functions that are defined as a method in a Java class. This can only be specified when LANGUAGE JAVA is used.

SQL
Used to specify the conventions for passing parameters to and returning the value from external functions that conform to C language calling and linkage conventions, methods exposed by OLE automation objects, or public static methods of a .NET object. This must be specified when LANGUAGE C, LANGUAGE CLR, or LANGUAGE OLE is used.

NPSGENERIC

Used to specify the conventions for passing parameters to and returning the value from external functions that are defined as a method in a C++ class. This can specified only when the LANGUAGE option is set to CPP.

When NPSGENERIC is specified as the parameter style, the UDF is written in C++ as a derived class of the nz.udx_ver2.Udf class. The class must implement the following two methods in addition to its constructor and destructor:

static Udf* Udf::instantiate(UdxInit *pInit)
Static member method instantiate(), which must instantiate a new instance of the UDF derived class and return a pointer to the new instance as a class Udf pointer. The engine uses this method to create an instance of a UDF object.

virtual ReturnValue Udf::evaluate()
Member method evaluate(), which is called by the engine to evaluate the user function and return a value to the caller.

PARAMETER CCSID

Specifies the encoding scheme to use for all string data passed into and out of the function. If the PARAMETER CCSID clause is not specified, the default is PARAMETER CCSID UNICODE for Unicode databases, and PARAMETER CCSID ASCII for all other databases.

ASCII
Specifies that string data is encoded in the database code page. If the database is a Unicode database, PARAMETER CCSID ASCII cannot be specified (SQLSTATE 56031). When the function is invoked, the application code page for the function is the database code page.

UNICODE
Specifies that string data is encoded in Unicode. If the database is a Unicode database, character data is in UTF-8, and graphic data is in UCS-2. If the database is not a Unicode database, character data is in UTF-8. In either case, when the function is invoked, the application code page for the function is 1208.

If the database is not a Unicode database, and a function with PARAMETER CCSID UNICODE is created, the function cannot have any graphic types or user-defined types (SQLSTATE 560C1).

If the database is not a Unicode database, table functions can be created with PARAMETER CCSID UNICODE, but the following rules apply:

- The alternate collating sequence must be specified in the database configuration before creating the table function (SQLSTATE 56031). PARAMETER CCSID UNICODE table functions collate with the alternate collating sequence specified in the database configuration.
- Tables or table functions created with CCSID ASCII, and tables or table functions created with CCSID UNICODE, cannot both be used in a single SQL statement (SQLSTATE 53090). This applies to tables and table functions referenced directly in the statement, as well as to tables and table functions referenced indirectly (such as, for example, through referential integrity constraints, triggers, materialized query tables, and tables in the body of views).
- Table functions created with PARAMETER CCSID UNICODE cannot be referenced in SQL functions or SQL methods (SQLSTATE 560C0).
- An SQL statement that references a table function created with PARAMETER CCSID UNICODE cannot invoke an SQL function or SQL method (SQLSTATE 53090).
- Graphic types, the XML type, and user-defined types cannot be used as parameters to PARAMETER CCSID UNICODE table functions (SQLSTATE 560C1).
- SQL statements are always interpreted in the database code page. In particular, this means that every character in literals, hex literals, and delimited identifiers must have a representation in the database code page; otherwise, the character will be replaced with the substitution character.

If the database is not a Unicode database, and the alternate collating sequence has been specified in the database configuration, functions can be created with either PARAMETER CCSID ASCII or

PARAMETER CCSID UNICODE. All string data passed into and out of the function will be converted to the appropriate code page.

This clause cannot be specified with LANGUAGE CPP, LANGUAGE OLE, LANGUAGE JAVA, or LANGUAGE CLR (SQLSTATE 42613).

DETERMINISTIC or NOT DETERMINISTIC

This optional clause specifies whether the function always returns the same results for given argument values (DETERMINISTIC) or whether the function depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC function must always return the same table from successive invocations with identical inputs. Optimizations taking advantage of the fact that identical inputs always produce the same results are prevented by specifying NOT DETERMINISTIC. An example of a table function that is non-deterministic is one that references special registers, global variables, non-deterministic functions, or sequences in a way that affects the table function result table.

FENCED or NOT FENCED

This clause specifies whether the function is considered "safe" to run in the database manager operating environment's process or address space (NOT FENCED), or not (FENCED).

If a function is registered as FENCED, the database manager protects its internal resources (for example, data buffers) from access by the function. Most functions will have the option of running as FENCED or NOT FENCED. In general, a function running as FENCED will not perform as well as a similar one running as NOT FENCED.

CAUTION: Use of NOT FENCED for functions not adequately coded, reviewed and tested can compromise the integrity of your database. This database product safeguards against many of the common types of inadvertent failures that might occur, but cannot guarantee complete integrity when NOT FENCED user defined functions are used.

Only FENCED can be specified for a function with LANGUAGE OLE or NOT THREADSAFE (SQLSTATE 42613).

If the function is FENCED and has the NO SQL option, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

Either SYSADM authority, DBADM authority, or a special authority (CREATE_NOT_FENCED_ROUTINE) is required to register a user-defined function as NOT FENCED.

LANGUAGE CLR user-defined functions cannot be created when specifying the NOT FENCED clause (SQLSTATE 42601).

THREADSAFE or NOT THREADSAFE

Specifies whether the function is considered safe to run in the same process as other routines (THREADSAFE), or not (NOT THREADSAFE).

If the function is defined with LANGUAGE other than OLE:

- If the function is defined as THREADSAFE, the database manager can invoke the function in the same process as other routines. In general, to be threadsafe, a function should not use any global or static data areas. Most programming references include a discussion of writing threadsafe routines. Both FENCED and NOT FENCED functions can be THREADSAFE.
- If the function is defined as NOT THREADSAFE, the database manager will never simultaneously invoke the function in the same process as another routine.

For FENCED functions, THREADSAFE is the default if the LANGUAGE is JAVA or CLR. For all other languages, NOT THREADSAFE is the default. If the function is defined with LANGUAGE OLE, THREADSAFE may not be specified (SQLSTATE 42613).

For NOT FENCED functions, THREADSAFE is the default. NOT THREADSAFE cannot be specified (SQLSTATE 42613).

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT

This optional clause may be used to avoid a call to the external function if any of the arguments is null. If the user-defined function is defined to have no parameters, then of course this null argument condition cannot arise, and it does not matter how this specification is coded.

If RETURNS NULL ON NULL INPUT is specified, and if, at table function OPEN time, any of the function's arguments are null, then the user-defined function is not called. The result of the attempted table function scan is the empty table (a table with no rows).

If CALLED ON NULL INPUT is specified, then regardless of whether any arguments are null, the user-defined function is called. It can return a null value or a normal (non-null) value. But responsibility for testing for null argument values lies with the UDF.

The value NULL CALL may be used as a synonym for CALLED ON NULL INPUT for backwards and family compatibility. Similarly, NOT NULL CALL may be used as a synonym for RETURNS NULL ON NULL INPUT.

READS SQL DATA, NO SQL, CONTAINS SQL

Specifies the classification of SQL statements that the function can run. The database manager verifies that the SQL statements that the function issues are consistent with this specification.

For the classification of each statement, see "SQL statements that can be executed in routines and triggers" in Developing User-defined Routines (SQL and External).

The default is READS SQL DATA.

READS SQL DATA

Specifies that the function can run statements with a data access classification of READS SQL DATA, CONTAINS SQL, or NO SQL (SQLSTATE 38002 or 42985). The function cannot run SQL statements that modify data (SQLSTATE 38003 or 42985).

NO SQL

Specifies that the function can run only SQL statements with a data access classification of NO SQL. If the ALLOW PARALLEL, EXECUTE ON ALL DATABASE PARTITIONS, and RESULT TABLE DISTRIBUTED clauses are all specified, NO SQL is the only option allowed.

CONTAINS SQL

Specifies that the function can run only SQL statements with a data access classification of CONTAINS SQL or NO SQL (SQLSTATE 38004 or 42985). The function cannot run any SQL statements that read or modify data (SQLSTATE 38003 or 42985).

STATIC DISPATCH

This optional clause indicates that at function resolution time, a function is chosen by the database server based on the static types (declared types) of the parameters of the function.

EXTERNAL ACTION or NO EXTERNAL ACTION

Specifies whether the function takes an action that changes the state of an object that the database manager does not manage. An example of an external action is sending a message or writing a record to a file. The default is EXTERNAL ACTION.

EXTERNAL ACTION

Specifies that the function takes an action that changes the state of an object that the database manager does not manage.

A function with external actions might return incorrect results if the function is executed by parallel tasks. For example, if the function sends a note for each initial call to it, one note is sent for each parallel task instead of once for the function. Specify the DISALLOW PARALLEL clause for functions that do not work correctly with parallelism.

NO EXTERNAL ACTION

Specifies that the function does not take any action that changes the state of an object that the database manager does not manage. The database manager uses this information during optimization of SQL statements.

NO SCRATCHPAD or SCRATCHPAD *length*

This optional clause may be used to specify whether a scratchpad is to be provided for an external function. (It is strongly recommended that user-defined functions be re-entrant, so a scratchpad provides a means for the function to "save state" from one call to the next.)

- If SCRATCHPAD is specified, then at first invocation of the user-defined function, memory is allocated for a scratchpad to be used by the external function. On each invocation of the user-defined function, an additional argument is passed to the external function which addresses the scratchpad. The scratchpad has the following characteristics:
  - *length*, if specified, sets the size of the scratchpad in bytes and must be between 1 and 32,767 (SQLSTATE 42820). The default value is 100.
  - It is initialized to all X'00''s.
  - Its scope is the SQL statement. There is one scratchpad per reference to the external function in the SQL statement. So if the UDFX function in the following statement is defined with the SCRATCHPAD keyword, two scratchpads would be assigned.

    ```
    SELECT A.C1, B.C2
      FROM TABLE (UDFX(:hv1)) AS A,
           TABLE (UDFX(:hv1)) AS B
        WHERE ...
    ```

  - It is persistent. It is initialized at the beginning of the execution of the statement, and can be used by the external table function to preserve the state of the scratchpad from one call to the next. If the FINAL CALL keyword is also specified for the UDF, then the scratchpad is NEVER altered, and any resources anchored in the scratchpad should be released when the special FINAL call is made.

    If NO FINAL CALL is specified or defaulted, then the external table function should clean up any such resources on the CLOSE call, as the database server will re-initialize the scratchpad on each OPEN call. This determination of FINAL CALL or NO FINAL CALL and the associated behavior of the scratchpad could be an important consideration, particularly if the table function will be used in a subquery or join, since that is when multiple OPEN calls can occur during the execution of a statement.
  - It can be used as a central point for system resources (for example, memory) which the external function might acquire. The function could acquire the memory on the first call, keep its address in the scratchpad, and refer to it in subsequent calls.

    (As previously outlined, the FINAL CALL/NO FINAL CALL keyword is used to control the re-initialization of the scratchpad, and also dictates when the external table function should release resources anchored in the scratchpad.)
- If NO SCRATCHPAD is specified then no scratchpad is allocated or passed to the external function.

FINAL CALL or NO FINAL CALL

This optional clause specifies whether a final call (and a separate first call) is to be made to an external function. It also controls when the scratchpad is re-initialized. If NO FINAL CALL is specified, then the database server can only make three types of calls to the table function: open, fetch and close. However, if FINAL CALL is specified, then in addition to open, fetch and close, a first call and a final call can be made to the table function.

For external table functions, the call-type argument is ALWAYS present, regardless of which option is chosen.

If the final call is being made because of an interrupt or end-of-transaction, the UDF may not issue any SQL statements except for CLOSE cursor (SQLSTATE 38505). A special value is passed in the "call type" argument for these special final call situations.

FINAL CALL cannot be specified in combination with LANGUAGE CPP.

DISALLOW PARALLEL or ALLOW PARALLEL EXECUTE ON ALL DATABASE PARTITIONS RESULT TABLE DISTRIBUTED

Specifies whether or not, for a single reference to the function, the invocation of the function is to be parallelized.

DISALLOW PARALLEL

Specifies that on each invocation of the function, the function is invoked on a single database partition.

ALLOW PARALLEL EXECUTE ON ALL DATABASE PARTITIONS RESULT TABLE DISTRIBUTED

Specifies that on each invocation of the function, the function is invoked on all database partitions. The union of the result sets obtained on each database partition is returned. The function cannot execute SQL statements (the NO SQL clause must also be specified).

NO DBINFO or DBINFO

This optional clause specifies whether certain specific information known to the database server is to be passed to the function as an additional invocation-time argument (DBINFO) or not (NO DBINFO). NO DBINFO is the default. DBINFO is not supported for LANGUAGE OLE or PARAMETER TYLE NPSGENERIC (SQLSTATE 42613).

If DBINFO is specified, a structure containing the following information is passed to the function:

- Database name - the name of the currently connected database
- Application ID - the unique application ID that is established for each connection to the database
- Application authorization ID - the application runtime authorization ID, regardless of any nested functions between this function and the application
- Code page - the database code page
- Schema name - not applicable to external table functions
- Table name - not applicable to external table functions
- Column name - not applicable to external table functions
- Database version or release - the version, release, and modification level of the database server that is invoking the function
- Platform - the server's platform type
- Table function result column numbers - an array of result column numbers that is used by the statement referencing the function; this information enables the function to return only required column values instead of all column values
- Database partition number - the number of the database partition on which the external table function is invoked; in a single database partition environment, this value is 0

CARDINALITY *integer*

This optional clause provides an estimate of the expected number of rows to be returned by the function for optimization purposes. Valid values range from 0 to 9,223,372,036,854,775,807.

If the CARDINALITY clause is not specified for a table function, assume a finite value is assumed as a default; the same value assumed for tables for which the RUNSTATS utility has not gathered statistics.

Warning: If a function has infinite cardinality (that is, if it returns a row every time it is called to do so, and never returns the "end-of-table" condition), then queries that require the end-of-table condition to correctly function will never terminate and will have to be interrupted. Examples of such queries are those that contain a GROUP BY or an ORDER BY clause. For that reason, writing UDFs that have infinite cardinality is not recommended.

TRANSFORM GROUP *group-name*

Indicates the transform group to be used for user-defined structured type transformations when invoking the function. A transform is required if the function definition includes a user-defined structured type as a parameter data type. If this clause is not specified, the default group name DB2_FUNCTION is used. If the specified (or default) *group-name* is not defined for a referenced structured type, an error results (SQLSTATE 42741). If a required FROM SQL transform function is not defined for the given group-name and structured type, an error results (SQLSTATE 42744).

INHERIT SPECIAL REGISTERS
This optional clause specifies that updatable special registers in the function will inherit their initial values from the environment of the invoking statement. For a function invoked in the select-statement of a cursor, the initial values are inherited from the environment when the cursor is opened. For a routine invoked in a nested object (for example a trigger or view), the initial values are inherited from the runtime environment (not inherited from the object definition).

No changes to the special registers are passed back to the invoker of the function.

Non-updatable special registers, such as the datetime special registers, reflect a property of the statement currently executing, and are therefore set to their default values.

NOT SECURED or SECURED
Specifies whether the function is considered secure for row and column access control. The default is NOT SECURED.

NOT SECURED
Indicates that the function is not considered secure. When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled and column level access control is activated for its table (SQLSTATE 428HA). This rule applies to the non secure user-defined functions that are invoked anywhere in the statement.

SECURED
Indicates that the function is considered secure. The function must be secure when it is referenced in a row permission or a column mask (SQLSTATE 428H8, SQLCODE -20470).

STAY RESIDENT NO
Specifies that the library that is loaded for the function is not to remain resident in memory after the function ends. This clause is ignored when:

- The NOT FENCED clause is specified.
- The LANGUAGE option is set to JAVA or CLR.

Rules

- In a partitioned database environment, the use of SQL in external user-defined functions or methods is not supported (SQLSTATE 42997).
- Only routines defined as NO SQL can be used to define an index extension (SQLSTATE 428F8).
- If the function allows SQL, the external program must not attempt to access any federated objects (SQLSTATE 55047).
- ***Table access restrictions*** If a function is defined as READS SQL DATA, no statement in the function can access a table that is being modified by the statement which invoked the function (SQLSTATE 57053). For example, suppose the user-defined function BONUS() is defined as READS SQL DATA. If the statement UPDATE EMPLOYEE SET SALARY = SALARY + BONUS(EMPNO) is invoked, no SQL statement in the BONUS function can read from the EMPLOYEE table.

Notes

- When choosing the data types for the parameters of a user-defined function, consider the rules for promotion that will affect its input values. For example, a constant which may be used as an input value could have a built-in data type that is different from the one expected and, more significantly, may not be promoted to the data type expected. Based on the rules for promotion, it is generally recommended to use the following data types for parameters:
  - INTEGER instead of SMALLINT
  - DOUBLE instead of REAL
  - VARCHAR instead of CHAR
  - VARGRAPHIC instead of GRAPHIC
- For portability of UDFs across platforms, it is recommended to use the following data types:

  - DOUBLE or REAL instead of FLOAT
  - DECIMAL instead of NUMERIC
  - CLOB (or BLOB) instead of LONG VARCHAR
- Creating a function with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- A Java routine defined as NOT FENCED will be invoked as if it had been defined as FENCED THREADSAFE.
- ***Privileges***: The definer of a function always receives the EXECUTE privilege WITH GRANT OPTION on the function, as well as the right to drop the function. When the function is used in an SQL statement, the function definer must have the EXECUTE privilege on any packages used by the function or EXECUTEIN privilege or DATAACCESS authority on the schema containing the packages. .
- ***Setting of the default value***: Parameters of a function that are defined with a default value are set to their default value when the functions is invoked, but only if a value is not supplied for the corresponding argument, or is specified as DEFAULT, when the function is invoked.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of this database product and with other database products. These alternatives are non-standard and should not be used.
  - PARAMETER STYLE DB2SQL can be specified in place of PARAMETER STYLE SQL
  - NOT VARIANT can be specified in place of DETERMINISTIC
  - VARIANT can be specified in place of NOT DETERMINISTIC
  - NULL CALL can be specified in place of CALLED ON NULL INPUT
  - NOT NULL CALL can be specified in place of RETURNS NULL ON NULL INPUT
  - DB2GENRL can be specified in place of DB2GENERAL

  The following syntax is accepted as the default behavior:
  - ASUTIME NO LIMIT
  - NO COLLID
  - PROGRAM TYPE SUB
  - STAY RESIDENT NO
  - CCSID UNICODE in a Unicode database
  - CCSID ASCII in a non-Unicode database if PARAMETER CCSID UNICODE is not specified
- ***Creating a secure function***: Normally users with SECADM authority do not have privileges to create database objects such as triggers and functions. Typically they will examine the data accessed by the function, ensure it is secure, then grant the CREATE_SECURE_OBJECT authority to someone who currently has required privileges to create a secure user-defined function. After the function is created, they will revoke the CREATE_SECURE_OBJECT authority from the function owner.

  The SECURED attribute is considered to be an assertion that declares the user has established a change control audit procedure for all changes to the user-defined function. The database manager assumes that such a control audit procedure is in place for all subsequent ALTER FUNCTION statements or changes to external packages.
- ***Invoking other user-defined functions in a secure function***: If a secure user-defined function invokes other user-defined functions, the database manager does not validate whether those nested user-defined functions have the SECURED attribute. If those nested functions can access sensitive data, the user with SECADM authority needs to ensure those functions are allowed to access those data and a change control audit procedure has been established for all changes to those functions.
- ***Replacing an existing function such that the secure attribute is changed (from SECURED to NOT SECURED and vice versa)***: Packages and dynamically cached SQL statements that depend on the function may be invalidated because the secure attribute affects the access path selection for statements involving tables for which row or column level access control is activated.

- ***EXTERNAL ACTION functions***: If an EXTERNAL ACTION function is invoked in other than the outermost select list, the results are unpredictable since the number of times the function is invoked will vary depending on the access plan used.

Examples

- *Example 1:* The following example registers a table function written to return a row consisting of a single document identifier column for each known document in a text management system. The first parameter matches a given subject area and the second parameter contains a given string.

  Within the context of a single session, the UDF will always return the same table, and therefore it is defined as DETERMINISTIC. Note the RETURNS clause which defines the output from DOCMATCH. FINAL CALL must be specified for each table function. In addition, the DISALLOW PARALLEL keyword is added as table functions cannot operate in parallel. Although the size of the output for DOCMATCH is highly variable, CARDINALITY 20 is a representative value, and is specified to help the database optimizer.

```
CREATE FUNCTION DOCMATCH (VARCHAR(30), VARCHAR(255))
  RETURNS TABLE (DOC_ID CHAR(16))
  EXTERNAL NAME '/common/docfuncs/rajiv/udfmatch'
  LANGUAGE C
  PARAMETER STYLE SQL
  NO SQL
  DETERMINISTIC
  NO EXTERNAL ACTION
  NOT FENCED
  SCRATCHPAD
  FINAL CALL
  DISALLOW PARALLEL
  CARDINALITY 20
```

- *Example 2:* The following example registers an OLE table function that is used to retrieve message header information and the partial message text of messages in Microsoft Exchange.

```
CREATE FUNCTION MAIL()
  RETURNS TABLE (TIMERECEIVED DATE,
                 SUBJECT VARCHAR(15),
                 SIZE INTEGER,
                 TEXT VARCHAR(30))
  EXTERNAL NAME 'tfmail.header!list'
  LANGUAGE OLE
  PARAMETER STYLE SQL
  NOT DETERMINISTIC
  FENCED
  CALLED ON NULL INPUT
  SCRATCHPAD
  FINAL CALL
  NO SQL
  EXTERNAL ACTION
  DISALLOW PARALLEL
```

CREATE FUNCTION (OLE DB external table)

The CREATE FUNCTION (OLE DB External Table) statement is used to register a user-defined OLE DB external table function to access data from an OLE DB provider.

A *table function* can be used in the FROM clause of a SELECT.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

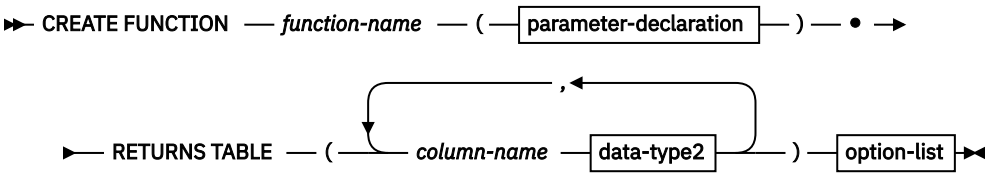
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- CREATE_EXTERNAL_ROUTINE authority on the database and at least one of the following authorities:
  - IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the function does not exist
  - CREATEIN privilege on the schema, if the schema name of the function exists
  - SCHEMAADM authority on the schema, if the schema name of the function exists
- DBADM authority

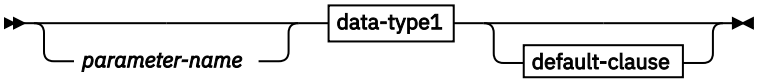
Group privileges are not considered for any table or view specified in the CREATE FUNCTION statement.

If the SECURED option is specified, the authorization ID of the statement must include SECADM authority or CREATE_SECURE_OBJECT authority (SQLSTATE 42501).

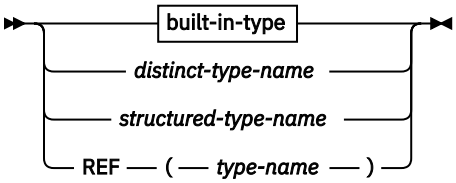
Syntax



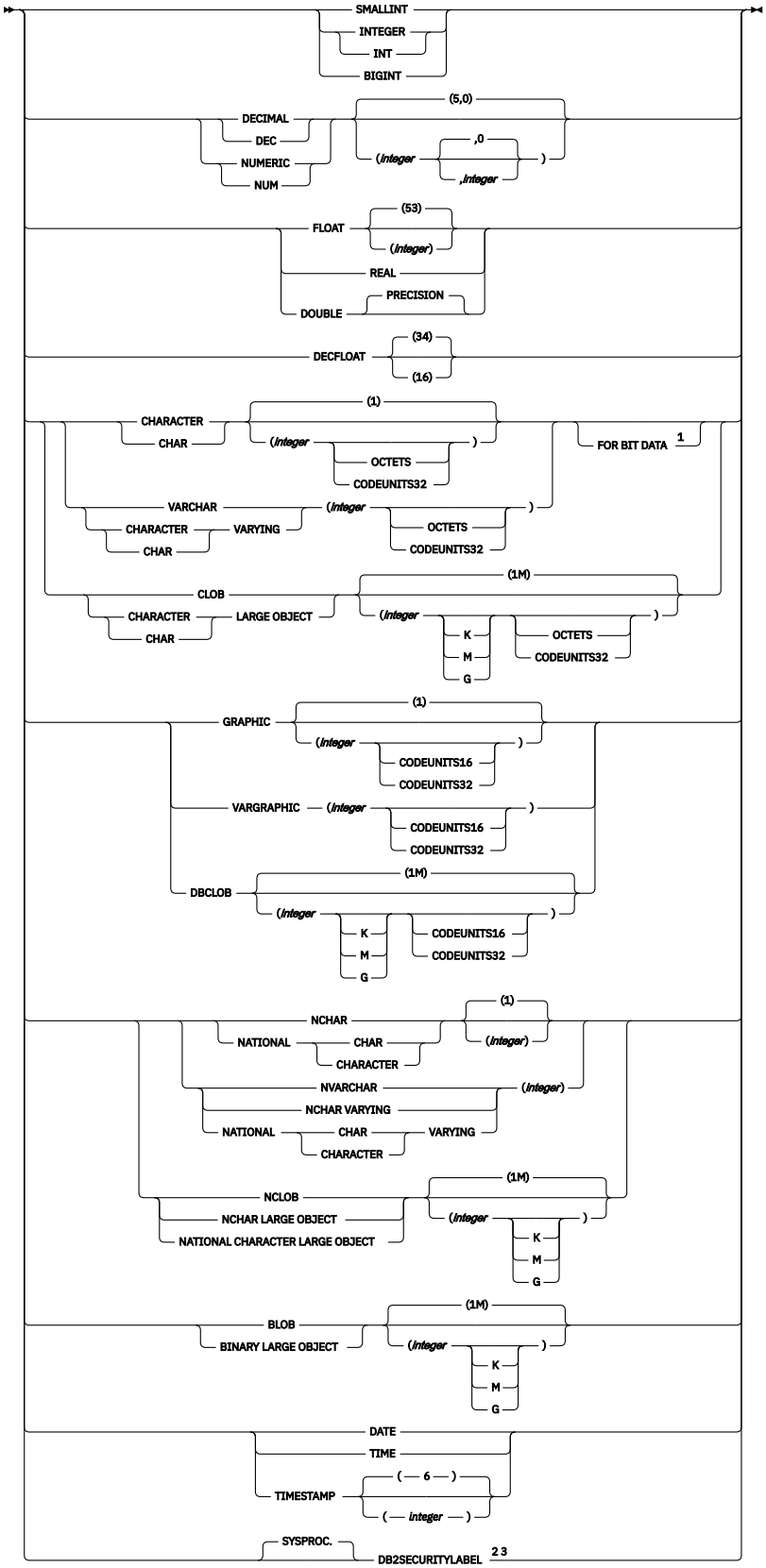
parameter-declaration



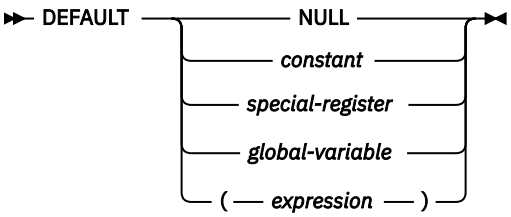
data-type1, data-type2



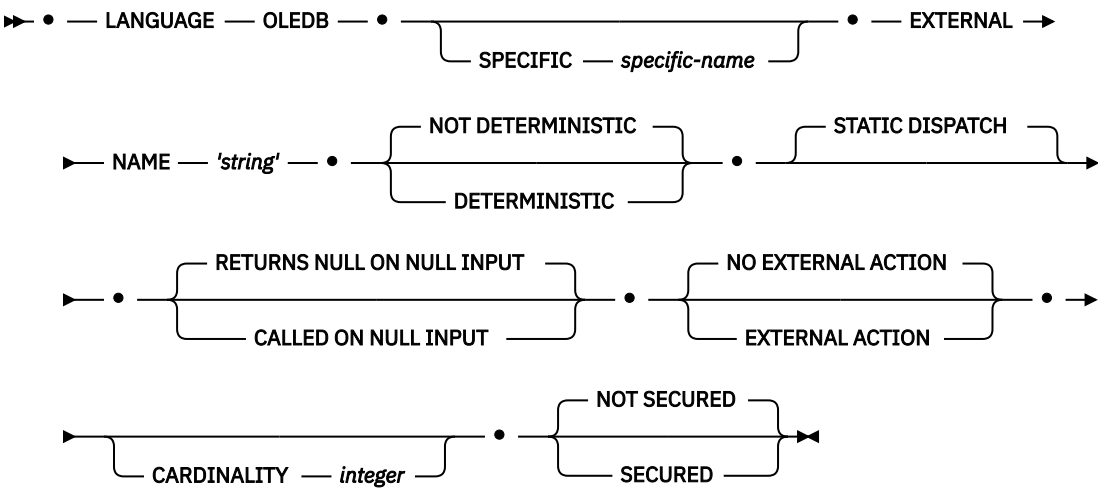
built-in-type



default-clause



option-list



Notes:

[1] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

[2] DB2SECURITYLABEL is the built-in distinct type that must be used to define the row security label column of a protected table.

[3] For a column of type DB2SECURITYLABEL, NOT NULL WITH DEFAULT is implicit and cannot be explicitly specified (SQLSTATE 42842). The default value for a column of type DB2SECURITYLABEL is the session authorization ID's security label for write access.

Description

function-name

Names the function being defined. It is a qualified or unqualified name that designates a function. The unqualified form of *function-name* is an SQL identifier. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a *schema-name* followed by a period and an SQL identifier.

The name, including the implicit or explicit qualifiers, together with the number of parameters and the data type of each parameter (without regard for any length, precision or scale attributes of the data type) must not identify a function described in the catalog (SQLSTATE 42723). The unqualified name, together with the number and data types of the parameters, while of course unique within its schema, need not be unique across schemas.

If a two-part name is specified, the *schema-name* cannot begin with 'SYS' (SQLSTATE 42939).

A number of names used as keywords in predicates are reserved for system use, and cannot be used as a *function-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators.

The same name can be used for more than one function if there is some difference in the signature of the functions. Although there is no prohibition against it, an external user-defined table function should not be given the same name as a built-in function.

(*parameter-declaration*,...)

Identifies the number of input parameters of the function, and specifies the data type and optional default value of each parameter. If no input parameter is specified, data is retrieved from the external source possibly subsetted through query optimization. The input parameter passes command text to an OLE DB provider.

It is possible to register a function that has no parameters. In this case, the parentheses must still be coded, with no intervening data types. For example:

```
CREATE FUNCTION WOOFER() ...
```

No two identically-named functions within a schema are permitted to have exactly the same type for all corresponding parameters. Lengths, precisions, and scales are not considered in this type comparison. Therefore, CHAR(8) and CHAR(35) are considered to be the same type. A weakly typed distinct type specified for a parameter is considered to be the same data type as the source type of the distinct type. For a Unicode database, CHAR(13) and GRAPHIC(8) are considered to be the same type. A duplicate signature returns an error (SQLSTATE 42723).

parameter-name

Specifies an optional name for the input parameter.

data-type1

Specifies the data type of the input parameter. The data type can be any character or graphic string data type or a distinct type based on a character or graphic string data type. Parameters of type BINARY, VARBINARY, and XML are not supported (SQLSTATE 42815).

For a more complete description of each built-in data type, see "CREATE TABLE".

For a user-defined distinct type, the length, precision, or scale attributes for the parameter are those of the source type of the distinct type (those specified on CREATE TYPE). A distinct type parameter is passed as the source type of the distinct type. If the name of the distinct type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

DEFAULT

Specifies a default value for the parameter. The default can be a constant, a special register, a global variable, an expression, or the keyword NULL. The special registers that can be specified as the default are that same as those that can be specified for a column default (see *default-clause* in the CREATE TABLE statement). Other special registers can be specified as the default by using an expression.

The expression can be any expression of the type described in "Expressions". If a default value is not specified, the parameter has no default and the corresponding argument cannot be omitted on invocation of the procedure. The maximum size of the *expression* is 64K bytes.

The default expression must not modify SQL data (SQLSTATE 428FL or SQLSTATE 429BL). The expression must be assignment compatible to the parameter data type (SQLSTATE 42821).

A default cannot be specified for a parameter of type ARRAY, ROW, or CURSOR (SQLSTATE 429BB).

RETURNS TABLE

Specifies that the output of the function is a table. The parentheses that follow this keyword delimit a list of the names and types of the columns of the table, resembling the style of a simple CREATE TABLE statement which has no additional specifications (constraints, for example).

column-name

Specifies the name of the column which must be the same as the corresponding rowset column name. The name cannot be qualified and the same name cannot be used for more than one column of the table.

data-type2

Specifies the data type of the column. BINARY, VARBINARY, and XML are not supported (SQLSTATE 42815).

built-in-type
See "CREATE TABLE" for the description of built-in data types.

SPECIFIC *specific-name*
Provides a unique name for the instance of the function that is being defined. This specific name can be used when sourcing on this function, dropping the function, or commenting on the function. It can never be used to invoke the function. The unqualified form of *specific-name* is an SQL identifier. The qualified form is a *schema-name* followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another function instance that exists at the application server; otherwise an error (SQLSTATE 42710) is raised.

The *specific-name* may be the same as an existing *function-name*.

If no qualifier is specified, the qualifier that was used for *function-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *function-name* or an error (SQLSTATE 42882) is raised.

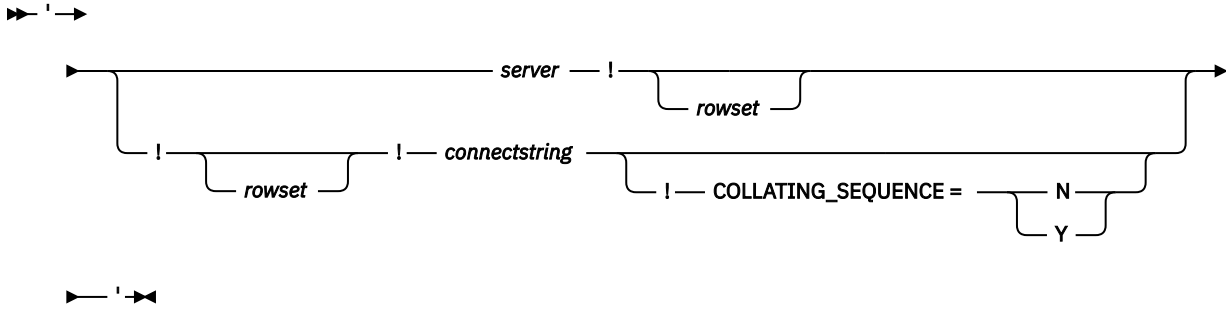
If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmssxxx.

EXTERNAL NAME *'string'*
This clause identifies the external table and an OLE DB provider.

The 'string' option is a string constant with a maximum of 254 bytes.

The string specified is used to establish a connection and session with an OLE DB provider, and retrieve data from a rowset. The OLE DB provider and data source do not need to exist when the CREATE FUNCTION statement is executed.

The *string* can be specified as follows:



server
Identifies the local name of a data source as defined by "CREATE SERVER".

rowset
Identifies the rowset (table) exposed by the OLE DB provider. Fully qualified table names must be provided for OLE DB providers that support catalog or schema names.

connectstring
String version of the initialization properties needed to connect to a data source. The basic format of a connection string is based on the ODBC connection string. The string contains a series of keyword/value pairs separated by semicolons. The equal sign (=) separates each keyword and its value. Keywords are the descriptions of the OLE DB initialization properties (property set DBPROPSET_DBINIT) or provider-specific keywords.

COLLATING_SEQUENCE
Specifies whether the data source uses the same collating sequence as Db2. For details, see "CREATE SERVER". Valid values are as follows:

- Y = Same collating sequence
- N = Different collating sequence

If COLLATING_SEQUENCE is not specified, the data source is assumed to have a different collating sequence than Db2.

If *server* is provided, *connectstring* or COLLATING_SEQUENCE are not allowed in the external name. They are defined as server options CONNECTSTRING and COLLATING_SEQUENCE. If no *server* is provided, a *connectstring* must be provided. If *rowset* is not provided, the table function must have an input parameter to pass through command text to the OLE DB provider.

LANGUAGE OLEDB
This means the database manager will deploy a built-in generic OLE DB consumer to retrieve data from the OLE DB provider. No table function implementation is required by the developer.

LANGUAGE OLEDB table functions can be created on any platform, but only executed on platforms supported by Microsoft OLE DB.

DETERMINISTIC or NOT DETERMINISTIC
This optional clause specifies whether the function always returns the same results for given argument values (DETERMINISTIC) or whether the function depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC function must always return the same table from successive invocations with identical inputs. Optimizations taking advantage of the fact that identical inputs always produce the same results are prevented by specifying NOT DETERMINISTIC.

STATIC DISPATCH
This optional clause indicates that at function resolution time, the database manager chooses a function based on the static types (declared types) of the parameters of the function.

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT
This optional clause may be used to avoid a call to the external function if any of the arguments is null. If the user-defined function is defined to have no parameters, then of course this null argument condition cannot arise.

If RETURNS NULL ON NULL INPUT is specified and if at execution time any one of the function's arguments is null, the user-defined function is not called and the result is the empty table; that is, a table with no rows.

If CALLED ON NULL INPUT is specified, then at execution time regardless of whether any arguments are null, the user-defined function is called. It can return an empty table or not, depending on its logic. But responsibility for testing for null argument values lies with the UDF.

The value NULL CALL may be used as a synonym for CALLED ON NULL INPUT for backwards and family compatibility. Similarly, NOT NULL CALL may be used as a synonym for RETURNS NULL ON NULL INPUT.

NO EXTERNAL ACTION or EXTERNAL ACTION
Specifies whether the function takes an action that changes the state of an object that the database manager does not manage. An example of an external action is sending a message or writing a record to a file. The default is NO EXTERNAL ACTION.

NO EXTERNAL ACTION
Specifies that the function does not take any action that changes the state of an object that the database manager does not manage. The database manager uses this information during optimization of SQL statements.

EXTERNAL ACTION
Specifies that the function takes an action that changes the state of an object that the database manager does not manage.

CARDINALITY *integer*
This optional clause provides an estimate of the expected number of rows to be returned by the function for optimization purposes. Valid values for *integer* range from 0 to 2 147 483 647 inclusive.

If the CARDINALITY clause is not specified for a table function, a finite value is assumed as the default. The finite value is the same value that is assumed for tables for which the RUNSTATS utility has not gathered statistics.

Warning: If a function does, in fact, have infinite cardinality - that is, it returns a row every time it is called to do so, and never returns the "end-of-table" condition - then queries that require the

end-of-table condition to correctly function will be infinite, and will have to be interrupted. Examples of such queries are those that contain a GROUP BY or an ORDER BY clause. Writing such UDFs is not recommended.

NOT SECURED or SECURED
Specifies whether the function is considered secure for row and column access control. The default is NOT SECURED.

NOT SECURED
Indicates that the function is not considered secure. When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled and column level access control is activated for its table (SQLSTATE 428HA). This rule applies to the non secure user-defined functions that are invoked anywhere in the statement.

SECURED
Indicates that the function is considered secure. The function must be secure when it is referenced in a row permission or a column mask (SQLSTATE 428H8).

Notes

- FENCED, FINAL CALL, SCRATCHPAD, PARAMETER STYLE SQL, DISALLOW PARALLEL, NO DBINFO, NOT THREADSAFE, and NO SQL are implicit in the statement and can be specified.
- When choosing the data types for the parameters of a user-defined function, consider the rules for promotion that will affect its input values. For example, a constant which may be used as an input value could have a built-in data type that is different from the one expected and, more significantly, may not be promoted to the data type expected. Based on the rules for promotion, it is generally recommended to use the following data types for parameters:
  - VARCHAR instead of CHAR
  - VARGRAPHIC instead of GRAPHIC
- For portability of UDFs across platforms, it is recommended to use the following data types:
  - DOUBLE or REAL instead of FLOAT
  - DECIMAL instead of NUMERIC
  - CLOB (or BLOB) instead of LONG VARCHAR
- Creating a function with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- ***Privileges***: The definer of a function always receives the EXECUTE privilege WITH GRANT OPTION on the function, as well as the right to drop the function.
- ***Setting of the default value***: Parameters of a function that are defined with a default value are set to their default value when the functions is invoked, but only if a value is not supplied for the corresponding argument, or is specified as DEFAULT, when the function is invoked.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - NOT VARIANT can be specified in place of DETERMINISTIC
  - VARIANT can be specified in place of NOT DETERMINISTIC
  - NULL CALL can be specified in place of CALLED ON NULL INPUT
  - NOT NULL CALL can be specified in place of RETURNS NULL ON NULL INPUT
- ***Creating a secure function***: Normally users with SECADM authority do not have privileges to create database objects such as triggers or functions. Typically they will examine the data accessed by the function, ensure it is secure, then grant the CREATE_SECURE_OBJECT authority to someone who currently has required privileges to create a secure user-defined function. After the function is created, they will revoke the CREATE_SECURE_OBJECT authority from the function owner.

The SECURED attribute is considered to be an assertion that declares the user has established a change control audit procedure for all changes to the user-defined function. The database manager assumes that such a control audit procedure is in place for all subsequent ALTER FUNCTION statements or changes to external packages.

- ***Invoking other user-defined functions in a secure function***: If a secure user-defined function invokes other user-defined functions, the database manager does not validate whether those nested user-defined functions have the SECURED attribute. If those nested functions can access sensitive data, the user with SECADM authority needs to ensure those functions are allowed to access those data and a change control audit procedure has been established for all changes to those functions.
- ***EXTERNAL ACTION functions***: If an EXTERNAL ACTION function is invoked in other than the outermost select list, the results are unpredictable since the number of times the function is invoked will vary depending on the access plan used.

Examples

1. Register an OLE DB table function, which retrieves order information from a Microsoft Access database. The connection string is defined in the external name.

```
CREATE FUNCTION orders ()
  RETURNS TABLE (orderid INTEGER,
                 customerid CHAR(5),
                 employeeid INTEGER,
                 orderdate TIMESTAMP,
                 requireddate TIMESTAMP,
                 shippeddate TIMESTAMP,
                 shipvia INTEGER,
                 freight dec(19,4))
  LANGUAGE OLEDB
  EXTERNAL NAME '!orders!Provider=Microsoft.Jet.OLEDB.3.51;
                       Data Source=c:\sqllib\samples\oledb\nwind.mdb
  !COLLATING_SEQUENCE=Y';
```

2. Register an OLE DB table function, which retrieves customer information from an Oracle database. The connection string is provided through a server definition. The table name is fully qualified in the external name. The local user john is mapped to the remote user dave. Other users will use the guest user ID in the connection string.

```
CREATE SERVER spirit
  WRAPPER OLEDB
  OPTIONS (CONNECTSTRING 'Provider=MSDAORA;Persist Security Info=False;
                          User ID=guest;password=pwd;Locale Identifier=1033;
                          OLE DB Services=CLIENTCURSOR;Data Source=spirit');

CREATE USER MAPPING FOR john
  SERVER spirit
  OPTIONS (REMOTE_AUTHID 'dave', REMOTE_PASSWORD 'mypwd');

CREATE FUNCTION customers ()
  RETURNS TABLE (customer_id INTEGER,
                 name  VARCHAR(20),
                 address VARCHAR(20),
                 city VARCHAR(20),
                 state VARCHAR(5),
                 zip_code INTEGER)
  LANGUAGE OLEDB
  EXTERNAL NAME 'spirit!demo.customer';
```

3. Register an OLE DB table function, which retrieves information about stores from a MS SQL Server 7.0 database. The connection string is provided in the external name. The table function has an input parameter to pass through command text to the OLE DB provider. The rowset name does not need to be specified in the external name. The query example passes in SQL statement text to retrieve the top three stores.

```
CREATE FUNCTION favorites (varchar(600))
  RETURNS TABLE (store_id CHAR (4),
                 name  VARCHAR (41),
                 sales INTEGER)
  SPECIFIC favorites
  LANGUAGE OLEDB
```

```
    EXTERNAL NAME '!!Provider=SQLOLEDB.1;Persist Security Info=False;
                   User ID=sa;Initial Catalog=pubs;Data Source=WALTZ;
                   Locale Identifier=1033;Use Procedure for Prepare=1;
                   Auto Translate=False;Packet Size=4096;Workstation ID=WALTZ;
                   OLE DB Services=CLIENTCURSOR;';

  SELECT *
    FROM TABLE (favorites
               (' select top 3 sales.stor_id as store_id, ' CONCAT
                    ' stores.stor_name as name, ' CONCAT
                    ' sum(sales. qty) as sales ' CONCAT
                ' from sales, stores ' CONCAT
                ' where sales.stor_id = stores.stor_id ' CONCAT
                ' group by sales.stor_id, stores.stor_name ' CONCAT
                ' order by sum(sales.qty) desc ')) as f;
```

CREATE FUNCTION (sourced or template)

The CREATE FUNCTION (Sourced or Template) statement is used to register a function or function template with a server.

This statement can register the following objects:

- A user-defined function, based on another existing scalar or aggregate function, at the current server.
- A function template with an application server that is designated as a federated server. A *function template* is a partial function that contains no executable code. The user creates it for the purpose of mapping it to a data source function. After the mapping is created, the user can specify the function template in queries submitted to the federated server. When such a query is processed, the federated server will invoke the data source function to which the template is mapped, and return values whose data types correspond to those in the RETURNS portion of the template's definition.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

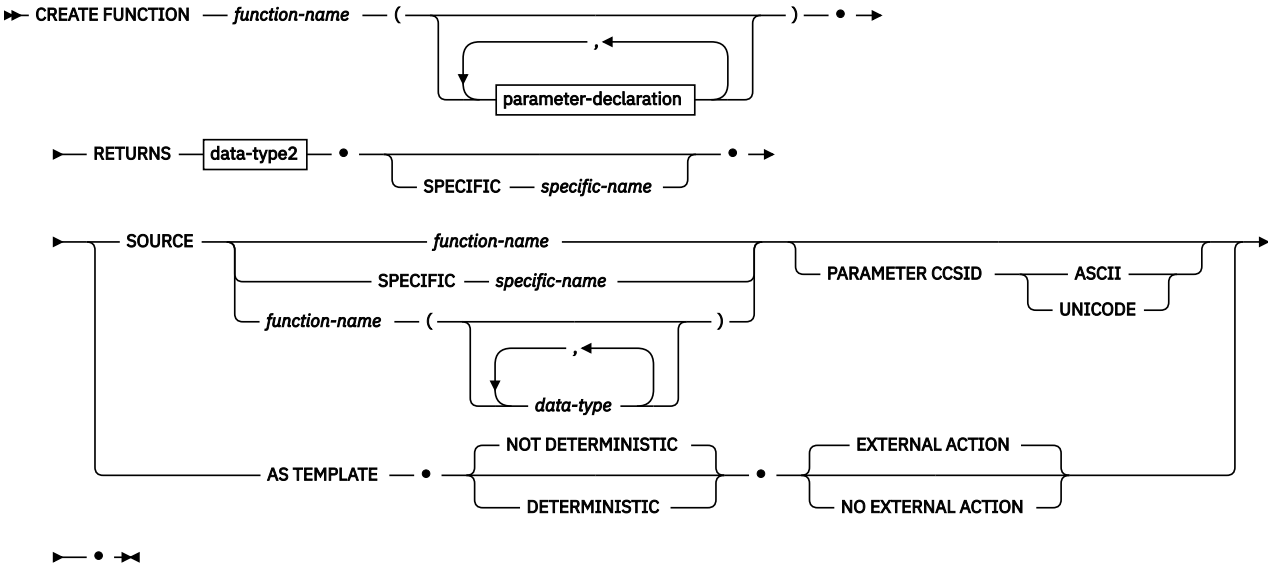
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the function does not exist
- CREATEIN privilege on the schema, if the schema name of the function exists
- SCHEMAADM authority on the schema, if the schema name of the function exists
- DBADM authority

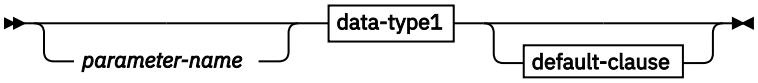
The privileges held by the authorization ID of the statement must also include EXECUTE privilege on the source function or EXECUTEIN privilege on the schema containing the source function if the authorization ID of the statement does not have DATAACCESS authority on the database or DATAACCESS authority on the schema containing the source function and the SOURCE clause is specified.

Group privileges are not considered for any table or view specified in the CREATE FUNCTION statement.

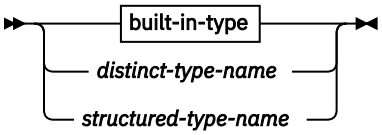
Syntax



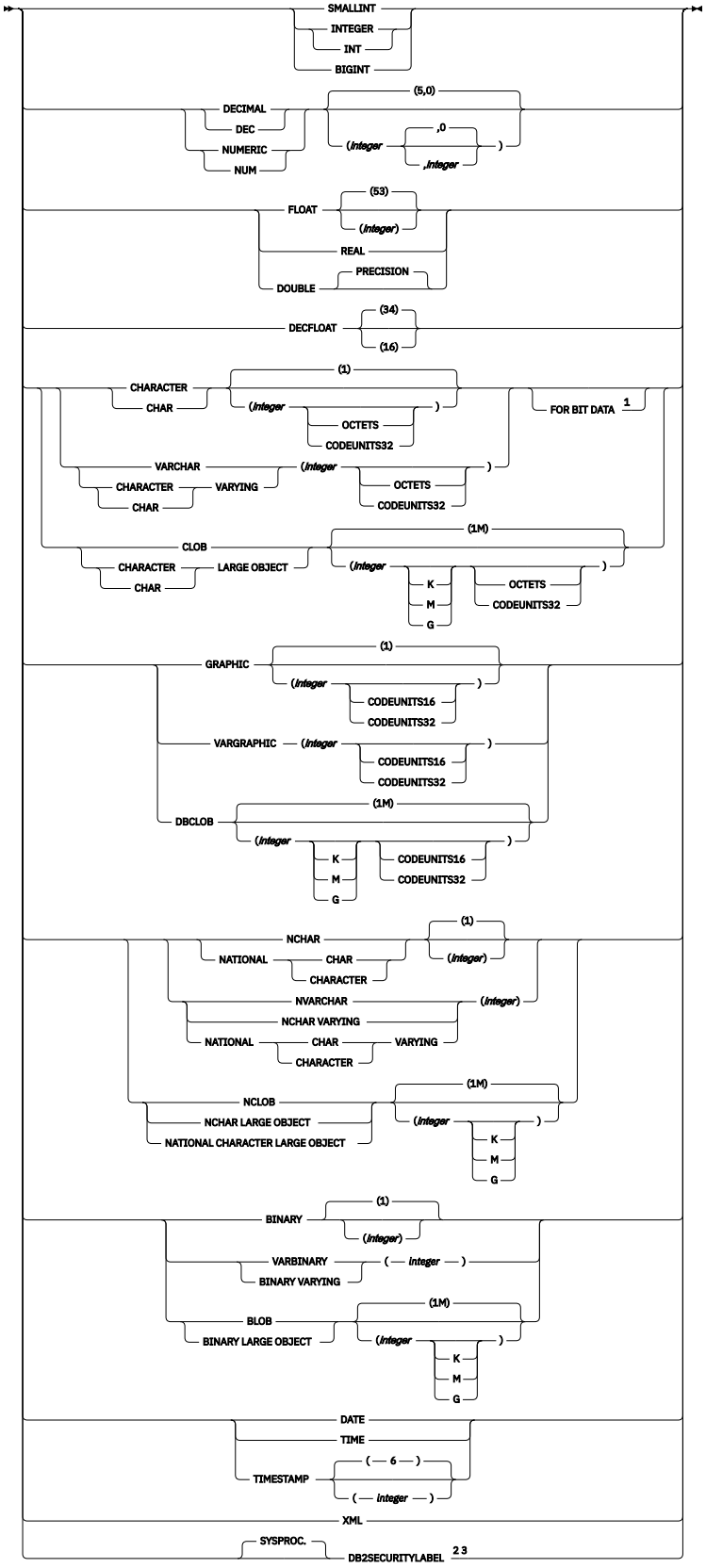
parameter-declaration



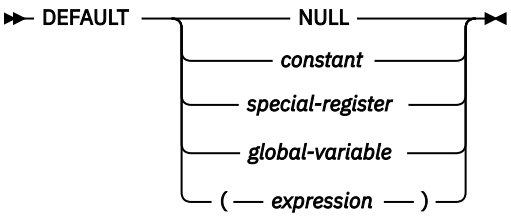
data-type1, data-type2



built-in-type



default-clause



Notes:

[1] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).
[2] DB2SECURITYLABEL is the built-in distinct type that must be used to define the row security label column of a protected table.
[3] For a column of type DB2SECURITYLABEL, NOT NULL WITH DEFAULT is implicit and cannot be explicitly specified (SQLSTATE 42842). The default value for a column of type DB2SECURITYLABEL is the session authorization ID's security label for write access.

Description

function-name
Names the function or function template being defined. It is a qualified or unqualified name that designates a function. The unqualified form of *function-name* is an SQL identifier. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a *schema-name* followed by a period and an SQL identifier.

The name, including the implicit or explicit qualifiers, together with the number of parameters and the data type of each parameter (without regard for any length, precision or scale attributes of the data type) must not identify a function or function template described in the catalog (SQLSTATE 42723). The unqualified name, together with the number and data types of the parameters, while of course unique within its schema, need not be unique across schemas.

If a two-part name is specified, the *schema-name* cannot begin with 'SYS' (SQLSTATE 42939).

A number of names used as keywords in predicates are reserved for system use, and cannot be used as a *function-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators.

When naming a user-defined function that is sourced on an existing function with the purpose of supporting the same function with a user-defined distinct type, the same name as the sourced function may be used. This allows users to use the same function with a user-defined distinct type without realizing that an additional definition was required. In general, the same name can be used for more than one function if there is some difference in the signature of the functions.

(parameter-declaration,...)
Identifies the number of input parameters of the function or function template, and specifies the data type and optional default value of each parameter. One entry in the list must be specified for each parameter that the function or function template will expect to receive. No more than 90 parameters are allowed (SQLSTATE 54023).

It is possible to register a function that has no parameters. In this case, the parentheses must still be coded, with no intervening data types. For example:

```
CREATE FUNCTION WOOFER() ...
```

No two identically-named functions within a schema are permitted to have exactly the same type for all corresponding parameters. This restriction also applies to a function and function template with the same name within the same schema. Lengths, precisions, and scales are not considered in this type comparison. Therefore, CHAR(8) and CHAR(35) are considered to be the same type, as

are DECIMAL(11,2) and DECIMAL (4,3). A weakly typed distinct type specified for a parameter is considered to be the same data type as the source type of the distinct type. For a Unicode database, CHAR(13) and GRAPHIC(8) are considered to be the same type. There is some further bundling of types that causes them to be treated as the same type for this purpose, such as DECIMAL and NUMERIC. A duplicate signature returns an error (SQLSTATE 42723).

parameter-name
Specifies an optional name for the input parameter. The name cannot be the same as any other *parameter-name* in the parameter list (SQLSTATE 42734).

data-type1
Specifies the data type of the input parameter. The data type can be a built-in data type, a distinct type, or a structured type.

Any valid SQL data type can be used if it is castable to the type of the corresponding parameter of the function identified in the SOURCE clause (for information, see "Casting between data types"). However, this checking does not guarantee that an error will not occur when the function is invoked.

For a more complete description of each built-in data type, see "CREATE TABLE".

- A datetime type parameter is passed as a character data type, and the data is passed in the ISO format.
- Array types cannot be specified (SQLSTATE 42879).
- A reference type specified as REF(*type-name*) cannot be specified (SQLSTATE 42879).

For a user-defined distinct type, the length, precision, or scale attributes for the parameter are those of the source type of the distinct type (those specified on CREATE TYPE). A distinct type parameter is passed as the source type of the distinct type. If the name of the distinct type is unqualified, the database manager resolves the schema name by searching the schemas in the SQL path.

For a user-defined structured type, the appropriate transform functions must exist in the associated transform group.

Because the function is sourced, it is not necessary (but still permitted) to specify length, precision, or scale for the parameterized data types. Empty parentheses can be used instead; for example, CHAR(). A *parameterized data type* is any one of the data types that can be defined with a specific length, scale, or precision. The parameterized data types are the string data types, the decimal data types, and the TIMESTAMP data type.

With a function template, empty parentheses can also be used instead of specifying length, precision, or scale for the parameterized data types. It is recommended to use empty parentheses for the parameterized data types. If you use empty parentheses, the length, precision, or scale is the same as that of the remote function, which is determined when the function template is mapped to a remote function by creating a function mapping. If you omit parentheses altogether, the default length for the data type is used (see "CREATE TABLE").

DEFAULT
Specifies a default value for the parameter. The default can be a constant, a special register, a global variable, an expression, or the keyword NULL. The special registers that can be specified as the default are that same as those that can be specified for a column default (see *default-clause* in the CREATE TABLE statement). Other special registers can be specified as the default by using an expression.

The expression can be any expression of the type described in "Expressions". If a default value is not specified, the parameter has no default and the corresponding argument cannot be omitted on invocation of the procedure. The maximum size of the *expression* is 64K bytes.

The default expression must not modify SQL data (SQLSTATE 428FL or SQLSTATE 429BL). The expression must be assignment compatible to the parameter data type (SQLSTATE 42821).

A default cannot be specified for a parameter of type ARRAY, ROW, or CURSOR (SQLSTATE 429BB).

RETURNS

This mandatory clause identifies the output of the function or function template.

data-type2

Specifies the data type of the output.

With a sourced scalar function, any valid SQL data type is acceptable, as is a distinct type, provided it is castable from the result type of the source function. An array type cannot be specified as the data type of a parameter (SQLSTATE 42879).

The parameter of a parameterized type need not be specified for parameters of a sourced function. Instead, empty parentheses can be used; for example, VARCHAR().

For additional considerations and rules that apply to the specification of the data type in the RETURNS clause when the function is sourced on another, see the "Rules" section of this statement.

With a function template, empty parentheses are not allowed (SQLSTATE 42611). Length, precision, or scale must be specified for the parameterized data types. It is recommended to specify the same length, precision, or scale as that of the remote function.

built-in-type

See "CREATE TABLE" for the description of built-in data types.

SPECIFIC *specific-name*

Provides a unique name for the instance of the function that is being defined. This specific name can be used when sourcing on this function, dropping the function, or commenting on the function. It can never be used to invoke the function. The unqualified form of *specific-name* is an SQL identifier. The qualified form is a *schema-name* followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another function instance that exists at the application server; otherwise an error (SQLSTATE 42710) is returned.

The *specific-name* may be the same as an existing *function-name*.

If no qualifier is specified, the qualifier that was used for *function-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *function-name* or an error (SQLSTATE 42882) is returned.

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmssxxx.

SOURCE

Specifies that the new function is being defined as a sourced function. A *sourced function* is implemented by another function (the *source function*). The function must be a scalar or aggregate function that exists at the current server, and it must be one of the following types of functions:

- A function that was defined with a CREATE FUNCTION statement
- A cast function that was generated by a CREATE TYPE statement
- A built-in function

If the source function is not a built-in function, the particular function can be identified by its name, function signature, or specific name.

If the source function is a built-in function, the SOURCE clause must include a function signature for the built-in function. The source function must not be any of the following built-in functions (If a particular syntax is indicated, only the indicated form cannot be specified.):

- CARDINALITY
- CHAR when more than one parameter is specified and the first parameter is a datetime data type
- CHARACTER_LENGTH with the string units parameter
- COALESCE
- CONTAINS
- CURSOR_ROWCOUNT

- DATAPARTITIONNUM
- DBPARTITIONNUM
- DEREF
- EXTRACT
- GRAPHIC when more than one parameter is specified and the first parameter is a datetime data type
- GREATEST
- HASHEDVALUE
- INSERT with the string units parameter
- INSTR with the string units parameter
- LCASE with the string units parameter
- LEAST
- LEFT with the string units parameter
- LENGTH with the string units parameter
- LOCATE with the string units parameter
- LOCATE_IN_STRING with the string units parameter
- LOWER with the string units parameter
- MAX
- MAX_CARDINALITY
- MIN
- NODENUMBER
- NULLIF
- NVL
- OVERLAY with the string units parameter
- PARAMETER
- POSITION with the string units parameter
- RAISE_ERROR
- REC2XML
- RID
- RID_BIT
- RIGHT with the string units parameter
- SCORE
- STRIP
- SUBSTRING with the string units parameter
- TRIM
- TRIM_ARRAY
- TYPE_ID
- TYPE_NAME
- TYPE_SCHEMA
- UCASE with the string units parameter
- UPPER with the string units parameter
- VALUE
- VARCHAR when more than one parameter is specified and the first parameter is a datetime data type

- VARGRAPHIC when more than one parameter is specified and the first parameter is a datetime data type
- XMLATTRIBUTES
- XMLCOMMENT
- XMLCONCAT
- XMLDOCUMENT
- XMLELEMENT
- XMLFOREST
- XMLNAMESPACES
- XMLPARSE
- XMLPI
- XMLQUERY
- XMLROW
- XMLSERIALIZE
- XMLTEXT
- XMLVALIDATE
- XMLXSROBJECTID
- XSLTRANSFORM

function-name

Identifies the function to use as the source. Valid only if this name is unique within the schema and the authorization ID has DATAACCESS authority on the schema, EXECUTEIN privilege on the schema, or EXECUTE privilege on the function. This syntax variant is not valid for a source function that is a built-in function.

If you provide an unqualified name, the SQL path will be used to locate the function. This is the value of the CURRENT PATH special register. The first schema in the SQL path that has this function name and whose authorization ID has DATAACCESS authority on the schema, EXECUTEIN privilege on the schema, or EXECUTE privilege on the function is selected.

The database will return error SQLSTATE 42704 for each of the following cases:

- no function by this name exists in the named schema
- the name is not valid
- there is no function with this name in the SQL path

The database will return error SQLSTATE 42725 if there is more than one authorized instance of the function in the named or located schema.

The database will return error SQLSTATE 42501 if a function with this name exists but the authorization ID of the statement does not have EXECUTE privilege, or EXECUTEIN privilege, or DATAACCESS authority on the schema of the function.

SPECIFIC *specific-name*

Identifies the particular user-defined function that is to be used as the source, by the *specific-name* either specified or defaulted to at function creation time. This syntax variant is not valid for a source function that is a built-in function.

If an unqualified name is provided, the current SQL path is used to locate the function. The first schema in the SQL path that has a function with this specific name for which the authorization ID of the statement has EXECUTE privilege or EXECUTEIN privilege or DATAACCESS authority on the schema is selected.

If no function by this *specific-name* exists in the named schema or if the name is not qualified and there is no function with this *specific-name* in the SQL path, an error (SQLSTATE 42704) is returned. If a function by this *specific-name* exists, and the authorization ID of the statement does

not have EXECUTE privilege on this function or EXECUTEIN privilege or DATAACCESS authority on the schema, an error (SQLSTATE 42501) is returned.

function-name (data-type,...)
Provides the function signature, which uniquely identifies the source function. This is the only valid syntax variant for a source function that is a built-in function.

The rules for function resolution are applied to select one function from the functions with the same function name, given the data types specified in the SOURCE clause. However, the data type of each parameter in the function selected must have the exact same type as the corresponding data type specified in the source function.

function-name
Gives the function name of the source function. If an unqualified name is provided, then the schemas of the user's SQL path are considered.

data-type
Must match the data type that was specified on the CREATE FUNCTION statement in the corresponding position (comma separated).

It is not necessary to specify the length, precision or scale for the parameterized data types. Instead an empty set of parentheses may be coded to indicate that these attributes are to be ignored when looking for a data type match. For example, DECIMAL() will match a parameter whose data type was defined as DECIMAL(7,2)).

FLOAT() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (REAL or DOUBLE).

However, if length, precision, or scale is coded, the value must exactly match that specified in the CREATE FUNCTION statement. This can be useful in assuring that the intended function will be used. Note also that synonyms for data types will be considered a match (for example DEC and NUMERIC will match).

A type of FLOAT(n) does not need to match the defined value for n, because 0<n<25 means REAL and 24<n<54 means DOUBLE. Matching occurs based on whether the type is REAL or DOUBLE.

If no function with the specified signature exists in the named or implied schema, an error (SQLSTATE 42883) is returned.

PARAMETER CCSID
Specifies the encoding scheme to use for all string data passed into and out of the function. If the PARAMETER CCSID clause is not specified, the default is PARAMETER CCSID UNICODE for Unicode databases, and PARAMETER CCSID ASCII for all other databases.

ASCII
Specifies that string data is encoded in the database code page. If the database is a Unicode database, PARAMETER CCSID ASCII cannot be specified (SQLSTATE 56031). When the function is invoked, the application code page for the function is the database code page.

UNICODE
Specifies that string data is encoded in Unicode. If the database is a Unicode database, character data is in UTF-8, and graphic data is in UCS-2. If the database is not a Unicode database, character data is in UTF-8. In either case, when the function is invoked, the application code page for the function is 1208.

The PARAMETER CCSID clause must specify the same encoding scheme as the source function (SQLSTATE 53090).

AS TEMPLATE
Indicates that this statement will be used to create a function template, not a function with executable code.

NOT DETERMINISTIC or DETERMINISTIC
Specifies whether the function returns the same results for identical input arguments. The default is NOT DETERMINISTIC.

NOT DETERMINISTIC
Specifies that the function might not return the same result each time that the function is invoked with the same input arguments. The function depends on some state values that affect the results. The database manager uses this information during optimization of SQL statements. An example of a function that is not deterministic is one that generates random numbers.

A function that is not deterministic might receive incorrect results if it is executed by parallel tasks.

DETERMINISTIC
Specifies that the function always returns the same result each time that the function is invoked with the same input arguments. The database manager uses this information during optimization of SQL statements. An example of a function that is deterministic is one that calculates the square root of the input argument.

EXTERNAL ACTION or NO EXTERNAL ACTION
Specifies whether the function takes an action that changes the state of an object that the database manager does not manage. An example of an external action is sending a message or writing a record to a file. The default is EXTERNAL ACTION.

EXTERNAL ACTION
Specifies that the function takes an action that changes the state of an object that the database manager does not manage. EXTERNAL ACTION must be implicitly or explicitly specified if the SQL routine body invokes a function that is defined with EXTERNAL ACTION (SQLSTATE 428C2).

A function with external actions might return incorrect results if the function is executed by parallel tasks. For example, if the function sends a note for each initial call to it, one note is sent for each parallel task instead of once for the function.

NO EXTERNAL ACTION
Specifies that the function does not take any action that changes the state of an object that the database manager does not manage. The database manager uses this information during optimization of SQL statements.

Rules

- For convenience, in this section the function being created will be called CF and the function identified in the SOURCE clause will be called SF, no matter which of the three allowable syntaxes was used to identify SF.
  - The unqualified name of CF and the unqualified name of SF can be different.
  - A function named as the source of another function can, itself, use another function as its source. Extreme care should be exercised when exploiting this facility, because it could be very difficult to debug an application if an indirectly invoked function returns an error.
  - The following clauses are invalid if specified in conjunction with the SOURCE clause (because CF will inherit these attributes from SF):
    - CAST FROM ...,
    - EXTERNAL ...,
    - LANGUAGE ...,
    - PARAMETER STYLE ...,
    - DETERMINISTIC / NOT DETERMINISTIC,
    - FENCED / NOT FENCED,
    - RETURNS NULL ON NULL INPUT / CALLED ON NULL INPUT
    - EXTERNAL ACTION / NO EXTERNAL ACTION
    - NO SQL / CONTAINS SQL / READS SQL DATA

- SCRATCHPAD / NO SCRATCHPAD
- FINAL CALL / NO FINAL CALL
- RETURNS TABLE (...)
- CARDINALITY ...
- ALLOW PARALLEL / DISALLOW PARALLEL
- DBINFO / NO DBINFO
- THREADSAFE / NOT THREADSAFE
- INHERIT SPECIAL REGISTERS

An error (SQLSTATE 42613) will result from violation of these rules.

- The number of input parameters in CF must be the same as those in SF; otherwise an error (SQLSTATE 42624) is returned.
- It is not necessary for CF to specify length, precision, or scale for a parameterized data type in the case of:
  - The function's input parameters,
  - Its RETURNS parameter

  Instead, empty parentheses may be specified as part of the data type (for example: VARCHAR()) in order to indicate that the length/precision/scale will be the same as those of the source function, or determined by the casting.

  However, if length, precision, or scale is specified then the value in CF is checked against the corresponding value in SF as outlined in the remaining rules for input parameters and returns value.
- The specification of the input parameters of CF are checked against those of SF. The data type of each parameter of CF must either be the same as or be *castable* to the data type of the corresponding parameter of SF. If any parameter is not the same type or castable, an error (SQLSTATE 42879) is returned.

  Note that this rule provides no guarantee against an error occurring when CF is used. An argument that matches the data type and length or precision attributes of a CF parameter may not be assignable if the corresponding SF parameter has a shorter length or less precision. In general, parameters of CF should not have length or precision attributes that are greater than the attributes of the corresponding SF parameters.
- The specifications for the RETURNS data type of CF are checked against that of SF. The final RETURNS data type of SF, after any casting, must either be the same as or castable to the RETURNS data type of CF. Otherwise an error (SQLSTATE 42866) is returned.

  Note that this rule provides no guarantee against an error occurring when CF is used. A result value that matches the data type and length or precision attributes of the SF RETURNS data type may not be assignable if the CF RETURNS data type has a shorter length or less precision. Caution should be used when choosing to specify the RETURNS data type of CF as having length or precision attributes that are less than the attributes of the SF RETURNS data type.
- Revalidation of CF that does not have a parameter with a default expression is not supported (SQLSTATE 42997).

Notes

- Determining whether one data type is castable to another data type does not consider length or precision and scale for parameterized data types such as CHAR and DECIMAL. Therefore, errors may occur when using a function as a result of attempting to cast a value of the source data type to a value of the target data type. For example, VARCHAR is castable to DATE but if the source type is actually defined as VARCHAR(5), an error will occur when using the function.
- When choosing the data types for the parameters of a user-defined function, consider the rules for promotion that will affect its input values (see "Promotion of data types"). For example, a constant which may be used as an input value could have a built-in data type different from the one expected

and, more significantly, may not be promoted to the data type expected. Based on the rules for promotion, it is generally recommended to use the following data types for parameters:

  - INTEGER instead of SMALLINT
  - DOUBLE instead of REAL
  - VARCHAR instead of CHAR
  - VARGRAPHIC instead of GRAPHIC
- Creating a function with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- For a federated server to recognize a data source function, the function must map to a counterpart at the federated database. If the database contains no counterpart, the user must create the counterpart and then the mapping.

  The counterpart can be a function (scalar or source) or a function template. If the user creates a function and the required mapping, then, each time a query that specifies the function is processed, the database manager (1) compares strategies for invoking it with strategies for invoking the data source function, and (2) invokes the function that is expected to require less overhead.

  If the user creates a function template and the mapping, then each time a query that specifies the template is processed, the database manager invokes the data source function that it maps to, provided that an access plan for invoking this function exists.
- ***Privileges***: The definer of a function always receives the EXECUTE privilege on the function, as well as the right to drop the function. The definer of the function is also given the WITH GRANT OPTION if any of the following conditions apply:
  - The source function is a built-in function.
  - The definer of the function has EXECUTE WITH GRANT OPTION on the source function.
  - The definer of the function has EXECUTEIN WITH GRANT OPTION on the schema containing the source function.
  - The function is a template.
- ***EXTERNAL ACTION functions***: If an EXTERNAL ACTION function is invoked in other than the outermost select list, the results are unpredictable since the number of times the function is invoked will vary depending on the access plan used.
- ***Setting of the default value***: Parameters of a function that are defined with a default value are set to their default value when the functions is invoked, but only if a value is not supplied for the corresponding argument, or is specified as DEFAULT, when the function is invoked.
- **Create function mapping to table or row functions:** A create function mapping to remote functions that returns a table or a row is not supported in a federated database.
- **Inheriting SECURED or NOT SECURED attributes from the source function:** The sourced user-defined function inherits the SECURED or NOT SECURED attribute from the source function in which only the topmost user-defined function is considered. If the topmost user-defined function is secure, any nested user-defined functions are considered secure. The database manager does not validate whether those nested user-defined functions are secure. If those nested functions can access sensitive data, the user with SECADM authority needs to ensure those functions are allowed to access those data and that a change control audit procedure has been established for all changes to those functions.

Examples

- *Example 1:* Some time after the creation of Pellow's original CENTER external scalar function, another user wants to create a function based on it, except this function is intended to accept only integer arguments.

```
CREATE FUNCTION MYCENTER (INTEGER, INTEGER)
  RETURNS FLOAT
  SOURCE PELLOW.CENTER (INTEGER, FLOAT)
```

- *Example 2:* A distinct type, HATSIZE, has been created based on the built-in INTEGER data type. It would be useful to have an AVG function to compute the average hat size of different departments. This is easily done as follows:

```
CREATE FUNCTION AVG (HATSIZE) RETURNS HATSIZE
  SOURCE SYSIBM.AVG (INTEGER)
```

  The creation of the distinct type has generated the required cast function, allowing the cast from HATSIZE to INTEGER for the argument and from INTEGER to HATSIZE for the result of the function.
- *Example 3:* In a federated system, a user wants to invoke an Oracle UDF that returns table statistics in the form of values with double-precision floating points. The federated server can recognize this function only if there is a mapping between the function and a federated database counterpart. But no such counterpart exists. The user decides to provide one in the form of a function template, and to assign this template to a schema called NOVA. The user uses the following code to register the template with the federated server.

```
CREATE FUNCTION NOVA.STATS (DOUBLE, DOUBLE)
  RETURNS DOUBLE
  AS TEMPLATE DETERMINISTIC NO EXTERNAL ACTION
```

- *Example 4:* In a federated system, a user wants to invoke an Oracle UDF that returns the dollar amounts that employees of a particular organization earn as bonuses. The federated server can recognize this function only if there is a mapping between the function and a federated database counterpart. No such counterpart exists; thus, the user creates one in the form of a function template. The user uses the following code to register this template with the federated server.

```
CREATE FUNCTION BONUS ()
  RETURNS DECIMAL (8,2)
  AS TEMPLATE DETERMINISTIC NO EXTERNAL ACTION
```

CREATE FUNCTION (SQL scalar, table, or row)

The CREATE FUNCTION (SQL scalar, table, or row) statement is used to define a user-defined SQL scalar, table, or row function.

A *scalar function* returns a single value each time it is invoked, and is generally valid wherever an SQL expression is valid. A *table function* can be used in a FROM clause and returns a table. A *row function* can be used as a transform function and returns a row.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the function does not exist
- CREATEIN privilege on the schema, if the schema name of the function refers to an existing schema
- SCHEMAADM authority on the schema, if the schema name of the function exists
- DBADM authority

and at least one of the following authorities on each table, view, or nickname identified in any fullselect:

- CONTROL privilege on that table, view, or nickname
- SELECT privilege on that table, view, or nickname

- SELECTIN privilege on the schema containing the table, view, or nickname
- DATAACCESS authority on the schema containing the table, view, or nickname
- DATAACCESS authority on the database

Group privileges other than PUBLIC are not considered for any table or view specified in the CREATE FUNCTION statement.

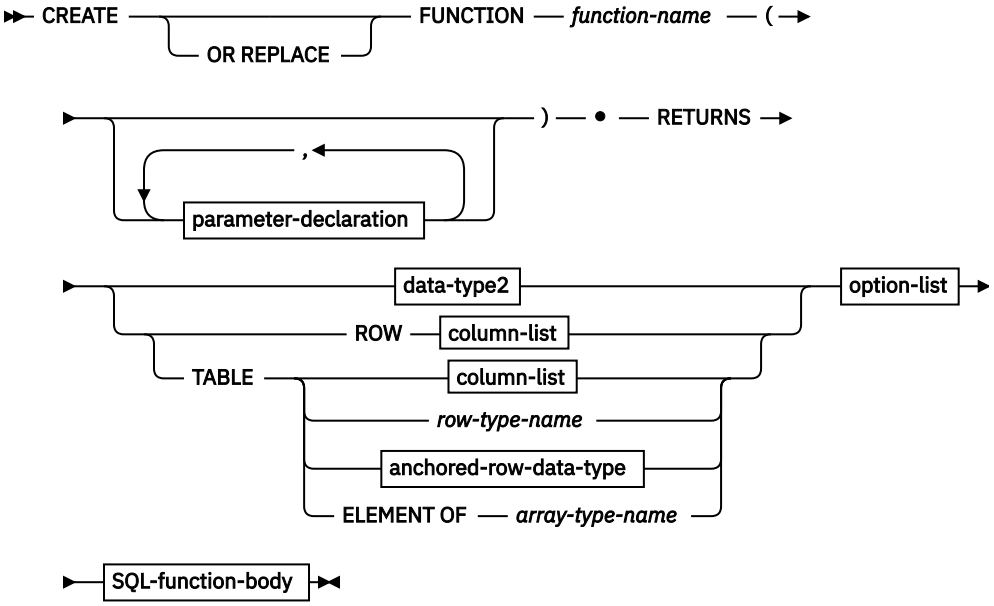
Authorization requirements of the data source for the table or view referenced by the nickname are applied when the function is invoked. The authorization ID of the connection can be mapped to a different remote authorization ID.

The privileges held by the authorization ID of the statement must also include all of the privileges necessary to invoke the SQL statements that are specified in the function body.

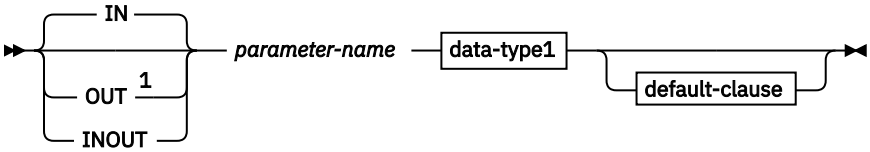
To replace an existing function, the authorization ID of the statement must be the owner of the existing function (SQLSTATE 42501).

If the SECURED option is specified, the authorization ID of the statement must include SECADM or CREATE_SECURE_OBJECT authority (SQLSTATE 42501).

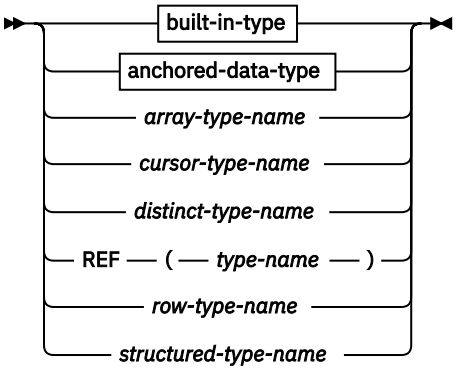
Syntax



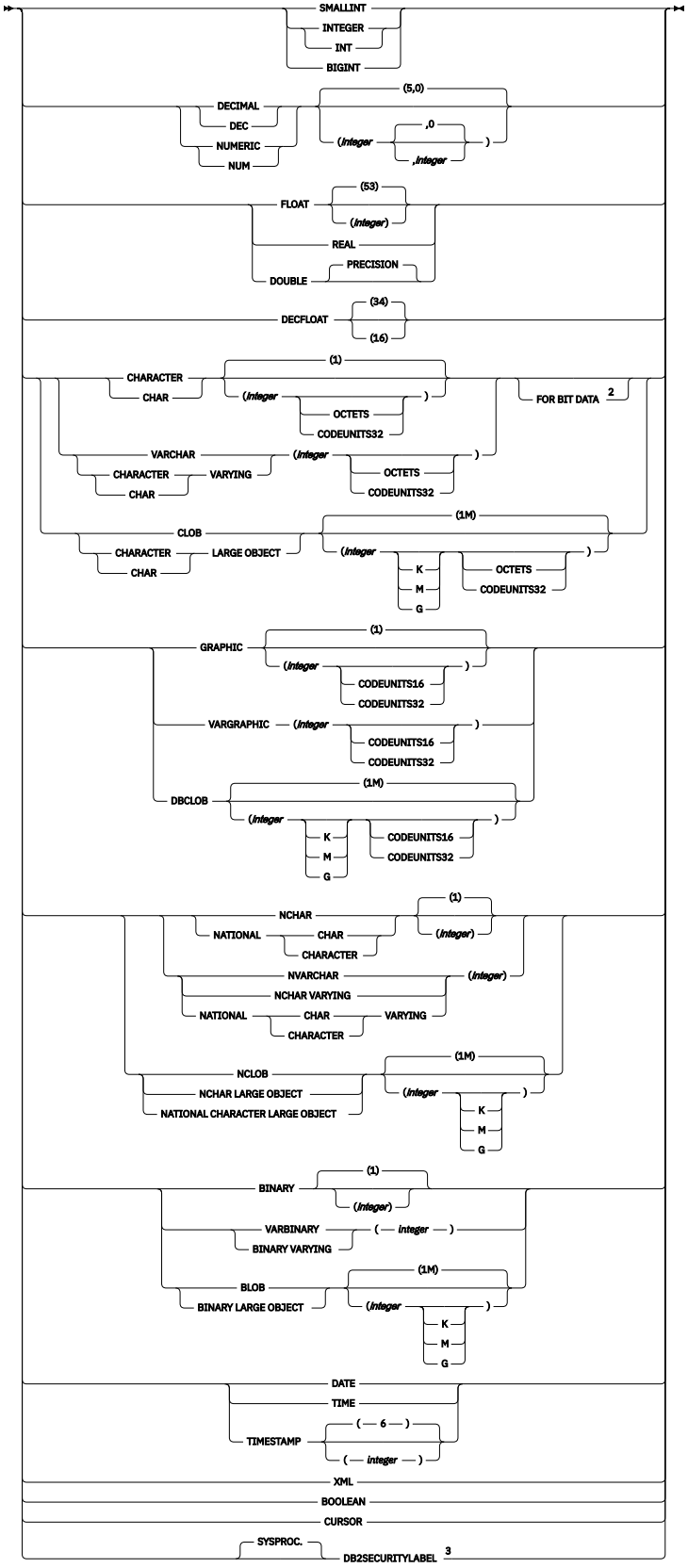
parameter-declaration



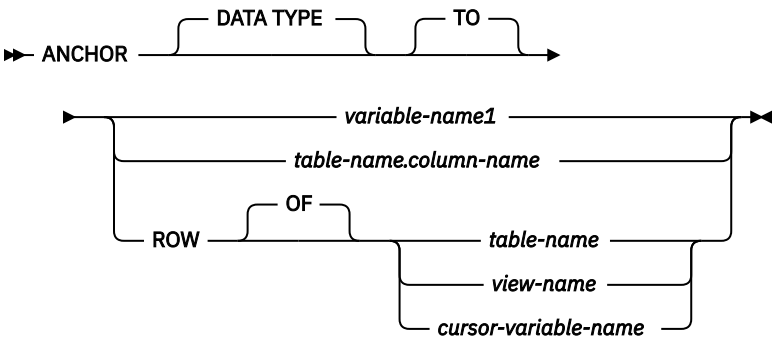
data-type1, data-type2



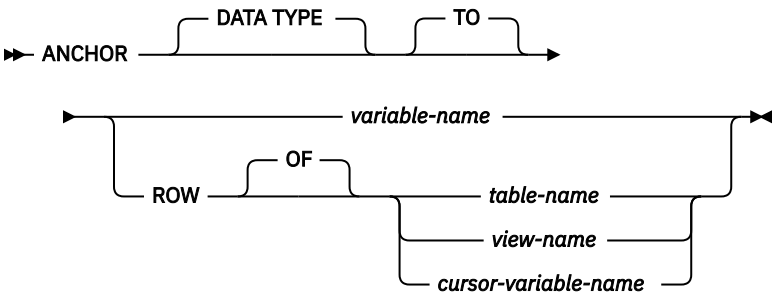
built-in type



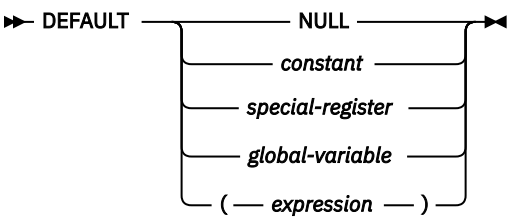
anchored-data-type



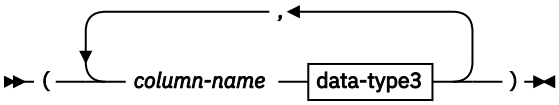
anchored-row-data-type



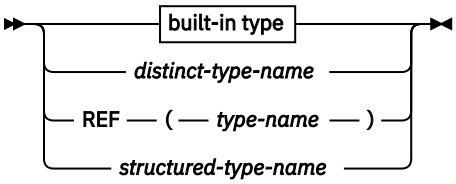
default-clause



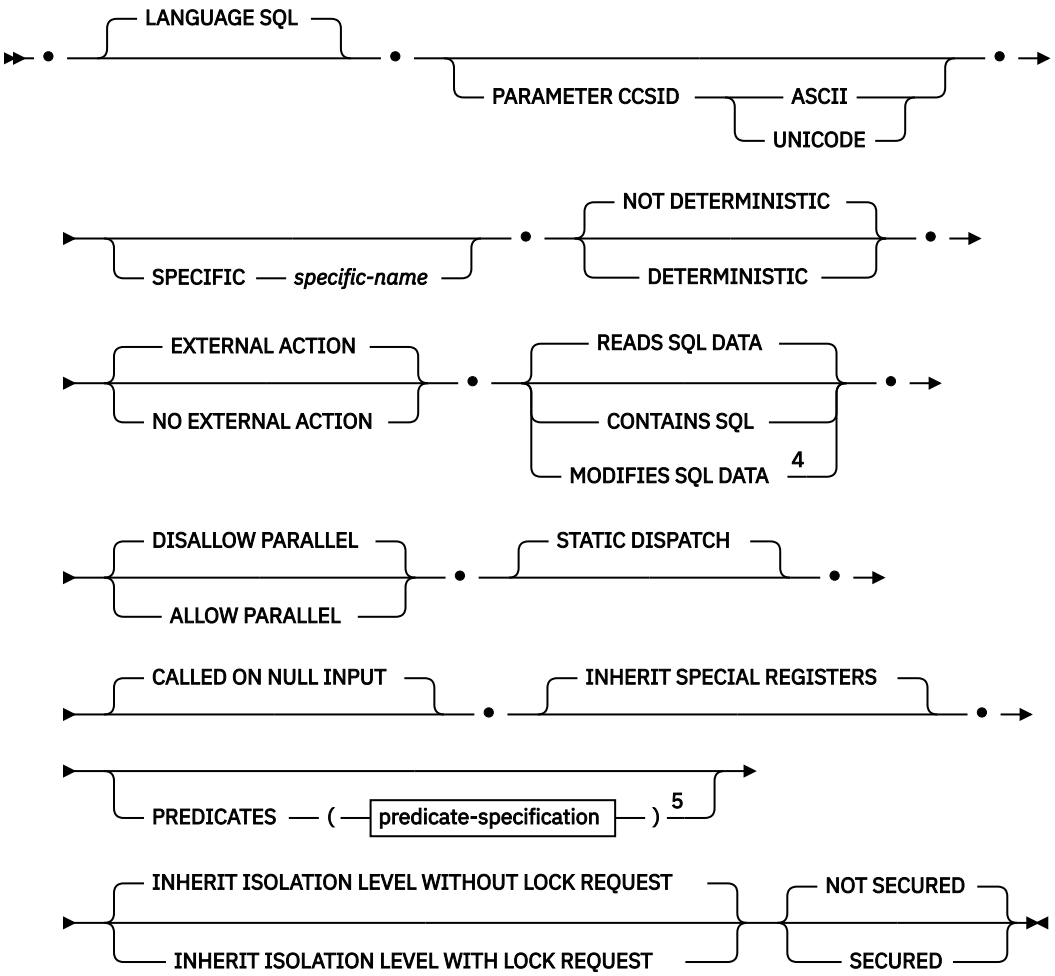
column-list



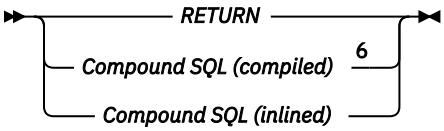
data-type3



option-list



SQL-function-body



Notes:

[1] OUT and INOUT are valid only if RETURNS specifies a scalar result and the SQL-function-body is a compound SQL (compiled) statement.

[2] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

[3] DB2SECURITYLABEL is the built-in distinct type that must be used to define the row security label column of a protected table.

[4] Valid only for compiled scalar function definition and an inlined table function definition. A compiled scalar function defined as MODIFIES SQL DATA can only be used as the only element on the right side of an assignment statement that is within a compound SQL (compiled) statement.

[5] Valid only if RETURNS specifies a scalar result (*data-type2*)

[6] The following apply to the specification of a compound SQL (compiled) statement: a) Must be used if the parameter data types or returned data types include a row type, array type, or cursor type; b) Must be used if the RETURNS TABLE clause specifies any syntax other than a column-list; c) Not supported if RETURNS ROW is specified; d) Not supported when defining a table function in a partitioned database environment.

Description

OR REPLACE

Specifies to replace the definition for the function if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog, with the exception that privileges that were granted on the function are not affected. This option can be specified only by the owner of the object. This option is ignored if a definition for the function does not exist at the current server. To replace an existing function, the specific name and function name of the new definition must be the same as the specific name and function name of the old definition, or the signature of the new definition must match the signature of the old definition. Otherwise, a new function is created.

If the function is referenced in the definition of a row permission or a column mask, the function cannot be replaced (SQLSTATE 42893).

function-name

Names the function being defined. It is a qualified or unqualified name that designates a function. The unqualified form of *function-name* is an SQL identifier. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a *schema-name* followed by a period and an SQL identifier.

The name, including the implicit or explicit qualifiers, together with the number of parameters and the data type of each parameter (without regard for any length, precision or scale attributes of the data type) must not identify a function described in the catalog (SQLSTATE 42723). The unqualified name, together with the number and data types of the parameters, while of course unique within its schema, need not be unique across schemas.

If a two-part name is specified, the *schema-name* cannot begin with 'SYS' (SQLSTATE 42939).

A number of names used as keywords in predicates are reserved for system use, and cannot be used as a *function-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators.

The same name can be used for more than one function if there is some difference in the signature of the functions. Although there is no prohibition against it, an external user-defined table function should not be given the same name as a built-in function.

(*parameter-declaration*, ...)

Identifies the number of input parameters of the function, and specifies the mode, name, data type, and optional default value of each parameter. One entry in the list must be specified for each parameter that the function will expect to receive. No more than 90 parameters are allowed (SQLSTATE 54023).

It is possible to register a function that has no parameters. In this case, the parentheses must still be coded, with no intervening data types. For example:

```
CREATE FUNCTION WOOFER() ...
```

No two identically-named functions within a schema are permitted to have exactly the same type for all corresponding parameters. Lengths, precisions, and scales are not considered in this type comparison. Therefore, CHAR(8) and CHAR(35) are considered to be the same type, as are DECIMAL(11,2) and DECIMAL (4,3), as well as DECFLOAT(16) and DECFLOAT(34). A weakly typed distinct type specified for a parameter is considered to be the same data type as the source type of the distinct type. For a Unicode database, CHAR(13) and GRAPHIC(8) are considered to be the same type. There is some further bundling of types that causes them to be treated as the same type for this purpose, such as DECIMAL and NUMERIC. A duplicate signature returns an error (SQLSTATE 42723).

If the data type for a parameter is a Boolean data type, array type, cursor type, or row type, the SQL function body can only reference the parameter within a compound SQL (compiled) statement (SQLSTATE 428H2).

IN | OUT | INOUT
Specifies the mode of the parameter. If an error is returned by the function, OUT parameters are undefined and INOUT parameters are unchanged. The default is IN.

IN
Identifies the parameter as an input parameter to the function. Any changes made to the parameter within the function are not available to the invoking context when control is returned.

OUT
Identifies the parameter as an output parameter for the function.

The function must be a scalar function that is defined with a compound SQL (compiled) statement (SQLSTATE 42613).

The function can be referenced only on the right side of an assignment statement that is in a compound SQL (compiled) statement, and the function reference cannot be part of an expression (SQLSTATE 42887).

INOUT
Identifies the parameter as both an input and output parameter for the function.

The function must be a scalar function that is defined with a compound SQL (compiled) statement (SQLSTATE 42613).

The function can be referenced only on the right side of an assignment statement that is in a compound SQL (compiled) statement, and the function reference cannot be part of an expression (SQLSTATE 42887).

parameter-name
Specifies a name for the parameter. The name cannot be the same as any other *parameter-name* in the parameter list (SQLSTATE 42734).

data-type1
Specifies the data type of the parameter.

built-in-type
Specifies a built-in data type. For a more complete description of each built-in data type except BOOLEAN and CURSOR, which cannot be specified for a table, see "CREATE TABLE".

BOOLEAN
For a Boolean.

CURSOR
For a reference to an underlying cursor.

anchored-data-type
Identifies another object used to define the parameter data type. The data type of the anchor object can be any of the data types explicitly allowed as *data-type1*. The data type of the anchor object has the same limitations that apply to specifying the data type directly, or in the case of a row, to creating a row type.

ANCHOR DATA TYPE TO
Indicates an anchored data type is used to specify the data type.

variable-name1
Identifies a global variable. The data type of the global variable is used as the data type for *parameter-name*.

table-name.column-name
Identifies a column name of an existing table or view. The data type of the column is used as the data type for *parameter-name*.

ROW OF *table-name* or *view-name*
Specifies a row of fields with names and data types that are based on the column names and column data types of the table identified by *table-name* or the view identified by *view-name*. The data type of *parameter-name* is an unnamed row type.

ROW OF *cursor-variable-name*
Specifies a row of fields with names and data types that are based on the field names and field data types of the cursor variable identified by *cursor-variable-name*. The specified cursor variable must be one of the following elements (SQLSTATE 428HS):

- A global variable with a strongly typed cursor data type
- A global variable with a weakly typed cursor data type that was created or declared with a CONSTANT clause specifying a *select-statement* where all the result columns are named.

If the cursor type of the cursor variable is not strongly typed using a named row type, the data type of *parameter-name* is an unnamed row type.

array-type-name
Specifies the name of a user-defined array type. If *array-type-name* is specified without a schema name, the array type is resolved by searching the schemas in the SQL path.

cursor-type-name
Specifies the name of a cursor type. If *cursor-type-name* is specified without a schema name, the cursor type is resolved by searching the schemas in the SQL path.

distinct-type-name
Specifies the name of a distinct type. The length, precision, and scale of the parameter are, respectively, the length, precision, and scale of the source type of the distinct type. A distinct type parameter is passed as the source type of the distinct type. If *distinct-type-name* is specified without a schema name, the distinct type is resolved by searching the schemas in the SQL path.

REF (*type-name*)
Specifies a reference type without a scope. The specified *type-name* must identify a user-defined structured type (SQLSTATE 428DP). The system does not attempt to infer the scope of the parameter or result. Inside the body of the function, a reference type can be used in a dereference operation only by first casting it to have a scope. Similarly, a reference returned by an SQL function can be used in a dereference operation only by first casting it to have a scope. If a type name is specified without a schema name, the *type-name* is resolved by searching the schemas in the SQL path.

row-type-name
Specifies the name of a user-defined row type. The fields of the parameter are the fields of the row type. If *row-type-name* is specified without a schema name, the row type is resolved by searching the schemas in the SQL path.

structured-type-name
Specifies the name of a user-defined structured type. If *structured-type-name* is specified without a schema name, the structured type is resolved by searching the schemas in the SQL path.

DEFAULT
Specifies a default value for the parameter. The default can be a constant, a special register, a global variable, an expression, or the keyword NULL. The special registers that can be specified as the default are that same as those that can be specified for a column default (see *default-clause* in the CREATE TABLE statement). Other special registers can be specified as the default by using an expression.

The expression can be any expression of the type described in "Expressions". If a default value is not specified, the parameter has no default and the corresponding argument cannot be omitted on invocation of the procedure. The maximum size of the *expression* is 64K bytes.

The default expression must not modify SQL data (SQLSTATE 428FL or SQLSTATE 429BL). The expression must be assignment compatible to the parameter data type (SQLSTATE 42821).

A default cannot be specified in the following situations:

- For INOUT or OUT parameters (SQLSTATE 42601)

- For a parameter of type ARRAY, ROW, or CURSOR (SQLSTATE 429BB)
- For a parameter to a function definition that also specified RETURNS ROW or a PREDICATES clause (SQLSTATE 42613)

RETURNS

This mandatory clause identifies the type of output of the function.

If the data type of the output of the function is a Boolean data type, array type, cursor type, or row type, the SQL function body must be a compound SQL (compiled) statement (SQLSTATE 428H2).

data-type2

Specifies the data type of the output.

In this statement, exactly the same considerations apply as for the parameters of SQL functions described previously in *data-type1* for function parameters.

ROW

Specifies that the output of the function is a single row. If the function returns more than one row, an error is returned (SQLSTATE 21505).

This form of a row function can be used only as a transform function for a structured type (having one structured type as its parameter and returning only built-in data types).

column-list

The list of column names and data types returned for a ROW function. The *column-list* must include at least two columns (SQLSTATE 428F0).

column-name

Specifies the name of this column. The name cannot be qualified and the same name cannot be used for more than one column in the list.

data-type3

Specifies the data type of the column, and can be any data type supported by a parameter of the SQL function.

The same considerations apply as for the parameters of SQL functions described previously in *data-type1* for function parameters. However, *data-type3* does not support *anchored-data-type*, *array-type-name*, *cursor-type-name*, and *row-type-name*.

TABLE

Specifies that the output of the function is a table.

column-list

The list of column names and data types returned for a TABLE function

column-name

Specifies the name of this column. The name cannot be qualified and the same name cannot be used for more than one column in the list.

data-type3

Specifies the data type of the column, and can be any data type supported by a parameter of the SQL function.

The same considerations apply as for the parameters of SQL functions described previously in *data-type1* for function parameters. However, *data-type3* does not support *anchored-data-type*, *array-type-name*, *cursor-type-name*, and *row-type-name*.

row-type-name

Specifies a row type from which the fields are used to derive the column list. The field names of the row type are used as the column names.

anchored-row-data-type

Identifies row information from another object to use as the columns of the returned table.

ANCHOR DATA TYPE TO

Indicates an anchored data type is used to specify the data type.

variable-name
Identifies a global variable. The data type of the referenced variable must be a row type.

ROW OF *table-name* or *view-name*
Specifies a row of fields with names and data types that are based on the column names and column data types of the table identified by *table-name* or the view identified by *view-name*. The data types of the anchor object columns have the same limitations that apply to *data-type3*.

ROW OF *cursor-variable-name*
Specifies a row of fields with names and data types that are based on the field names and field data types of the cursor variable identified by *cursor-variable-name*. The specified cursor variable must be one of the following objects (SQLSTATE 428HS):

- A global variable with a strongly typed cursor data type.
- A global variable with a weakly typed cursor data type that was created or declared with a CONSTANT clause specifying a select-statement where all the result columns are named.

ELEMENT OF *array-type-name*
Specifies an array type from which the element data type is used to derive the column list. If *array-type-name* identifies an array type with elements that are a row type, the field names of the row type are used as the column names. If the *array-type-name* identifies an array type with elements that are not row types, the single result column name is COLUMN_VALUE.

built-in-type
See "CREATE TABLE" for the description of built-in data types.

SPECIFIC *specific-name*
Provides a unique name for the instance of the function that is being defined. This specific name can be used when sourcing on this function, dropping the function, or commenting on the function. It can never be used to invoke the function. The unqualified form of *specific-name* is an SQL identifier. The qualified form is a *schema-name* followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another function instance that exists at the application server; otherwise an error is raised (SQLSTATE 42710).

The *specific-name* may be the same as an existing *function-name*.

If no qualifier is specified, the qualifier that was used for *function-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *function-name* or an error is raised (SQLSTATE 42882).

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmssxxx.

LANGUAGE SQL
Specifies that the function is written using SQL.

PARAMETER CCSID
Specifies the encoding scheme to use for all string data passed into and out of the function. If the PARAMETER CCSID clause is not specified, the default is PARAMETER CCSID UNICODE for Unicode databases, and PARAMETER CCSID ASCII for all other databases.

ASCII
Specifies that string data is encoded in the database code page. If the database is a Unicode database, PARAMETER CCSID ASCII cannot be specified (SQLSTATE 56031).

UNICODE
Specifies that character data is in UTF-8, and that graphic data is in UCS-2. If the database is not a Unicode database, PARAMETER CCSID UNICODE cannot be specified (SQLSTATE 56031).

DETERMINISTIC or NOT DETERMINISTIC
This optional clause specifies whether the function always returns the same results for given argument values (DETERMINISTIC) or whether the function depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC function must always return

the same table from successive invocations with identical inputs. Optimizations taking advantage of the fact that identical inputs always produce the same results are prevented by specifying NOT DETERMINISTIC.

EXTERNAL ACTION or NO EXTERNAL ACTION
Specifies whether the function takes an action that changes the state of an object that the database manager does not manage. An example of an external action is sending a message or writing a record to a file. The default is EXTERNAL ACTION.

EXTERNAL ACTION
Specifies that the function takes an action that changes the state of an object that the database manager does not manage.

NO EXTERNAL ACTION
Specifies that the function does not take any action that changes the state of an object that the database manager does not manage. The database manager uses this information during optimization of SQL statements.

READS SQL DATA, CONTAINS SQL, or MODIFIES SQL DATA
Specifies the classification of SQL statements that the function can run. The database manager verifies that the SQL statements that the function issues are consistent with this specification.

For the classification of each statement, see "SQL statements that can be executed in routines and triggers" in Developing User-defined Routines (SQL and External).

READS SQL DATA
Specifies that the function can run statements with a data access classification of READS SQL DATA, CONTAINS SQL, or NO SQL. The function cannot run SQL statements that modify data (SQLSTATE 42985). This is the default.

CONTAINS SQL
Specifies that the function can run only SQL statements with a data access classification of **CONTAINS SQL**. The function cannot run any SQL statements that read or modify data (SQLSTATE 42985).

MODIFIES SQL DATA
Specifies that the function can run any SQL statement except those statements that are not supported in any function.

ALLOW PARALLEL or DISALLOW PARALLEL
This clause specifies whether a UDF can be parallelized, that is, whether a single invocation of the UDF can cause several instances of the UDF (usually one instance per partition) to run in parallel. Parallelization usually improves overall performance, but is allowed only when all the following conditions are met:

- The **CONTAINS SQL** clause is specified.
- All invocations of the UDF are completely independent of each other.

DISALLOW PARALLEL is the default.

STATIC DISPATCH
This optional clause indicates that at function resolution time, a function is chosen based on the static types (declared types) of the parameters of the function.

CALLED ON NULL INPUT
This clause indicates that the function is called regardless of whether any of its arguments are null. It can return a null value or a non-null value. Responsibility for testing null argument values lies with the user-defined function.

The phrase NULL CALL may be used in place of CALLED ON NULL INPUT.

INHERIT SPECIAL REGISTERS
This optional clause indicates that updatable special registers in the function will inherit their initial values from the environment of the invoking statement. For a function that is invoked in the select-statement of a cursor, the initial values are inherited from the environment when the cursor is opened.

For a routine that is invoked in a nested object (for example, a trigger or a view), the initial values are inherited from the runtime environment (not the object definition).

No changes to the special registers are passed back to the caller of the function.

Some special registers, such as the datetime special registers, reflect a property of the statement currently executing, and are therefore never inherited from the caller.

PREDICATES

For predicates using this function, this clause identifies those that can exploit the index extensions, and can use the optional SELECTIVITY clause for the predicate's search condition. If the PREDICATES clause is specified, the function must be defined as DETERMINISTIC with NO EXTERNAL ACTION (SQLSTATE 42613). If the PREDICATES clause is specified, and the database is not a Unicode database, PARAMETER CCSID UNICODE must not be specified (SQLSTATE 42613). PREDICATES cannot be specified if SQL-function-body is a compound SQL (compiled) statement (SQLSTATE 42613).

predicate-specification

For details on predicate specification, see "CREATE FUNCTION (External Scalar)".

INHERIT ISOLATION LEVEL WITHOUT LOCK REQUEST or INHERIT ISOLATION LEVEL WITH LOCK REQUEST

Specifies whether or not a lock request can be associated with the isolation-clause of the statement when the function inherits the isolation level of the statement that invokes the function. The default is INHERIT ISOLATION LEVEL WITHOUT LOCK REQUEST.

INHERIT ISOLATION LEVEL WITHOUT LOCK REQUEST

Specifies that, as the function inherits the isolation level of the invoking statement, it cannot be invoked in the context of an SQL statement which includes a lock-request-clause as part of a specified isolation-clause (SQLSTATE 42601).

INHERIT ISOLATION LEVEL WITH LOCK REQUEST

Specifies that, as the function inherits the isolation level of the invoking statement, it also inherits the specified lock-request-clause.

SQL-function-body

Specifies the body of the function. Parameter names can be referenced in the SQL-function-body. Parameter names may be qualified with the function name to avoid ambiguous references.

For RETURN statement, see: RETURN statement.

For *Compound SQL (compiled)*, see: Compound SQL (compiled) statement.

For *Compound SQL (inlined)*, see: Compound SQL (inlined) statement.

NOT SECURED or SECURED

Specifies whether the function is considered secure for row and column access control. The default is NOT SECURED.

NOT SECURED

Indicates that the function is not considered secure. When the function is invoked, the arguments of the function must not reference a column for which a column mask is enabled and column level access control is activated for its table (SQLSTATE 428HA). This rule applies to the non secure user-defined functions that are invoked anywhere in the statement.

SECURED

Indicates that the function is considered secure. The function must be secure when it is referenced in a row permission or a column mask (SQLSTATE 428H8).

Rules

- ***Use of anchored data types:*** An anchored data type cannot refer to the following objects (SQLSTATE 428HS): a nickname, typed table, typed view, statistical view that is associated with an expression-based index, declared temporary table, row definition that is associated with a weakly typed cursor, object with a code page or collation that is different from the database code page or database collation.

- ***Use of cursor and row types:*** A function that uses a cursor type or row type for a parameter or returns a cursor type or row type can only be invoked from within a compound SQL (compiled) statement (SQLSTATE 428H2).
- ***Table access restrictions***: If a function is defined as READS SQL DATA, no statement in the function can access a table that is being modified by the statement that invoked the function (SQLSTATE 57053). For example, suppose the user-defined function BONUS() is defined as READS SQL DATA. If the statement UPDATE EMPLOYEE SET SALARY = SALARY + BONUS(EMPNO) is invoked, no SQL statement in the BONUS function can read from the EMPLOYEE table.

  If a function defined with MODIFIES SQL DATA contains nested CALL statements, read access to the tables being modified by the function (by either the function definition or the statement that invoked the function) is not allowed (SQLSTATE 57053).
- ***Use in a partitioned database environment***:
  - In a partitioned database environment, a scalar function defined using a compound SQL (compiled) statement defined as MODIFIES SQL can be referenced only on the right side of an assignment statement and the function reference cannot be part of an expression. Such an assignment statement cannot be in a Compound SQL (inlined) statement.
  - In a partitioned database environment, a scalar function defined using a compound SQL (compiled) statement which is defined as CONTAINS SQL is subject to additional restrictions: no explicit statement execution and all procedural logic must be supported without the need for executing a SQL statement.
  - In a partitioned database environment, a scalar function defined using a compound SQL (compiled) statement which is defined as READS SQL is always forced to run in the coordinator agent. It can also not be used in context of an UPDATE or DELETE statement.

Notes

- Resolution of function calls inside the function body is done according to the SQL path that is effective for the CREATE FUNCTION statement and does not change after the function is created.
- If an SQL function contains multiple references to any of the date or time special registers, all references return the same value, and it will be the same value returned by the register invocation in the statement that called the function.
- The body of an SQL function cannot contain a recursive call to itself or to another function or method that calls it, since such a function could not exist to be called.
- If an object referenced in the SQL function body does not exist or is marked invalid, or the definer temporarily doesn't have privileges to access the object, and if the database configuration parameter **auto_reval** is not set to DISABLED, then the SQL function will still be created successfully. The SQL function will be marked invalid and will be revalidated the next time it is invoked.
- The following rules are enforced by all statements that create functions or methods:
  - A function may not have the same signature as a method (comparing the first *parameter-type* of the function with the *subject-type* of the method).
  - A function and a method may not be in an overriding relationship. That is, if the function were a method with its first parameter as subject, it must not override, or be overridden by, another method. For more information about overriding methods, see the "CREATE TYPE (Structured)" statement.
  - Because overriding does not apply to functions, it is permissible for two functions to exist such that, if they were methods, one would override the other.

  For the purpose of comparing parameter-types in the preceding rules:
  - Parameter-names, lengths, AS LOCATOR, and FOR BIT DATA are ignored.
  - A subtype is considered to be different from its supertype.
- ***Privileges***: The definer of a function always receives the EXECUTE privilege on the function, as well as the right to drop the function. The definer of a function is also given the WITH GRANT OPTION on the

function if the definer has WITH GRANT OPTION on all privileges required to define the function, or if the definer has SYSADM or DBADM authority.

The definer of a function only acquires privileges if the privileges from which they are derived exist at the time the function is created. The definer must have these privileges either directly, or because PUBLIC has the privileges. Privileges held by groups of which the function definer is a member are not considered. When using the function, the connected user's authorization ID must have the valid privileges on the table or view that the nickname references at the data source.

- ***Setting of the default value***: Parameters of a function that are defined with a default value are set to their default value when the functions is invoked, but only if a value is not supplied for the corresponding argument, or is specified as DEFAULT, when the function is invoked.
- ***EXTERNAL ACTION functions***: If an EXTERNAL ACTION function is invoked in other than the outermost select list, the results are unpredictable since the number of times the function is invoked will vary depending on the access plan used.
- ***Creating a secure function***: Normally users with SECADM authority do not have privileges to create database objects such as triggers or functions. Typically they will examine the data accessed by the function, ensure it is secure, then grant the CREATE_SECURE_OBJECT authority to someone who currently has required privileges to create a secure user-defined function. After the function is created, they will revoke the CREATE_SECURE_OBJECT authority from the function owner.

  The SECURED attribute is considered to be an assertion that declares the user has established a change control audit procedure for all changes to the user-defined function. The database manager assumes that such a control audit procedure is in place for all subsequent ALTER FUNCTION statements or changes to external packages.
- ***Invoking other user-defined functions in a secure function***: If a secure user-defined function invokes other user-defined functions, the database manager does not validate whether those nested user-defined functions have the SECURED attribute. If those nested functions can access sensitive data, the user with SECADM authority needs to ensure those functions are allowed to access those data and a change control audit procedure has been established for all changes to those functions.
- ***Replacing an existing function such that the secure attribute is changed (from SECURED to NOT SECURED and vice versa)***: Packages and dynamically cached SQL statements that depend on the function may be invalidated because the secure attribute affects the access path selection for statements involving tables for which row or column level access control is activated.
- ***Rebinding dependent packages***: Every compiled SQL function has a dependent package. The package can be rebound at any time by using the REBIND_ROUTINE_PACKAGE procedure. Explicitly rebinding the dependent package does not revalidate an invalid function. Revalidate an invalid function with automatic revalidation or explicitly by using the ADMIN_REVALIDATE_DB_OBJECTS procedure. Function revalidation automatically rebinds the dependent package.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used:

  – NULL CALL can be specified in place of CALLED ON NULL INPUT

  The following syntax is accepted as the default behavior:

  – CCSID UNICODE in a Unicode database

  – CCSID ASCII in a non-Unicode database

Examples

- *Example 1:* Define a scalar function that returns the tangent of a value using the existing sine and cosine functions.

```
CREATE FUNCTION TAN (X DOUBLE)
  RETURNS DOUBLE
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
```

```
  DETERMINISTIC
  RETURN SIN(X)/COS(X)
```

- *Example 2:* Define a transform function for the structured type PERSON.

```
CREATE FUNCTION FROMPERSON (P PERSON)
  RETURNS ROW (NAME VARCHAR(10), FIRSTNAME VARCHAR(10))
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN VALUES (P..NAME, P..FIRSTNAME)
```

- *Example 3:* Define a table function that returns the employees in a specified department number.

```
CREATE FUNCTION DEPTEMPLOYEES (DEPTNO CHAR(3))
  RETURNS TABLE (EMPNO CHAR(6),
                 LASTNAME VARCHAR(15),
                 FIRSTNAME VARCHAR(12))
  LANGUAGE SQL
  READS SQL DATA
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN
    SELECT EMPNO, LASTNAME, FIRSTNME
      FROM EMPLOYEE
      WHERE EMPLOYEE.WORKDEPT = DEPTEMPLOYEES.DEPTNO
```

- *Example 4:* Define the table function from Example 3 with auditing.

```
CREATE FUNCTION DEPTEMPLOYEES (DEPTNO CHAR(3))
  RETURNS TABLE (EMPNO CHAR(6),
                 LASTNAME VARCHAR(15),
                 FIRSTNAME VARCHAR(12))
  LANGUAGE SQL
  MODIFIES SQL DATA
  NO EXTERNAL ACTION
  DETERMINISTIC
  BEGIN ATOMIC
    INSERT INTO AUDIT
    VALUES (USER,
            'Table: EMPLOYEE Prd: DEPTNO = ' CONCAT DEPTNO);
    RETURN
      SELECT EMPNO, LASTNAME, FIRSTNME
        FROM EMPLOYEE
        WHERE EMPLOYEE.WORKDEPT = DEPTEMPLOYEES.DEPTNO
  END
```

- *Example 5:* Define a scalar function that reverses a string.

```
CREATE FUNCTION REVERSE(INSTR VARCHAR(4000))
  RETURNS VARCHAR(4000)
  DETERMINISTIC NO EXTERNAL ACTION CONTAINS SQL
  BEGIN ATOMIC
  DECLARE REVSTR, RESTSTR VARCHAR(4000) DEFAULT '';
  DECLARE LEN INT;
  IF INSTR IS NULL THEN
  RETURN NULL;
  END IF;
  SET (RESTSTR, LEN) = (INSTR, LENGTH(INSTR));
  WHILE LEN > 0 DO
  SET (REVSTR, RESTSTR, LEN)
    = (SUBSTR(RESTSTR, 1, 1) CONCAT REVSTR,
    SUBSTR(RESTSTR, 2, LEN - 1),
    LEN - 1);
  END WHILE;
  RETURN REVSTR;
END
```

- *Example 6:* Create a function that increments a variable passed as an INOUT parameter and return any error as the return code.

```
CREATE FUNCTION increment(INOUT result INTEGER, IN delta INTEGER)
  RETURNS INTEGER
  BEGIN
```

```
    DECLARE code INTEGER DEFAULT 0;
    DECLARE SQLCODE INTEGER;
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION BEGIN
    SET code = SQLCODE;
    RETURN code;
  END;
  SET result = result + delta;
  RETURN code;
END@
```

- *Example 7:* Create a compiled SQL function that takes an XML document as input and returns the customer name.

```
CREATE FUNCTION get_customer_name_compiled(doc XML)
  RETURNS VARCHAR(25)
  BEGIN
    RETURN XMLCAST(XMLQUERY
       ('$d/customerinfo/name' PASSING doc AS "d")AS VARCHAR(25));
  END
```

- *Example 8:* Create a compiled SQL function that takes a phone number and a region number passed as IN parameters and returns the complete number in an OUT XML parameter.

```
CREATE FUNCTION construct_xml_phone
  (IN  phoneNo VARCHAR(20),
   IN  regionNo VARCHAR(8),
   OUT full_phone_xml XML)
  RETURNS VARCHAR(28)
  LANGUAGE SQL
  NO EXTERNAL ACTION
  BEGIN
    SET full_phone_xml = XMLELEMENT (NAME "phone", regionNo || phoneNo);
    RETURN regionNo || phoneNo;
  END
```

CREATE FUNCTION MAPPING

The CREATE FUNCTION MAPPING statement can define a mapping between a federated database function or function template and a data source function, or disable a default mapping between a federated database function and a data source function.

When defining a mapping, the CREATE FUNCTION MAPPING statement can associate the federated database function or template with a function at the following sources:

- A specified data source
- A range of data sources; for example, all data sources of a particular type and version

If multiple function mappings are applicable to a function, the most recent one is applied.

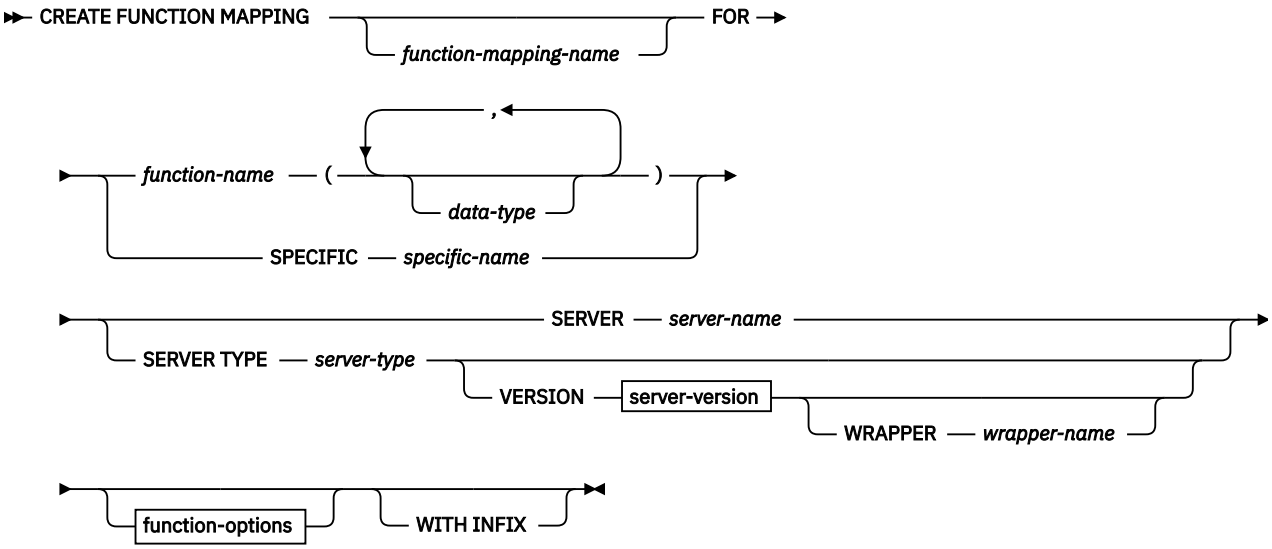
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

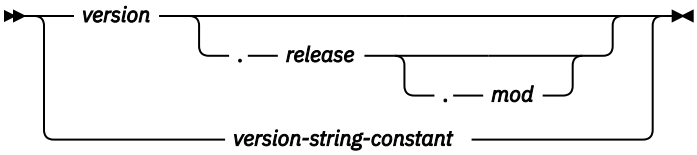
Authorization

The privileges held by the authorization ID of the statement must include DBADM authority.

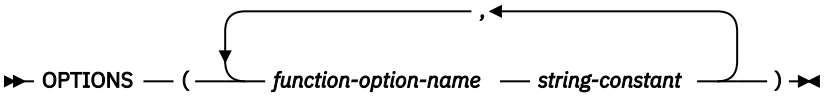
Syntax



server-version



function-options



Description

function-mapping-name
Names the function mapping. The name must not identify a function mapping that is already described in the catalog (SQLSTATE 42710).

If the *function-mapping-name* is omitted, a system-generated unique name is assigned.

function-name
Specifies the qualified or unqualified name of the federated database function or federated database function template from which to map.

data-type
For a function or function template that has input parameters, *data-type* specifies the data type of each parameter. The *data type* cannot be an XML or a user-defined type.

Empty parentheses can be used instead of specifying length, precision, or scale for the parameterized data types. It is recommended to use empty parentheses for the parameterized data types; for example, CHAR(). A parameterized data type is any one of the data types that can be defined with a specific length, scale, or precision. The parameterized data types are the string data types and the decimal data types. If you specify length, precision, or scale, it must be the same as that of the function template. If you omit parentheses altogether, the default length for the data type is used (see the description of the CREATE TABLE statement).

SPECIFIC *specific-name*
Identifies the function or function template from which to map. Specify *specific-name* to create a convenient function name.

SERVER *server-name*
Names the data source containing the function that is being mapped.

SERVER TYPE *server-type*
Identifies the type of data source containing the function that is being mapped.

VERSION
Identifies the version of the data source denoted by *server-type*.

version
Specifies the version number. The value must be an integer.

release
Specifies the number of the release of the version denoted by *version*. The value must be an integer.

mod
Specifies the number of the modification of the release denoted by *release*. The value must be an integer.

version-string-constant
Specifies the complete designation of the version. The *version-string-constant* can be a single value (for example, '8i'); or it can be the concatenated values of *version*, *release* and, if applicable, *mod* (for example, '8.0.3').

WRAPPER *wrapper-name*
Specifies the name of the wrapper that the federated server uses to interact with data sources of the type and version denoted by *server-type* and *server-version*.

OPTIONS
Specify configuration options for the function mapping to be created. Which options you can specify depends on the data source of the object for which a function mapping is being created. For a list of data sources and the function mapping options that apply to each, see Data source options. Each option value is a character string constant that must be enclosed in single quotation marks.

WITH INFIX
Specifies that the data source function be generated in infix format. The federated database system converts prefix notation to the infix notation that is used by the remote data source.

Notes

- A federated database function or function template can map to a data source function if:
  - The federated database function or template has the same number of input parameters as the data source function.
  - The data types that are defined for the federated function or template are compatible with the corresponding data types defined for the data source function.
- If a distributed request references a built-in database function that maps to a data source function, the optimizer develops strategies for invoking either function when the request is processed. The built-in database function is invoked if doing so requires less overhead than invoking the data source function. Otherwise, if invoking the built-in database function requires more overhead, the data source function is invoked.
- If a distributed request references a built-in database function template that maps to a data source function, only the data source function can be invoked when the request is processed. The template cannot be invoked because it has no executable code.
- Default function mappings can be rendered inoperable by disabling them (they cannot be dropped). To disable a default function mapping, code the CREATE FUNCTION MAPPING statement so that it specifies the name of the built-in database function within the mapping and sets the DISABLE option to 'Y'.
- Functions in the SYSIBM schema do not have a specific name. To override the default function mapping for a function in the SYSIBM schema, specify *function-name* using the explicit qualifier SYSIBM; for example, SYSIBM.LENGTH().

- A CREATE FUNCTION MAPPING statement within a given unit of work (UOW) cannot be processed (SQLSTATE 55007) under either of the following conditions:
  - The statement references a single data source, and the UOW already includes one of the following:
    - A SELECT statement that references a nickname for a table or view within this data source
    - An open cursor on a nickname for a table or view within this data source
    - Either an INSERT, DELETE, or UPDATE statement issued against a nickname for a table or view within this data source
  - The statement references a category of data sources (for example, all data sources of a specific type and version), and the UOW already includes one of the following:
    - A SELECT statement that references a nickname for a table or view within one of these data sources
    - An open cursor on a nickname for a table or view within one of these data sources
    - Either an INSERT, DELETE, or UPDATE statement issued against a nickname for a table or view within one of these data sources
- **Create function mapping to table or row functions:** A create function mapping to remote functions that returns a table or a row is not supported in a federated database.
- **Syntax alternatives:** The following syntax is supported for compatibility with previous versions of Db2:
  - ADD can be specified before *function-option-name string-constant*.

Examples

- *Example 1:* Map a function template to a UDF that all Oracle data sources can access. The template is called STATS and belongs to a schema called NOVA. The Oracle UDF is called STATISTICS and belongs to a schema called STAR.

```
CREATE FUNCTION MAPPING MY_ORACLE_FUN1
  FOR NOVA.STATS (DOUBLE, DOUBLE)
  SERVER TYPE ORACLE
  OPTIONS (REMOTE_NAME 'STAR.STATISTICS')
```

- *Example 2:* Map a function template called BONUS to a UDF, also called BONUS, that is used at an Oracle data source called ORACLE1.

```
CREATE FUNCTION MAPPING MY_ORACLE_FUN2
  FOR BONUS()
  SERVER ORACLE1
  OPTIONS (REMOTE_NAME 'BONUS')
```

- *Example 3:* Assume that there is a default function mapping between the WEEK system function that is defined to the federated database and a similar function that is defined to Oracle data sources. When a query that requests Oracle data and that references WEEK is processed, either WEEK or its Oracle counterpart will be invoked, depending on which one is estimated by the optimizer to require less overhead. The DBA wants to find out how performance would be affected if only WEEK were invoked for such queries. To ensure that WEEK is invoked each time, the DBA must disable the mapping.

```
CREATE FUNCTION MAPPING
  FOR SYSFUN.WEEK(INT)
  SERVER TYPE ORACLE
  OPTIONS (DISABLE 'Y')
```

- *Example 4:* Map the federated function UCASE(CHAR) to a UDF that is used at an Oracle data source called ORACLE2. Include the estimated number of instructions per invocation of the Oracle UDF.

```
CREATE FUNCTION MAPPING MY_ORACLE_FUN4
  FOR SYSFUN.UCASE(CHAR)
  SERVER ORACLE2
  OPTIONS
    (REMOTE_NAME 'UPPERCASE',
    INSTS_PER_INVOC '1000')
```

CREATE GLOBAL TEMPORARY TABLE

The CREATE GLOBAL TEMPORARY TABLE statement creates a description of a temporary table at the current server. Each session that selects from a created temporary table retrieves only rows that the same session has inserted. When the session terminates, the rows of the table associated with the session are deleted.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

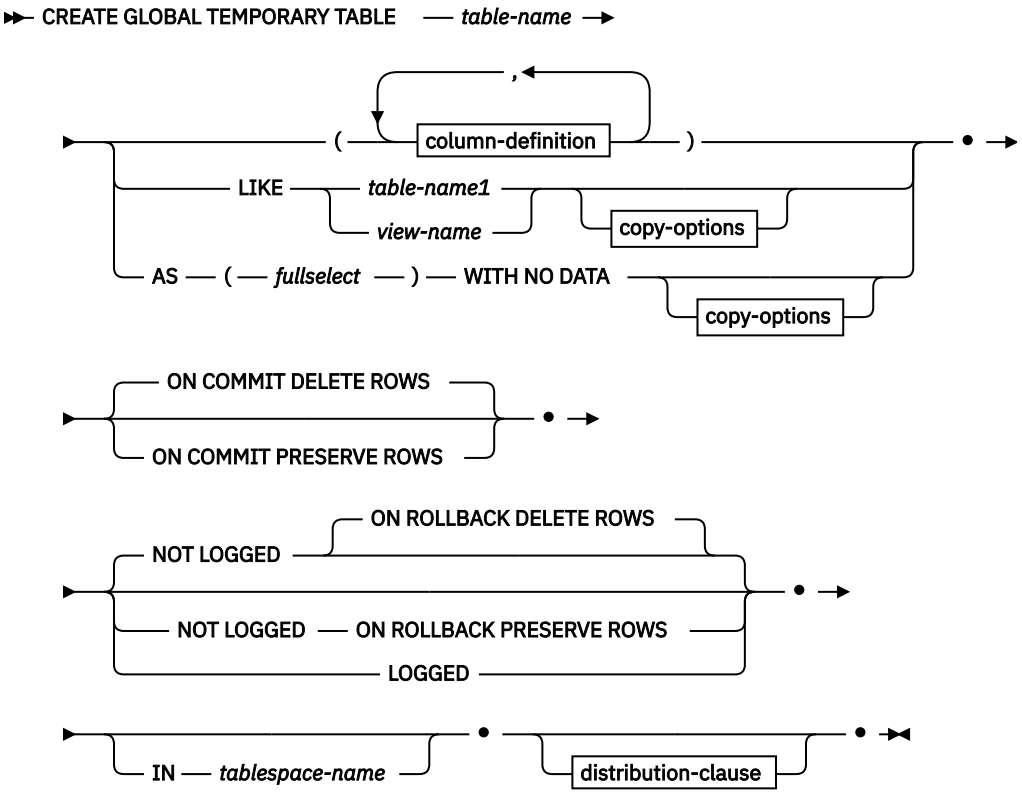
The privileges held by the authorization ID of the statement must include either DBADM authority, or CREATETAB authority in combination with further authorization, as described here:

- One of the following privileges and authorities:
  - USE privilege on the table space
  - SYSADM
  - SYSCTRL
- Plus one of these privileges and authorities:
  - IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the table does not exist
  - CREATEIN privilege on the schema, if the schema name of the table refers to an existing schema
  - SCHEMAADM authority on the schema, if the schema name of the table refers to an existing schema

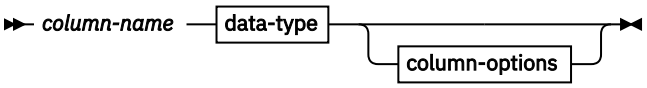
When defining a table using LIKE or a fullselect, the privileges held by the authorization ID of the statement must also include at least one of the following on each identified table or view:

- SELECT privilege on the table or view
- CONTROL privilege on the table or view
- SELECTIN privilege on the schema containing the table or view
- DATAACCESS authority on the schema containing the table or view
- DATAACCESS authority

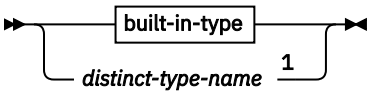
Syntax



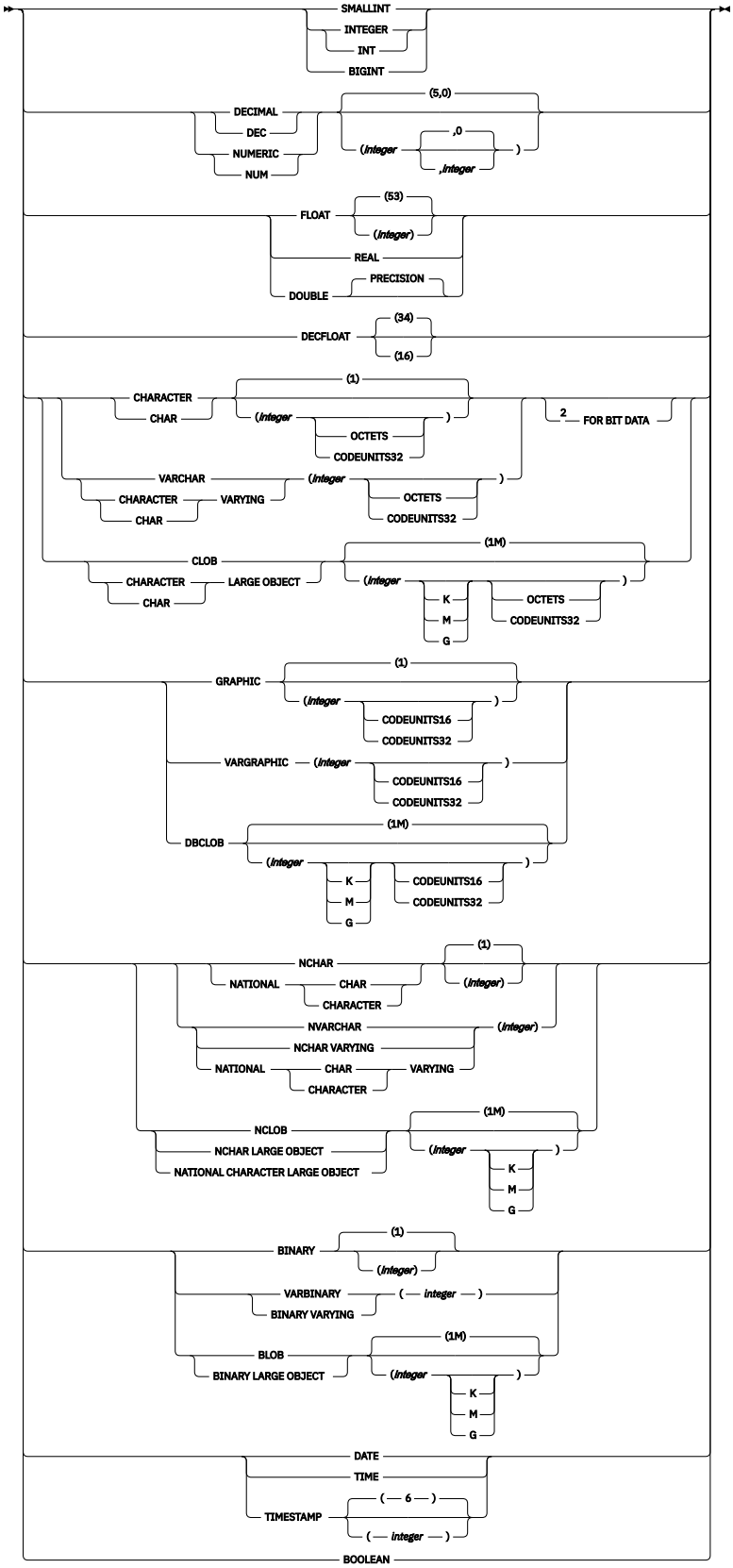
column-definition



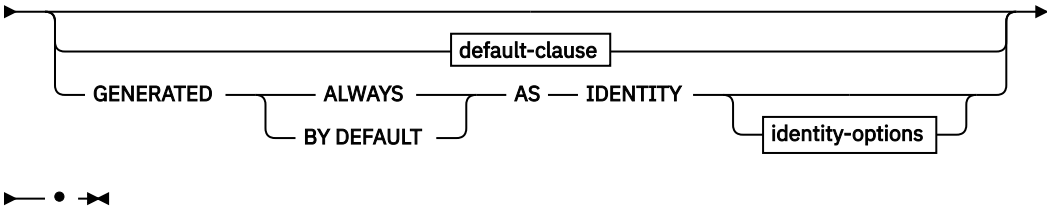
data-type



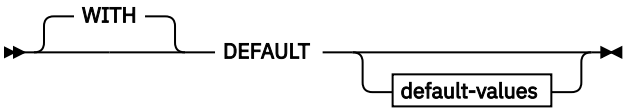
built-in-type



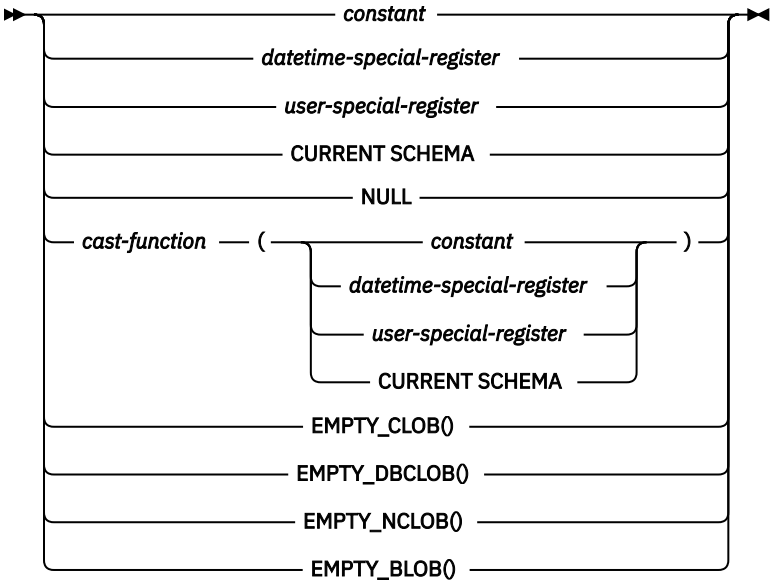
column-options



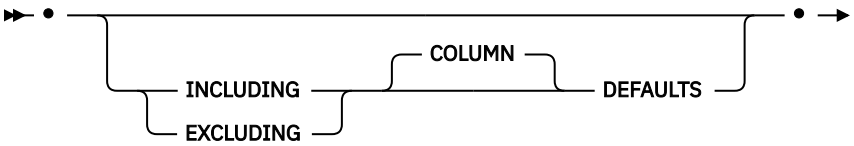
default-clause

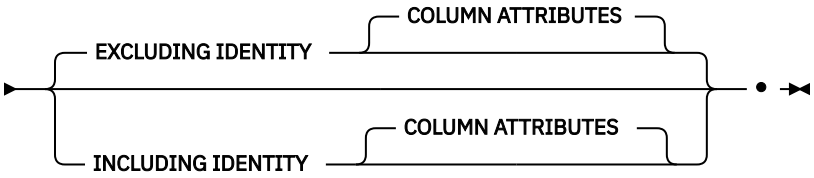


default-values

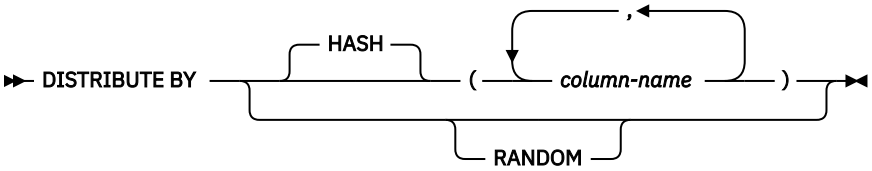


copy-options





distribution-clause



Notes:

[1] The specified distinct type cannot have any data type constraints and the source type cannot be an anchored data type.
[2] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

Description

table-name
Names the table. The name, including the implicit or explicit qualifier, must not identify a table, view, nickname, or alias described in the catalog. If a two-part name is specified, the schema name cannot begin with 'SYS' (SQLSTATE 42939).

column-definition
Defines the attributes of a column of the temporary table.

column-name
Names a column of the table. The name cannot be qualified, and the same name cannot be used for more than one column of the table (SQLSTATE 42711).

A table can have the following:

- A 4K page size with a maximum of 500 columns, where the byte counts of the columns must not be greater than 4 005.
- An 8K page size with a maximum of 1 012 columns, where the byte counts of the columns must not be greater than 8 101.
- A 16K page size with a maximum of 1 012 columns, where the byte counts of the columns must not be greater than 16 293.
- A 32K page size with a maximum of 1 012 columns, where the byte counts of the columns must not be greater than 32 677.

A created temporary table cannot have a row-begin column, row-end column, or a transaction-start-ID column.

For more details, see "Row Size" in “CREATE TABLE ” on page 1351.

data-type
Specifies the data type of the column

built-in-type
Specifies a built-in data type. See "CREATE TABLE" for a description of *built-in-type*.

An XML and SYSPROC.DB2SECURITYLABEL data type cannot be specified for a created temporary table.

distinct-type-name
For a user-defined type that is a distinct type. If a distinct type name is specified without a schema name, the distinct type name is resolved by searching the schemas on the SQL path (defined by the FUNCPATH preprocessing option for static SQL and by the CURRENT PATH register for dynamic SQL).

If a column is defined using a distinct type, then the data type of the column is the distinct type. The length and the scale of the column are respectively the length and the scale of the source type of the distinct type. The distinct type for a column cannot have any data type constraints and the source type cannot be an anchored data type (SQLSTATE 428H2).

column-options
Defines additional options related to the columns of the table.

NOT NULL
Prevents the column from containing null values. For specification of null values, see NOT NULL in "CREATE TABLE".

default-clause
Specifies a default value for the column.

WITH
An optional keyword.

DEFAULT
Provides a default value in the event a value is not supplied on INSERT or is specified as DEFAULT on INSERT or UPDATE. If a default value is not specified following the DEFAULT keyword, the default value depends on the data type of the column as shown in "ALTER TABLE".

If the column is based on a column of a typed table, a specific default value must be specified when defining a default. A default value cannot be specified for the object identifier column of a typed table (SQLSTATE 42997).

If a column is defined using a distinct type, then the default value of the column is the default value of the source data type cast to the distinct type.

If a column is defined using a structured type, the *default-clause* cannot be specified (SQLSTATE 42842).

Omission of DEFAULT from a *column-definition* results in the use of the null value as the default for the column. If such a column is defined NOT NULL, then the column does not have a valid default.

default-values
Specific types of default values that can be specified are as follows.

constant
Specifies the constant as the default value for the column. The specified constant must:

- represent a value that could be assigned to the column in accordance with the rules of assignment
- not be a floating-point constant unless the column is defined with a floating-point data type
- be a numeric constant or a decimal floating-point special value if the data type of the column is a decimal floating-point. Floating-point constants are first interpreted as DOUBLE and then converted to decimal floating-point if the target column is DECFLOAT. For DECFLOAT(16) columns, decimal constants having precision greater than 16 digits will be rounded using the rounding modes specified by the CURRENT DECFLOAT ROUNDING MODE special register.
- not have nonzero digits beyond the scale of the column data type if the constant is a decimal constant (for example, 1.234 cannot be the default for a DECIMAL(5,2) column)
- be expressed with no more than 254 bytes including the quote characters, any introducer character such as the X for a hexadecimal constant, and characters from the fully qualified function name and parentheses when the constant is the argument of a *cast-function*

datetime-special-register
Specifies the value of the datetime special register (CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP) at the time of INSERT, UPDATE, or LOAD as the default for the column. The data type of the column must be the data type that corresponds to the special register specified (for example, data type must be DATE when CURRENT DATE is specified).

user-special-register
Specifies the value of the user special register (CURRENT USER, SESSION_USER, SYSTEM_USER) at the time of INSERT, UPDATE, or LOAD as the default for the column. The data type of the column must be a character string with a length not less than the length attribute of a user special register. Note that USER can be specified in place of SESSION_USER and CURRENT_USER can be specified in place of CURRENT USER.

CURRENT SCHEMA
Specifies the value of the CURRENT SCHEMA special register at the time of INSERT, UPDATE, or LOAD as the default for the column. If CURRENT SCHEMA is specified, the data type of the column must be a character string with a length greater than or equal to the length attribute of the CURRENT SCHEMA special register.

NULL
Specifies NULL as the default for the column. If NOT NULL was specified, DEFAULT NULL may be specified within the same column definition but will result in an error on any attempt to set the column to the default value.

cast-function
This form of a default value can only be used with columns defined as a distinct type, BLOB or datetime (DATE, TIME or TIMESTAMP) data type. For distinct type, with the exception of distinct types based on BLOB or datetime types, the name of the function must match the name of the distinct type for the column. If qualified with a schema name, it must be the same as the schema name for the distinct type. If not qualified, the schema name from function resolution must be the same as the schema name for the distinct type. For a distinct type based on a datetime type, where the default value is a constant, a function must be used and the name of the function must match the name of the source type of the distinct type with an implicit or explicit schema name of SYSIBM. For other datetime columns, the corresponding datetime function may also be used. For a BLOB or a distinct type based on BLOB, a function must be used and the name of the function must be BLOB with an implicit or explicit schema name of SYSIBM.

constant
Specifies a constant as the argument. The constant must conform to the rules of a constant for the source type of the distinct type or for the data type if not a distinct type. If the *cast-function* is BLOB, the constant must be a string constant.

datetime-special-register
Specifies CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP. The source type of the distinct type of the column must be the data type that corresponds to the specified special register.

user-special-register
Specifies CURRENT USER, SESSION_USER, or SYSTEM_USER. The data type of the source type of the distinct type of the column must be a string data type with a length of at least 8 bytes. If the *cast-function* is BLOB, the length attribute must be at least 8 bytes.

CURRENT SCHEMA
Specifies the value of the CURRENT SCHEMA special register. The data type of the source type of the distinct type of the column must be a character string with a length greater than or equal to the length attribute of the CURRENT SCHEMA special register. If the cast-function is BLOB, the length attribute must be at least 8 bytes.

EMPTY_CLOB(), EMPTY_DBCLOB(), or EMPTY_BLOB()
Specifies a zero-length string as the default for the column. The column must have the data type that corresponds to the result data type of the function.

If the value specified is not valid, an error is returned (SQLSTATE 42894).

IDENTITY and *identity-options*
For specification of identity columns, see IDENTITY and *identity-options* in "CREATE TABLE".

LIKE *table-name1* or *view-name* or *nickname*
Specifies that the columns of the table have exactly the same name and description as the columns of the identified table (*table-name1*), view (*view-name*), or nickname (*nickname*). The name specified after LIKE must identify a table, view, or nickname that exists in the catalog, or a declared temporary table. A typed table or typed view cannot be specified (SQLSTATE 428EC). A protected table cannot be specified (SQLSTATE 42962). A table that has a column defined as IMPLICITLY HIDDEN cannot be specified (SQLSTATE 560AE).

The use of LIKE is an implicit definition of *n* columns, where *n* is the number of columns in the identified table (including implicitly hidden columns), view, or nickname. The implicit definition depends on what is identified after LIKE.

- If a table is identified, then the implicit definition includes the column name, data type and nullability characteristic of each of the columns of *table-name1*. If EXCLUDING COLUMN DEFAULTS is not specified, then the column default is also included.
- If a view is identified, then the implicit definition includes the column name, data type, and nullability characteristic of each of the result columns of the fullselect defined in *view-name*. The data types of the view columns must be data types that are valid for columns of a table.
- If a nickname is identified, then the implicit definition includes the column name, data type, and nullability characteristic of each column of *nickname*.
- If a random distribution table using the random by generation method is identified, then the **RANDOM_DISTRIBUTION_KEY** column used for generation of random distribution values is not included. Unless the new table being created shares the same table distribution.

Column default and identity column attributes may be included or excluded, based on the *copy-attributes* clauses. The implicit definition does not include any other attributes of the identified table, view, or nickname. Thus the new table does not have any unique constraints, foreign key constraints, triggers, indexes, table partitioning keys, or distribution keys. The table is created in the table space implicitly or explicitly specified by the IN clause, and the table has any other optional clause only if the optional clause is specified.

When a table is identified in the LIKE clause and that table contains a ROW CHANGE TIMESTAMP column, the corresponding column of the new table inherits only the data type of the ROW CHANGE TIMESTAMP column. The new column is not considered to be a generated column.

If row or column level access control (RCAC) is enforced for *table-name1*, RCAC is not inherited by the new table.

AS (*fullselect*) WITH NO DATA
Specifies that the columns of the table have the same name and description as the columns that would appear in the derived result table of the fullselect if the fullselect were to be executed. The use of AS (*fullselect*) is an implicit definition of *n* columns for the created temporary table, where *n* is the number of columns that would result from the fullselect.

The implicit definition includes the following attributes of the *n* columns (if applicable to the data type):

- Column name
- Data type, length, precision, and scale
- Nullability

The following attributes are not included (the default value and identity attributes can be included by using the *copy-options*):

- Default value
- Identity attributes
- Hidden attribute
- ROW CHANGE TIMESTAMP

The implicit definition does not include any other optional attributes of the tables or views referenced in the fullselect.

Every select list element must have a unique name (SQLSTATE 42711). The AS clause can be used in the select clause to provide unique names. The fullselect must not refer to host variables or include parameter markers. The data types of the result columns of the fullselect must be data types that are valid for columns of a table.

If row or column level access control (RCAC) is enforced for any table that is specified in *fullselect*, RCAC is not cascaded to the new table.

copy-options

These options specify whether to copy additional attributes of the source result table definition (table, view, or fullselect).

INCLUDING COLUMN DEFAULTS

Column defaults for each updatable column of the source result table definition are copied. Columns that are not updatable will not have a default defined in the corresponding column of the created table.

If LIKE *table-name1* is specified, and *table-name1* identifies a base table, created temporary table, or declared temporary table, then INCLUDING COLUMN DEFAULTS is the default.

EXCLUDING COLUMN DEFAULTS

Column defaults are not copied from the source result table definition.

This clause is the default, except when LIKE *table-name* is specified and *table-name* identifies a base table, created temporary table, or declared temporary table.

INCLUDING IDENTITY COLUMN ATTRIBUTES

If available, identity column attributes (START WITH, INCREMENT BY, and CACHE values) are copied from the source's result table definition. It is possible to copy these attributes if the element of the corresponding column in the table, view, or fullselect is the name of a column of a table, or the name of a column of a view which directly or indirectly maps to the column name of a base table or created temporary table with the identity property. In all other cases, the columns of the new temporary table will not get the identity property. For example:

- The select list of the fullselect includes multiple instances of the name of an identity column (that is, selecting the same column more than once)
- The select list of the fullselect includes multiple identity columns (that is, it involves a join)
- The identity column is included in an expression in the select list
- The fullselect includes a set operation (union, except, or intersect).

EXCLUDING IDENTITY COLUMN ATTRIBUTES

Identity column attributes are not copied from the source result table definition.

ON COMMIT

Specifies the action taken on the created temporary table when a COMMIT operation is performed. The default is DELETE ROWS.

DELETE ROWS

All rows of the table will be deleted if no WITH HOLD cursor is open on the table.

PRESERVE ROWS

Rows of the table will be preserved.

LOGGED or NOT LOGGED

Specifies whether operations for the table are logged. The default is NOT LOGGED ON ROLLBACK DELETE ROWS.

NOT LOGGED

Specifies that insert, update, or delete operations against the table are not to be logged, but that the creation or dropping of the table is to be logged. During a ROLLBACK (or ROLLBACK TO SAVEPOINT) operation:

- If the table had been created within a unit of work (or savepoint), the table is dropped
- If the table had been dropped within a unit of work (or savepoint), the table is recreated, but without any data

ON ROLLBACK

Specifies the action that is to be taken on the not logged created temporary table when a ROLLBACK (or ROLLBACK TO SAVEPOINT) operation is performed. The default is DELETE ROWS.

DELETE ROWS

If the table data has been changed, all the rows will be deleted.

PRESERVE ROWS
Rows of the table will be preserved.

LOGGED
Specifies that insert, update, or delete operations against the table as well as the creation or dropping of the table are to be logged.

IN *tablespace-name*
Identifies the table space in which the created temporary table will be instantiated. The table space must exist and be a USER TEMPORARY table space (SQLSTATE 42838), over which the authorization ID of the statement has USE privilege (SQLSTATE 42501). If this clause is not specified, a table space for the table is determined by choosing the USER TEMPORARY table space with the smallest sufficient page size over which the authorization ID of the statement has USE privilege. When more than one table space qualifies, preference is given according to who was granted the USE privilege:

1. The authorization ID
2. A group to which the authorization ID belongs
3. PUBLIC

If more than one table space still qualifies, the final choice is made by the database manager. When no USER TEMPORARY table space qualifies, an error is raised (SQLSTATE 42727).

Determination of the table space can change when:

- Table spaces are dropped or created
- USE privileges are granted or revoked

The sufficient page size of a table is determined by either the byte count of the row or the number of columns. For more details, see "Row Size" in “CREATE TABLE ” on page 1351.

distribution-clause
Specifies the database partitioning or the way the data is distributed across multiple database partitions.

DISTRIBUTE BY HASH (*column-name*, ...)
Specifies the use of the default hashing function on the specified columns, called a *distribution key*, as the distribution method across database partitions. The *column-name* must be an unqualified name that identifies a column of the table (SQLSTATE 42703). The same column must not be identified more than once (SQLSTATE 42709). No column whose data type is BLOB, CLOB, DBCLOB, XML, distinct type based on any of these types, or structured type can be used as part of a distribution key (SQLSTATE 42962).

If this clause is not specified, and the table resides in a multiple partition database partition group with multiple database partitions, a default distribution key is automatically defined.

If none of the columns satisfies the requirements for a default distribution key, the table is created without one. Such tables are allowed only in table spaces that are defined on single-partition database partition groups.

For tables in table spaces that are defined on single-partition database partition groups, any collection of columns with data types that are valid for a distribution key can be used to define the distribution key. If this clause is not specified, no distribution key is created.

DISTRIBUTE BY RANDOM
Specifies that the database manager will select a distribution key to spread data evenly across all database partitions of the database partitioning group. Data distribution is accomplished by using a *random by generation* method. In this method, the database manager will include a column in the table to generate and store a generated value to use in the hashing function. The column will be created with the **IMPLICITLY HIDDEN** clause so that it does not appear in queries unless explicitly included. The value of the column will be automatically generated as new rows are added to the table. By default, the column name is **RANDOM_DISTRIBUTION_KEY**. If it collides with the existing column, a non-conflicting name will be generated by the database manager.

Notes

- A user temporary table space must exist before a created temporary table can be created (SQLSTATE 42727).
- Data row compression is enabled for a created temporary table. When the database manager determines that there is a performance gain, table row data with XML documents stored inline in the base table object is compressed. However, data compression of the XML storage object of a created temporary table is not supported.
- Index compression is enabled by default for indexes that are created on created temporary tables. Compression will be shown as on, but indexes will not be compressed if the correct license (IBM Db2 Storage Optimization Feature) is not applied.
- ***Instantiation and termination***: For the explanations that follow, P denotes a session and T is a created temporary table in the session P:
  - An empty instance of T is created as a result of the first reference to T that is executed in P.
  - Any SQL statement in P can make reference to T and any reference to T in P is a reference to that same instance of T.
  - Assuming that the ON COMMIT DELETE ROWS clause was specified implicitly or explicitly, then when a commit operation terminates a unit of work in P, and there is no open WITH HOLD cursor in P that is dependent on T, the commit includes the operation DELETE FROM T.
  - When a rollback operation terminates a unit of work or a savepoint in P, and that unit of work or savepoint includes a modification to T:
    - If NOT LOGGED was specified, all rows from T are deleted unless ON ROLLBACK PRESERVE ROWS was also specified
    - If NOT LOGGED was not specified, the changes to T are undone
  - If NOT LOGGED was specified and an INSERT, UPDATE or DELETE statement fails during execution (as opposed to a compilation error), all rows from T are deleted.
  - When a rollback operation terminates a unit of work or a savepoint in P, and that unit of work or savepoint includes the creation of T, then the rollback includes the operation DROP TABLE T.
  - If a rollback operation terminates a unit of work or a savepoint in P, and that unit of work or savepoint includes the drop of a created temporary table T, then the rollback will undo the drop of the table. If NOT LOGGED was specified, then the table will also have been emptied.
  - When the application process that referenced T terminates or disconnects from the database, the private instance of T is dropped and its instantiated rows are destroyed.
  - When the connection to the server at which T was referenced terminates, the private instance of T is dropped and its instantiated rows are destroyed.
- ***Restrictions on the use of created temporary tables***: Created temporary tables cannot:
  - Be specified in an ALTER, LOCK, or RENAME statement (SQLSTATE 42995)
  - Be specified in referential constraints (SQLSTATE 42995)
- ***Syntax alternatives***: The following alternatives are non-standard. They are supported for compatibility with earlier product versions or with other database products.
  - DEFINITION ONLY can be specified in place of WITH NO DATA
  - The PARTITIONING KEY clause or DISTRIBUTE ON clause can be specified in place of the DISTRIBUTE BY clause.
  - When specifying the value of the datetime special register, NOW() can be specified in place of CURRENT_TIMESTAMP.
  - In a CHAR or VARCHAR column definition, you do not need to specify the CCSID explicitly; the correct CCSID will be used automatically. However, if you do specify the CCSID explicitly, it must correspond to the type of database being used:
    - `CCSID ASCII` for a non-unicode database

- CCSID UNICODE for a unicode database

Examples

- *Example 1*: Create a temporary table, CURRENTMAP. Name two columns, CODE and MEANING, both of which cannot contain nulls. CODE contains numeric data and MEANING has character data.

```
CREATE GLOBAL TEMPORARY TABLE CURRENTMAP
   (CODE        INTEGER      NOT NULL,
    MEANING     VARCHAR(254) NOT NULL)
```

- *Example 2*: Create a temporary table, TMPDEPT.

```
CREATE GLOBAL TEMPORARY TABLE TMPDEPT
   (TMPDEPTNO   CHAR(3)      NOT NULL,
    TMPDEPTNAME VARCHAR(36)  NOT NULL,
    TMPMGRNO    CHAR(6),
    TMPLOCATION CHAR(16) )
```

CREATE HISTOGRAM TEMPLATE

The CREATE HISTOGRAM TEMPLATE statement defines a template describing the type of histogram that can be used to override one or more of the default histograms of a service class or a work class.

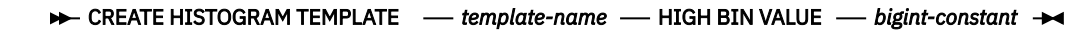
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include WLMADM or DBADM authority.

Syntax



Description

template-name
Names the histogram template. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The name must not identify an existing histogram template at the current server (SQLSTATE 42710). The name must not begin with the characters 'SYS' (SQLSTATE 42939).

HIGH BIN VALUE *bigint-constant*
Specifies the top value of the second to last bin (the last bin has an unbounded top value). The units depend on how the histogram is used. The maximum value is 268 435 456.

Rules

- A workload management (WLM)-exclusive SQL statement must be followed by a COMMIT or a ROLLBACK statement (SQLSTATE 5U021). WLM-exclusive SQL statements are:
  - CREATE HISTOGRAM TEMPLATE, ALTER HISTOGRAM TEMPLATE, or DROP (HISTOGRAM TEMPLATE)
  - CREATE SERVICE CLASS, ALTER SERVICE CLASS, or DROP (SERVICE CLASS)
  - CREATE THRESHOLD, ALTER THRESHOLD, or DROP (THRESHOLD)
  - CREATE WORK ACTION SET, ALTER WORK ACTION SET, or DROP (WORK ACTION SET)
  - CREATE WORK CLASS SET, ALTER WORK CLASS SET, or DROP (WORK CLASS SET)

  - CREATE WORKLOAD, ALTER WORKLOAD, or DROP (WORKLOAD)
  - GRANT (Workload Privileges) or REVOKE (Workload Privileges)
- A WLM-exclusive SQL statement cannot be issued within a global transaction (SQLSTATE 51041) such as, for example, an XA transaction.

Notes

- Only one uncommitted WLM-exclusive SQL statement at a time is allowed across all partitions. If an uncommitted WLM-exclusive SQL statement is executing, subsequent WLM-exclusive SQL statements will wait until the current WLM-exclusive SQL statement commits or rolls back.
- Changes are written to the system catalog, but do not take effect until they are committed, even for the connection that issues the statement.

Example

Create a histogram template named LIFETIMETEMP on service class PAYROLL in service superclass ADMIN that will override the default activity lifetime histogram template with a new high bin value of 90 000, which represents 90 000 milliseconds. This will produce a histogram with exponentially increasing bin ranges, ending with a bin whose range is 90 000 to infinity.

```
CREATE HISTOGRAM TEMPLATE LIFETIMETEMP
  HIGH BIN VALUE 90000

CREATE SERVICE CLASS PAYROLL
  UNDER ADMIN ACTIVITY LIFETIME HISTOGRAM TEMPLATE LIFETIMETEMP
```

CREATE INDEX

The CREATE INDEX statement is used to define an index on a database table.

An index can be defined on XML data, or on relational data. The CREATE INDEX statement is also used to create an index specification (metadata that indicates to the optimizer that a data source table has an index).

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

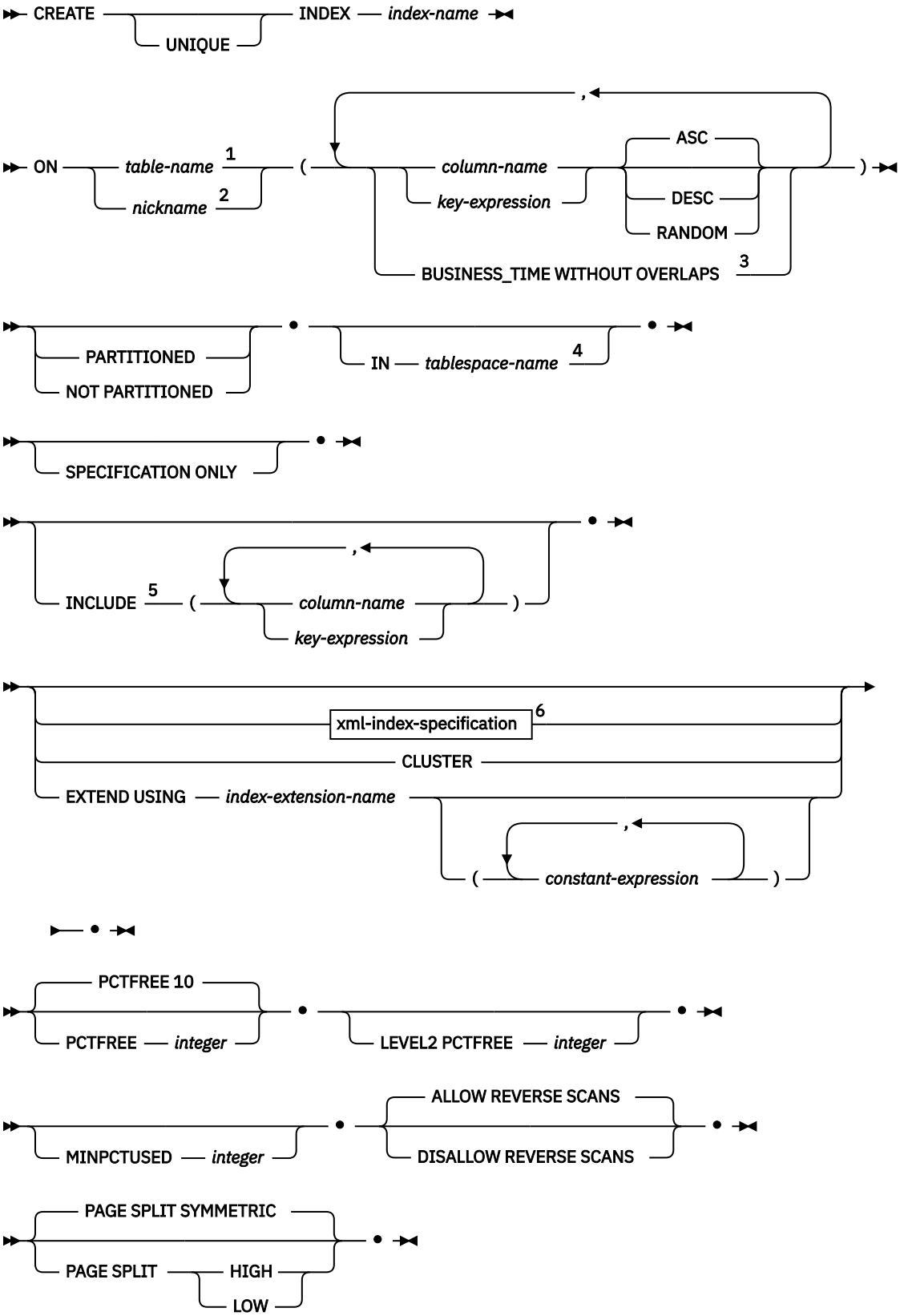
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

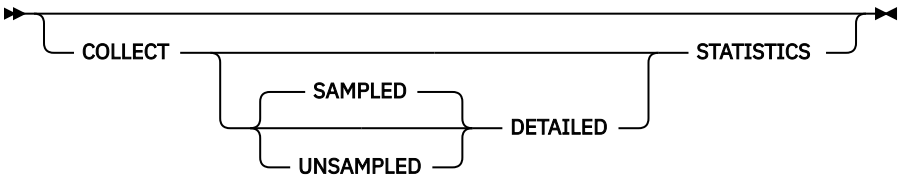
- One of:
  - CONTROL privilege on the table or nickname on which the index is defined
  - INDEX privilege on the table or nickname on which the index is defined
  - SCHEMAADM authority on the schema containing the table or nickname on which the index is defined

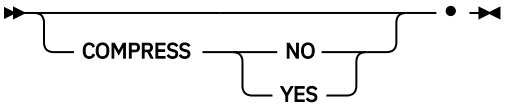
  and one of:
  - IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the index does not exist
  - CREATEIN privilege on the schema, if the schema name of the index refers to an existing schema
  - SCHEMAADM authority on the schema, if the schema name of the index refers to an existing schema
- DBADM authority

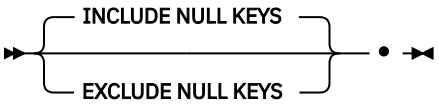
No explicit privilege is required to create an index on a declared temporary table.

Syntax





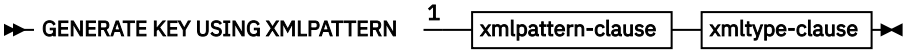




Notes:

[1] In a federated system, *table-name* must identify a table in the federated database. It cannot identify a data source table.
[2] If *nickname* is specified, the CREATE INDEX statement creates an index specification. In this case, INCLUDE, *xml-index-specification*, CLUSTER, EXTEND USING, PCTFREE, MINPCTUSED, DISALLOW REVERSE SCANS, ALLOW REVERSE SCANS, PAGE SPLIT, or COLLECT STATISTICS cannot be specified.
[3] The BUSINESS_TIME WITHOUT OVERLAPS clause can be specified only if UNIQUE is specified.
[4] The IN *tablespace-name* clause can be specified only for a nonpartitioned index on a partitioned table.
[5] The INCLUDE clause can be specified only if UNIQUE is specified.
[6] If *xml-index-specification* is specified, *column-name* DESC, INCLUDE, or CLUSTER cannot be specified.

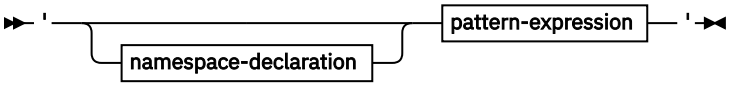
xml-index-specification



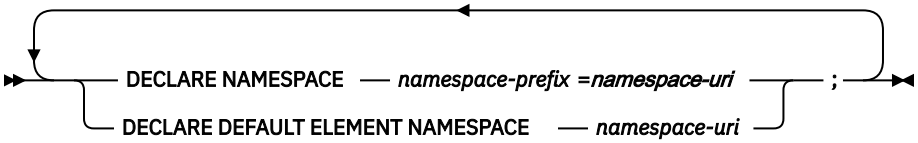
Notes:

[1] The alternative syntax GENERATE KEYS USING XMLPATTERN can be used.

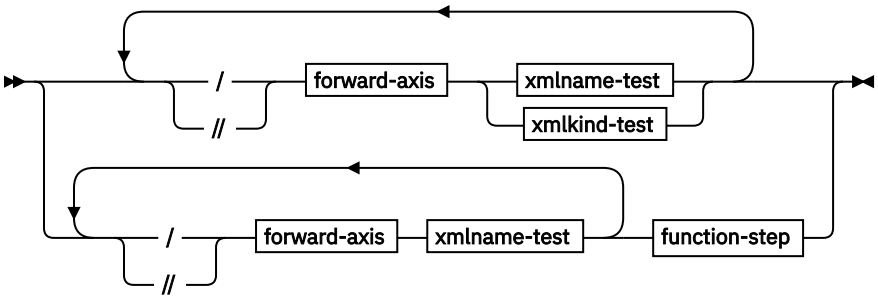
xmlpattern-clause



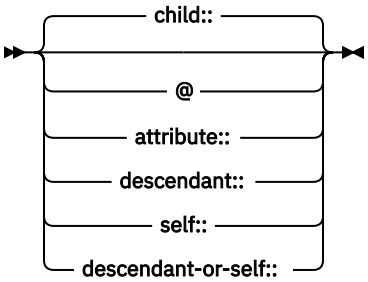
namespace-declaration



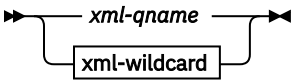
pattern-expression



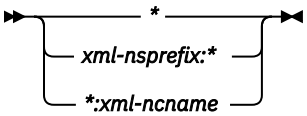
forward-axis



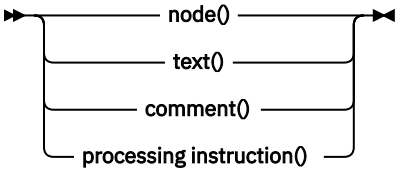
xmlname-test



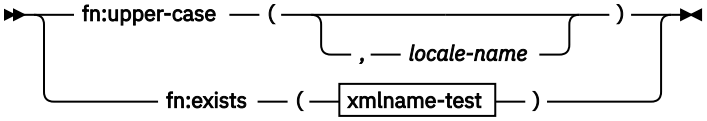
xml-wildcard



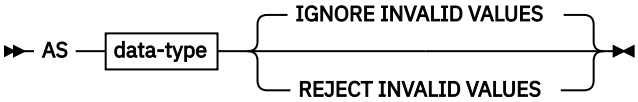
xmlkind-test



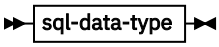
function-step



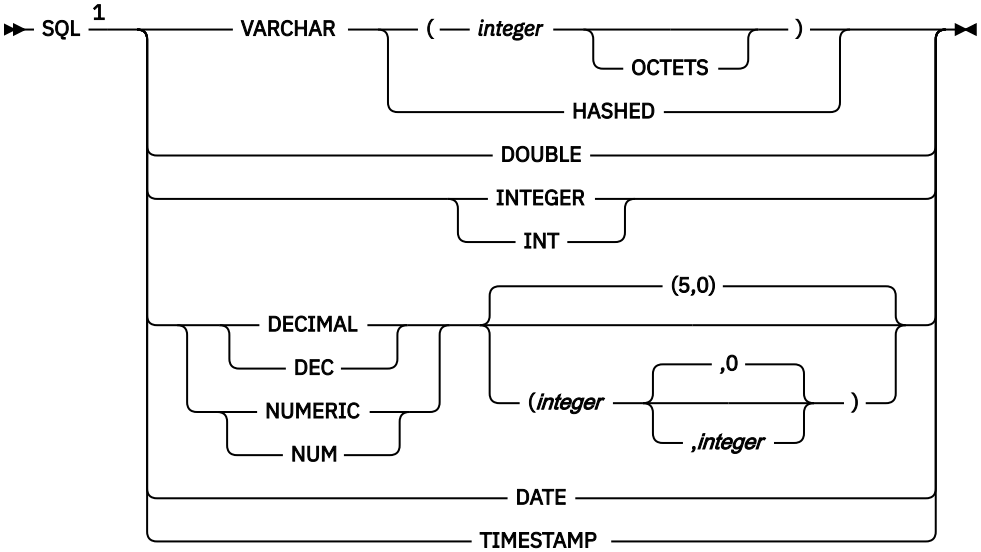
xmltype-clause



data-type



sql-data-type



Notes:

[1] If you specify a function name, such as fn:upper-case, at the end of the XML pattern, the supported index data types might be a subset of the index data types shown here. You can check for valid index data types in the description for xmlpattern-clause.

Description

UNIQUE

If ON *table-name* is specified, UNIQUE prevents the table from containing two or more rows with the same value of the index key. The uniqueness is enforced at the end of the SQL statement that updates rows or inserts new rows.

The uniqueness is also checked during the execution of the CREATE INDEX statement. If the table already contains rows with duplicate key values, the index is not created.

If the index is on an XML column (the index is an index over XML data), the uniqueness applies to values with the specified *pattern-expression* for all rows of the table. Uniqueness is enforced on each value after the value has been converted to the specified *sql-data-type*. Because converting to the specified *sql-data-type* might result in a loss of precision or range, or different values might be hashed to the same key value, multiple values that appear to be unique in the XML document might result in duplicate key errors. The uniqueness of character strings depends on XQuery semantics where trailing blanks are significant. Therefore, values that would be duplicates in SQL but differ in trailing blanks are considered unique values in an index over XML data.

When UNIQUE is used, null values are treated as any other values. For example, if the key is a single column that may contain null values, that column may contain no more than one null value.

If the UNIQUE option is specified, and the table has a distribution key, the columns in the index key must be a superset of the distribution key. That is, the columns specified for a unique index key must include all the columns of the distribution key (SQLSTATE 42997).

Primary or unique keys cannot be subsets of dimensions (SQLSTATE 429BE).

If ON *nickname* is specified, UNIQUE should be specified only if the data for the index key contains unique values for every row of the data source table. The uniqueness will not be checked.

For an index over XML data, UNIQUE can be included only if the context step of the *pattern-expression* specifies a single complete path and does not contain a descendant or descendant-or-self axis, "//", an *xml-wildcard*, *node()*, or *processing-instruction()* (SQLSTATE 429BS).

In a partitioned database environment, the following rules apply to a table with one or more XML columns:

- A distributed table cannot have a unique index over XML data.
- A unique index over XML data is supported only on a table that does not have a distribution key and that is on a single node multi-partition database.
- If a unique index over XML data exists on a table, the table cannot be altered to add a distribution key.

INDEX *index-name*

Names the index or index specification. The name, including the implicit or explicit qualifier, must not identify an index or index specification that is described in the catalog, or an existing index on a declared temporary table (SQLSTATE 42704). The qualifier must not be SYSIBM, SYSCAT, SYSFUN, or SYSSTAT (SQLSTATE 42939).

The implicit or explicit qualifier for indexes on declared temporary tables must be SESSION (SQLSTATE 428EK).

ON *table-name* or *nickname*

The *table-name* identifies a table on which an index is to be created. The table must be a base table (not a view), a created temporary table, a declared temporary table, a materialized query table that exists at the current server, or a declared temporary table. The name of a declared temporary table must be qualified with SESSION.

The *table-name* must not identify a catalog table (SQLSTATE 42832).

If UNIQUE is specified and *table-name* is a typed table, it must not be a subtable (SQLSTATE 429B3).

nickname is the nickname on which an index specification is to be created. The *nickname* references either a data source table whose index is described by the index specification, or a data source view that is based on such a table. The *nickname* must be listed in the catalog.

If the index key contains at least one *key-expression*, the *table-name* cannot be any of the following objects:

- A materialized query table (MQT) (SQLSTATE 429BX)
- A staging table (SQLSTATE 429BX)
- A typed table (SQLSTATE 429BX)
- A declared or created user temporary table (SQLSTATE 42995)
- A column-organized table (SQLSTATE 42858)
- A table that is an event monitor target (SQLSTATE 429BX)
- A nickname (SQLSTATE 42601)

column-name

For an index, *column-name* identifies a column that is to be part of the index key. For an index specification, *column-name* is the name by which the federated server references a column of a data source table.

The number of columns plus twice the number of identified periods cannot exceed 64 (SQLSTATE 54008). If *table-name* is a typed table, the number of columns cannot exceed 63 (SQLSTATE 54008). If *table-name* is a subtable, at least one *column-name* must be introduced in the subtable; that is, not inherited from a supertable (SQLSTATE 428DS). No *column-name* can be repeated (SQLSTATE 42711). The maximum number of columns in an index with random ordering is reduced by one for each column that is specified with random ordering

The sum of the stored lengths of the specified columns must not be greater than the index key length limit for the page size. For key length limits, see "SQL limits". If *table-name* is a typed table, the index key length limit is further reduced by 4 bytes. If the index has random ordering, the index key length is further reduced by 2 bytes.

Note that this length can be reduced by system overhead, which varies according to the data type of the column and whether it is nullable. For more information on overhead affecting this limit, see "Byte Counts" in "CREATE TABLE".

No LOB column or distinct type column based on a LOB can be used as part of an index, even if the length attribute of the column is small enough to fit within the index key length limit for the page size (SQLSTATE 54008). A structured type column can only be specified if the EXTEND USING clause is also specified (SQLSTATE 42962). If the EXTEND USING clause is specified, only one column can be specified, and the type of the column must be a structured type or a distinct type that is not based on a LOB (SQLSTATE 42997).

If an index has only one column, and that column has the XML data type, and the GENERATE KEY USING XMLPATTERN clause is also specified, the index is an index over XML data. A column with the XML data type can be specified only if the GENERATE KEY USING XMLPATTERN clause is also specified (SQLSTATE 42962). If the GENERATE KEY USING XMLPATTERN clause is specified, only one column can be specified, and the type of the column must be XML.

key-expression
Specifies an expression that must evaluate to a scalar value with the following restrictions:

- The expression must return a scalar value which can be indexed (no LOBs, XMLs, LONG VARCHAR or LONG VARGRAPHIC) (SQLSTATE 429BX)
- The following data types are not supported as input to the expression-based index key:
  - LONG VARCHAR and LONG VARGRAPHIC (deprecated data types)
  - XML
  - User defined distinct types on any of the types listed previously
  - User-defined weakly typed distinct types that include a data type constraint
  - User-defined structured types and reference types
  - Array, cursor, and row types
- The expression must contain at least one column reference (SQLSTATE 429BX)
- The expression cannot contain any of the following (SQLSTATE 429BX):
  - Subqueries
  - Aggregate functions
  - Non-deterministic functions
  - Functions with external actions
  - User-defined functions
  - Text search functions, such as SCORE, CONTAINS
  - Partitioning scalar functions, such as HASHEDVALUE
  - Dynamic data type scalar functions, such TYPE_ID, TYPE_NAME, TYPE_SCHEMA
  - Host Variables
  - Parameter markers
  - Sequence references
  - Special registers and built-in functions that depend on the value of a special register
  - Global variables and built-in functions that depend on the value of a global variable
  - A TYPE predicate
  - Regular expression functions or the REGEXP_LIKE predicate
  - A LIKE predicate
  - String scalar functions INSTR, INSTRB, LOCATE, LOCATE_IN_STRING, POSITION or POSSTR
  - OLAP specifications
  - Dereference operations or DEREF functions where the scoped reference argument is other than the object identifier (OID) column
  - CAST specifications with a SCOPE clause
  - Error tolerant nested-table-expressions

If an index key includes at least one *key-expression*, the index key is referred to as an expression-based index key.

ASC
Specifies that index entries are to be kept in ascending order of the column values; this is the default setting. ASC cannot be specified for indexes that are defined with EXTEND USING (SQLSTATE 42601).

DESC
Specifies that index entries are to be kept in descending order of the column values. DESC cannot be specified for indexes that are defined with EXTEND USING, or if the index is an index over XML data (SQLSTATE 42601).

RANDOM
Specifies that index entries are to be kept in random order of the column values. RANDOM cannot be specified in the following cases:

- With the EXTENDED USING clause (SQLSTATE 42613).
- With the SPECIFICATION ONLY clause (SQLSTATE 42613).
- For an index that is created on a declared or created globally temporary table (DGTT or CGTT) (SQLSTATE 42995).
- For an index that is created on a column-organized table (SQLSTATE 42858).
- If the CLUSTER option is specified (SQLSTATE 42613).
- On an indexed column that is of type CHAR or VARCHAR with ICU collations, except when the columns are declared as FOR BIT DATA (SQLSTATE 42997).
- On an indexed column that is of type GRAPHIC or VARGRAPHIC with ICU collations (SQLSTATE 42997).
- On an indexed column that is of type XML (SQLSTATE 42613).
- On an index which includes a *key-expression* (SQLSTATE 42997).

BUSINESS_TIME WITHOUT OVERLAPS
BUSINESS_TIME WITHOUT OVERLAPS can only be specified for an index defined as UNIQUE (SQLSTATE 428HW) to indicate that for the rest of the specified keys, the values are unique with respect to any period of time. BUSINESS_TIME WITHOUT OVERLAPS can only be specified as the last item in the list. When BUSINESS_TIME WITHOUT OVERLAPS is specified, the end column and begin column of the period BUSINESS_TIME are automatically added to the index key in ascending order and enforce that there are no overlaps in time. When BUSINESS_TIME WITHOUT OVERLAPS is specified, the columns of the BUSINESS_TIME period must not be specified as key columns, as columns in the partitioning key, or as columns in the distribution key (SQLSTATE 428HW).

PARTITIONED
Indicates that a partitioned index should be created. The *table-name* must identify a table defined with data partitions (SQLSTATE 42601).

If the table is partitioned and neither PARTITIONED nor NOT PARTITIONED is specified, the index is created as partitioned (with a few exceptions). A nonpartitioned index is created instead of a partitioned index if any of the following situations apply:

- UNIQUE is specified and the index key does not include all the table partitioning key columns.
- A spatial index is created.

A partitioned index with a definition that duplicates the definition of a nonpartitioned index is not considered to be a duplicate index. For more details, see the "Rules" on page 1257 section in this topic.

The PARTITIONED keyword cannot be specified for the following indexes:

- An index on a nonpartitioned table (SQLSTATE 42601)
- A unique index where the index key does not explicitly include all the table partitioning key columns (SQLSTATE 42990)
- A spatial index (SQLSTATE 42997)

A partitioned index cannot be created on a partitioned table that has detached dependent tables, for example, MQTs (SQLSTATE 55019).

The table space placement for an index partition of the partitioned index is determined by the following rules:

- If the table being indexed was created using the *partition-tablespace-options* INDEX IN clause of the CREATE TABLE statement, the index partition is created in the table space specified in that INDEX IN clause.
- If the CREATE TABLE statement for the table being indexed did not specify the *partition-tablespace-options* INDEX IN clause, the index partition partitioned index is created in the same table space as the corresponding data partition that it indexes.

The IN clause of the CREATE INDEX statement is not supported for partitioned indexes (SQLSTATE 42601). The *tablespace-clauses* INDEX IN clause of the CREATE TABLE statement is ignored for partitioned indexes. If BUSINESS_TIME WITHOUT OVERLAPS is specified for the index key, the partitioning key columns must not include the begin or end column of the BUSINESS_TIME period (SQLSTATE 428HW).

NOT PARTITIONED
Indicates that a nonpartitioned index should be created that spans all of the data partitions defined for the table. The *table-name* must identify a table defined with data partitions (SQLSTATE 42601).

A nonpartitioned index with a definition that duplicates the definition of a partitioned index is not considered to be a duplicate index. For more details, see the "Rules" on page 1257 section in this topic.

The table space placement for a the nonpartitioned index is determined by the following rules:

- If you specify the IN clause of the CREATE INDEX statement, the nonpartitioned index is placed in the table space specified in that IN clause.
- If you do not specify the IN clause of the CREATE INDEX statement, the following rules determine the table space placement of the nonpartitioned index:
  - If the table being indexed was created using the *tablespace-clauses* INDEX IN clause of the CREATE TABLE statement, the nonpartitioned index is placed in the table space specified in that INDEX IN clause.
  - If the table being indexed was created without using the *tablespace-clauses* INDEX IN clause of the CREATE TABLE statement, the nonpartitioned index is created in the table space of the first visible or attached data partition of the table. The first visible or attached data partition of the table is the first partition in the list of data partitions that are sorted on the basis of range specifications. Also, the authorization ID of the statement is not required to have the USE privilege on the default table space.

IN *tablespace-name*
Specifies the table space in which the nonpartitioned index on a partitioned table is created. This clause cannot be specified for a partitioned index or an index on a nonpartitioned table (SQLSTATE 42601). The specification of a table space specifically for the index overrides a specification made using the INDEX IN clause when the table was created.

The table space specified by *tablespace-name* must be in the same database partition group as the data table spaces for the table and manage space in the same way as the other table spaces of the partitioned table (SQLSTATE 42838); it must be a table space on which the authorization ID of the statement holds the USE privilege.

If the IN clause is not specified, the index is created in the table space that was specified by the INDEX IN clause on the CREATE TABLE statement. If no INDEX IN clause was specified, the table space of the first visible or attached data partition of the table is used. This is the first partition in the list of data partitions that are sorted on the basis of range specifications. If the IN clause is not specified, the authorization ID of the statement is not required to have the USE privilege on the default table space.

SPECIFICATION ONLY
Indicates that this statement will be used to create an index specification that applies to the data source table referenced by *nickname*. SPECIFICATION ONLY must be specified if *nickname* is specified (SQLSTATE 42601). It cannot be specified if *table-name* is specified (SQLSTATE 42601).

If the index specification applies to an index that is unique, the database manager does not verify that the column values in the remote table are unique. If the remote column values are not unique, queries against the nickname that include the index column might return incorrect data or errors.

This clause cannot be used when creating an index on a created temporary table or declared temporary table (SQLSTATE 42995).

INCLUDE
This keyword introduces a clause that specifies additional columns to be appended to the set of index key columns. Any columns included with this clause are not used to enforce uniqueness. These included columns might improve the performance of some queries through index only access. The columns must be distinct from the columns used to enforce uniqueness (SQLSTATE 42711). UNIQUE must be specified when INCLUDE is specified (SQLSTATE 42613). The limits for the number of columns and sum of the length attributes apply to all of the columns in the unique key and in the index.

This clause cannot be used with created temporary tables or declared temporary tables (SQLSTATE 42995).

This clause cannot be used with column-organized tables (SQLSTATE 42858).

column-name
Identifies a column that is included in the index but not part of the unique index key. The same rules apply as defined for columns of the unique index key. The keywords ASC, DESC, or RANDOM can be specified following *column-name* but have no effect on the order.

key-expression
Specifies an expression that is included in the index but not part of the unique index key. The same rules apply as defined for expressions of the unique index key. The keywords ASC, DESC, or RANDOM can be specified after *key-expression* but have no effect on the order.

INCLUDE cannot be specified for indexes that are defined with EXTEND USING, if *nickname* is specified, or if the index is defined on an XML column (SQLSTATE 42601).

xml-index-specification
Specifies how index keys are generated from XML documents that are stored in an XML column. *xml-index-specification* cannot be specified if there is more than one index column, or if the column does not have the XML data type.

This clause only applies to XML columns (SQLSTATE 429BS).

GENERATE KEY USING XMLPATTERN *xmlpattern-clause*
Specifies the parts of an XML document that are to be indexed. XML pattern values are the indexed values generated by the *xmlpattern-clause*. List data type nodes are not supported in the index. If a node is qualified by the *xmlpattern-clause* and an XML schema exists that specifies that the node is a list data type, then the list data type node cannot be indexed (SQLSTATE 23526 for CREATE INDEX statements, or SQLSTATE 23525 for INSERT and UPDATE statements).

xmlpattern-clause
Contains a pattern expression that identifies the nodes that are to be indexed. It consists of an optional *namespace-declaration* and a required *pattern-expression*.

namespace-declaration
If the pattern expression contains qualified names, a *namespace-declaration* must be specified to define namespace prefixes. A default namespace can be defined for unqualified names.

DECLARE NAMESPACE *namespace-prefix=namespace-uri*
Maps *namespace-prefix*, which is an NCName, to *namespace-uri*, which is a string literal. The *namespace-declaration* can contain multiple *namespace-prefix*-to-

namespace-uri mappings. The *namespace-prefix* must be unique within the list of *namespace-declaration* (SQLSTATE 10503).

DECLARE DEFAULT ELEMENT NAMESPACE *namespace-uri*
Declares the default namespace URI for unqualified element names or types. If no default namespace is declared, unqualified names of elements and types are in no namespace. Only one default namespace can be declared (SQLSTATE 10502).

pattern-expression
Specifies the nodes in an XML document that are indexed. The *pattern-expression* can contain pattern-matching characters (*). It is similar to a path expression in XQuery, but represents a subset of the XQuery language that this database supports.

/ (forward slash)
Separates path expression steps.

// (double forward slash)
This is the abbreviated syntax for /descendant-or-self::node()/. You cannot use *// (double forward slash)* if you also specify UNIQUE.

forward-axis

child::
Specifies children of the context node. This is the default, if no other forward axis is specified.

@
Specifies attributes of the context node. This is the abbreviated syntax for attribute::.

attribute::
Specifies attributes of the context node.

descendant::
Specifies the descendants of the context node. You cannot use *descendant::* if you also specify UNIQUE.

self::
Specifies just the context node itself.

descendant-or-self::
Specifies the context node and the descendants of the context node. You cannot use descendant-or-self:: if you also specify UNIQUE.

xmlname-test
Specifies the node name for the step in the path using a qualified XML name (xml-qname) or a wildcard (xml-wildcard).

xml-ncname
An XML name as defined by XML 1.0. It cannot include a colon character.

xml-qname
Specifies a qualified XML name (also known as a QName) that can have two possible forms:

- xml-nsprefix:xml-ncname, where the xml-nsprefix is an xml-ncname that identifies an in-scope namespace
- xml-ncname, which indicates that the default namespace should be applied as the implicit xml-nsprefix

xml-wildcard
Specifies an xml-qname as a wildcard that can have three possible forms:

- * (a single asterisk character) indicates any xml-qname
- *xml-nsprefix*:* indicates any xml-ncname within the specified namespace
- *:*xml-ncname* indicates a specific XML name in any in-scope namespace

You cannot use *xml-wildcard* in the context step of a pattern expression if you also specify UNIQUE.

xmlkind-test

Use these options to specify what types of nodes you pattern match. The following options are available to you:

node()

Matches any node. You cannot use *node()* if you also specify UNIQUE.

text()

Matches any text node.

comment()

Matches any comment node.

processing-instruction()

Matches any processing instruction node. You cannot use processing-instruction() if you also specify UNIQUE.

function-step

Use these function calls to specify indexes with special properties, such as case insensitivity. Only one function step is allowed per XMLPATTERN clause. Function steps can be applied only on elements or attributes. No *xmlkind-test* option can be placed immediately before the function step. The function cannot be used in the middle of the XMLPATTERN, and must appear only in the final step. Currently, only the fn:upper-case and fn:exists functions are supported.

Note that instead of specifying the prefix fn: for the function name, you can specify another valid namespace, or you can omit fn: entirely.

fn:upper-case

Force the index values to be stored in the uppercase form. The first parameter of fn:upper-case is mandatory, and must be a context item expression (' . '); the second parameter is optional, and is the locale. If fn:upper-case appears in the pattern, VARCHAR and VARCHAR HASHED are the only index types supported.

fn:exists

Check for the existence of an element or attribute item in the XML document. If the item exists, this predicate returns true. The parameter of fn:exists is mandatory, and must be an element or attribute. If this function is used in the index path, the index type must be defined as VARCHAR(1).

xmltype-clause

AS *data-type*

Specifies the data type to which indexed values are converted before they are stored. Values are converted to the index XML data type that corresponds to the specified index SQL data type.

Table 134. Corresponding index data types

Index XML data type	**Index SQL data type**
xs:string	VARCHAR(*integer*), VARCHAR HASHED
xs:double	DOUBLE
xs:int	INTEGER
xs:decimal	DECIMAL
xs:date	DATE
xs:dateTime	TIMESTAMP

For VARCHAR(*integer*) and VARCHAR HASHED, the value is converted to an xs:string value using the XQuery function fn:string. The length attribute of VARCHAR(*integer*) is applied as a constraint to the resulting xs:string value. An index SQL data type of VARCHAR HASHED applies a hash algorithm to the resulting xs:string value to generate a hash code that is inserted into the index.

For indexes using the data types DOUBLE, DATE, INTEGER, DECIMAL, and TIMESTAMP, the value is converted to the index XML data type using the XQuery cast expression.

If the index is unique, the uniqueness of the value is enforced after the value is converted to the indexed type.

data-type
The following data type is supported:

sql-data-type
Supported SQL data types are:

VARCHAR(*integer*[OCTETS])
If this form of VARCHAR is specified, *integer* is used as a constraint. If document nodes that are to be indexed have values that are longer than *integer*, the documents are not inserted into the table if the index already exists. If the index does not exist, the index is not created. *integer* is a value between 1 and a page size-dependent maximum. Table 135 on page 1252 shows the maximum value for each page size.

Table 135. Maximum length of document nodes by page size

Page size	Maximum length of document node (bytes)
4KB	817
8KB	1841
16KB	3889
32KB	7985

XQuery semantics are used for string comparisons, where trailing blanks are significant. This differs from SQL semantics, where trailing blanks are insignificant during comparisons.

OCTETS
Specifies that the units of the length attribute is bytes.

When no string units are specified for a character string data type in a Unicode database, the string units are implicit and determined by the value of the NLS_STRING_UNITS global variable or *string_units* database configuration parameter. When the implicit string units are CODEUNITS32, the OCTETS keyword must be specified (SQLSTATE 42601).

In a non-Unicode database, the string units for character string data types are OCTETS.

VARCHAR HASHED
Specify VARCHAR HASHED to handle indexing of arbitrary length character strings. The length of an indexed string has no limit. An eight-byte hash code is generated over the entire string. Indexes that use these hashed character strings can be used only for equality lookups. XQuery semantics are used for string equality comparisons, where trailing blanks are significant. This differs from SQL semantics, where trailing blanks are insignificant during comparisons. The hash on the string preserves XQuery semantics for equality and not SQL semantics.

DOUBLE
Specifies that the data type DOUBLE is used for indexing numeric values. Unbounded decimal types and 64 bit integers may lose precision when they are stored as a DOUBLE value. The values for DOUBLE may include the special numeric values *NaN*, *INF*, *-INF*, *+0*, and *-0*, even though the SQL data type DOUBLE itself does not support these values.

INTEGER
Specifies that the data type INTEGER is used for indexing XML values. Note that the XML schema data type xs:integer allows a greater range of values than does the integer SQL data type. If an out-of-range value is encountered, an error is returned. If a value conforms to the lexical format of xs:double but does not conform to the lexical format of xs:int, such as 3.5, 3.0, or 3E1, an error is also returned.

DECIMAL(*integer*, *integer*)
Specifies that the data type DECIMAL is used for indexing XML values. The DECIMAL type takes two parameters, *precision* and *scale*. The first parameter, *precision*, is an integer constant with a value in the range of 1 to 31 that specifies the total number of digits. The second parameter, *scale*, is an integer constant that is greater than or equal to zero, and less than or equal to precision. The *scale* specifies the number of digits to the right of the decimal point.

Digits are not truncated from the end of a decimal number. An error is returned if the number of digits to the right of the decimal separator character is greater than the scale. Also, an error is returned if the number of significant digits to the left of the decimal character (the whole part of the number) is greater than precision.

DATE
Specifies that the data type DATE is used for indexing XML values. Note that the XML schema data type for xs:date allows greater range of values than the pureXML® xs:date data type that corresponds to the SQL data type. If an out-of-range value is encountered, an error is returned.

TIMESTAMP
Specifies that the data type TIMESTAMP is used for indexing XML values. Note that the XML schema data type for xs:dateTime allows greater range of values and fractional seconds precision than the pureXML xs:dateTime data type that corresponds to the SQL data type. If an out-of range value is encountered, an error is returned.

IGNORE INVALID VALUES
Specifies that XML pattern values that are invalid lexical forms for the target index XML data type are ignored and that the corresponding values in the stored XML documents are not indexed by the CREATE INDEX statement. By default, invalid values are ignored. During insert and update operations, the invalid XML pattern values are not indexed, but XML documents are still inserted into the table. No error or warning is raised, because specifying these data types is not a constraint on the XML pattern values (XQuery expressions that search for the specific XML index data type will not consider these values).

The rules for what XML pattern values can be ignored are determined by the specified SQL data type.

- If the SQL data type is VARCHAR(*integer*) or VARCHAR HASHED, XML pattern values are never ignored since any sequence of characters is valid.
- If the SQL data type is DOUBLE, DECIMAL, or INTEGER, any XML pattern value that does not conform to the lexical format of the XML data type xs:double is ignored. If the SQL data type is DECIMAL or INTEGER and the XML pattern value conforms to the lexical format of the XML data type xs:double but not to the lexical format of xs:decimal or

xs:int, respectively, an error is returned. For example, if the SQL data type is INTEGER, the XML pattern values of 3.5, 3.0, and 3e0 conform to the lexical format of xs:double but return an error (SQLSTATE 23525) because they do not conform to the lexical format of xs:int. XML pattern values such as 'A123' or 'hello' are ignored for the same index.

- If the SQL data type is a datetime data type, any XML pattern value that does not conform to the lexical format of the corresponding XML data type (xs:date or xs:dateTime) is ignored.

If an XML pattern value does conform to the appropriate lexical format, an error is returned if the value is outside the value space for the data type or exceeds the maximum length or precision and scale of the specified SQL data type. If the index does not exist, the index is not created (SQLSTATE 23526).

REJECT INVALID VALUES
All XML pattern values must be valid in the context of the lexical definition of the index XML data type. In addition the value must be in the range of the value space of the index XML data type. See the Related reference section, later, for links to details on the lexical definition and value space for each data type. For example, when you specify the REJECT INVALID VALUES clause, if you create an index of INTEGER type, XML pattern values such as 3.5, 3.0, 3e0, 'A123' and 'hello' will return an error (SQLSTATE 23525). XML data is not inserted or updated in the table if the index already exists (SQLSTATE 23525). If the index does not exist, the index is not created (SQLSTATE 23526).

CLUSTER
Specifies that the index is the clustering index of the table. The cluster factor of a clustering index is maintained or improved dynamically as data is inserted into the associated table, by attempting to insert new rows physically close to the rows for which the key values of this index are in the same range. Only one clustering index may exist for a table so CLUSTER may not be specified if it was used in the definition of any existing index on the table (SQLSTATE 55012). A clustering index may not be created on a table that is defined to use append mode (SQLSTATE 428D8).

CLUSTER is disallowed if *nickname* is specified, or if the index is an index over XML data (SQLSTATE 42601).

This clause cannot be used with the following types of tables:

- Created temporary tables or declared temporary tables (SQLSTATE 42995)
- Range-clustered tables (SQLSTATE 429BG)
- Column-organized tables (SQLSTATE 42858)

EXTEND USING *index-extension-name*
Names the *index-extension* used to manage this index. If this clause is specified, then there must be only one *column-name* specified and that column must be a structured type or a distinct type (SQLSTATE 42997). The *index-extension-name* must name an index extension described in the catalog (SQLSTATE 42704). For a distinct type, the column must exactly match the type of the corresponding source key parameter in the index extension. For a structured type column, the type of the corresponding source key parameter must be the same type or a supertype of the column type (SQLSTATE 428E0).

This clause cannot be used with created temporary tables or declared temporary tables (SQLSTATE 42995).

This clause cannot be used with column-organized tables (SQLSTATE 42858).

Starting with IBM Db2 10.5, this clause is also supported in Db2 pureScale environments. For version 10.5 Fix Pack 3 and earlier fix pack releases, this clause is not supported in Db2 pureScale environments (SQLSTATE 56038).

This clause cannot be used if the index key contains at least one *key-expression* (SQLSTATE 42601).

constant-expression
Identifies values for any required arguments for the index extension. Each expression must be a constant value with a data type that exactly matches the defined data type of the corresponding

index extension parameters, including length or precision, and scale (SQLSTATE 428E0). This clause must not exceed 32 768 bytes in length in the database code page (SQLSTATE 22001).

PCTFREE *integer*
Specifies what percentage of each index page to leave as free space when building the index. The first entry in a page is added without restriction. When additional entries are placed in an index page at least *integer* percent of free space is left on each page. The value of *integer* can range from 0 to 99. If a value greater than 10 is specified, only 10 percent free space will be left in non-leaf pages.

If an explicit value for PCTFREE is not provided, and if **DB2_INDEX_PCTFREE_DEFAULT** is not set, then PCTFREE will have a default value of 10.

PCTFREE is disallowed if *nickname* is specified (SQLSTATE 42601). This clause cannot be used with created temporary tables or declared temporary tables (SQLSTATE 42995).

LEVEL2 PCTFREE *integer*
Specifies what percentage of each index level 2 page to leave as free space when building the index. The value of *integer* can range from 0 to 99. If LEVEL2 PCTFREE is not set, a minimum of 10 or PCTFREE percent of free space is left on all non-leaf pages. If LEVEL2 PCTFREE is set, *integer* percent of free space is left on level 2 intermediate pages, and a minimum of 10 or *integer* percent of free space is left on level 3 and higher intermediate pages.

LEVEL2 PCTFREE is disallowed if *nickname* is specified (SQLSTATE 42601). This clause cannot be used with created temporary tables or declared temporary tables (SQLSTATE 42995).

MINPCTUSED *integer*
Indicates whether index leaf pages are merged online, and the threshold for the minimum percentage of space used on an index leaf page. If, after a key is removed from an index leaf page, the percentage of space used on the page is at or below *integer* percent, an attempt is made to merge the remaining keys on this page with those of a neighboring page. If there is sufficient space on one of these pages, the merge is performed and one of the pages is deleted. The value of *integer* can be from 0 to 99. A value of 50 or below is recommended for performance reasons. Specifying this option will have an impact on update and delete performance. Merging is only done during update and delete operations when an exclusive table lock is held. If an exclusive table lock does not exist, keys are marked as pseudo deleted during update and delete operations, and no merging is done. Consider using the CLEANUP ONLY ALL option of REORG INDEXES to merge leaf pages instead of using the MINPCTUSED option of CREATE INDEX.

MINPCTUSED is disallowed if *nickname* is specified (SQLSTATE 42601). This clause cannot be used with created temporary tables or declared temporary tables (SQLSTATE 42995).

DISALLOW REVERSE SCANS
Specifies that an index only supports forward scans or scanning of the index in the order that was defined at index creation time.

DISALLOW REVERSE SCANS cannot be specified together with *nickname* (SQLSTATE 42601).

ALLOW REVERSE SCANS
Specifies that an index can support both forward and reverse scans; that is, scanning of the index in the order that was defined at index creation time, and scanning in the opposite order.

ALLOW REVERSE SCANS cannot be specified together with *nickname* (SQLSTATE 42601).

PAGE SPLIT
Specifies the page split behavior when values are inserted into an index. The default is SYMMETRIC.

SYMMETRIC
Specifies that pages are to be split roughly in the middle. Use this option in the following situations:

- When the insertion into an index is random
- When the insertion into an index does not follow the patterns that are addressed by the PAGE SPLIT HIGH and PAGE SPLIT LOW options

HIGH
Specifies an index page split behavior that uses the space on index pages efficiently when there are ever-increasing values in the index. Increasing values in the index might occur when the following conditions are met:

- There is an index with multiple key parts and there are multiple index pages of values where all except the last key part has the same value.
- All insert operations into the table consist of a new value, which has the same value as existing keys for all but the last key part.
- The last key part of the inserted value is larger than the values of the existing keys.

For example, assume that an index has the following key values:

```
(1,1),(1,2),(1,3), ... (1,n),
(2,1),(2,2),(2,3), ... (2,n),
...
(m,1),(m,2),(m,3), ... (m,n)
```

The next key to be inserted would have the value `(x,y)` where `1 <= x <= m` and `y > n`.

In such cases, use the PAGE SPLIT HIGH clause so that page splits do not result in many pages that are 50 percent empty.

LOW
Specifies an index page split behavior that uses the space on index pages efficiently when there are ever-decreasing values in the index. Decreasing values in the index might occur when the following conditions are met:

- There is an index with multiple key parts and there are multiple index pages of values where all except the last key part has the same value.
- All insert operations into the table consist of a new value, which has the same value as existing keys for all but the last key part.
- The last key part of the inserted value is smaller than the values of the existing keys.

In such cases, use the PAGE SPLIT LOW clause so that page splits do not result in many pages that are 50 percent empty.

COLLECT STATISTICS
Specifies that basic index statistics are to be collected during index creation.

SAMPLED
Specifies that a sampling technique is to be used when processing index entries to collect extended index statistics. This option is used to balance performance considerations with the need for accuracy of the statistics. This option is the default when DETAILED is specified immediately following the keyword COLLECT.

UNSAMPLED
Specifies that sampling is not to be used when processing index entries to collect extended index statistics. Instead, each index entry is examined individually. This option can significantly increase CPU and memory consumption.

DETAILED
Specifies that extended index statistics (CLUSTERFACTOR and PAGE_FETCH_PAIRS) are also to be collected during index creation.

COMPRESS
Specifies whether index compression is enabled. By default, index compression will be enabled if data row compression is enabled or if the table is a declared global temporary table (DGTT) or created global temporary table (CGTT); index compression will be disabled if data row compression is disabled. This option can be used to override the default behavior. COMPRESS is disallowed if *nickname* is specified (SQLSTATE 42601).

YES
Specifies that index compression is enabled. Insert and update operations on the index will be subject to compression.

NO
Specifies that index compression is disabled.

INCLUDE NULL KEYS
Specifies that an index entry is created when all parts of the index key contain the null value. This is the default setting.

EXCLUDE NULL KEYS
Specifies that an index entry is not created when all parts of the index key contain the null value. When any part of the index key is not a null value, an index entry is created. You cannot specify **EXCLUDE NULL KEYS** with the following syntax elements:

- A nickname
- The **GENERATE KEY USING XMLPATTERN** clause
- The **EXTEND USING** clause.

If an index is defined as unique, rows with null keys are not considered when enforcing uniqueness.

This clause cannot be used with column-organized tables (SQLSTATE 42858).

Rules

- The CREATE INDEX statement fails (SQLSTATE 01550) when attempting to create an index that matches an existing index.

  A number of factors are used to determine if two indexes match. These factors are combined in various different ways into the rules that determine if two indexes match. The following factors are used to determine if two indexes match:

  1. The sets of index columns and key expressions, including any INCLUDE columns and key expressions, are the same in both indexes.
  2. The ordering of index key columns and key expressions, including any INCLUDE columns, is the same in both indexes.
  3. The key columns and key expressions of the new index are the same or a superset of the key columns and key expressions in the existing index.
  4. The ordering attributes of the columns and key expressions are the same in both indexes.
  5. The existing index is unique.
  6. Both indexes are non-unique.

  The following combinations of these factors form the rules that determine when two indexes are considered duplicates:

  – "1" on page 1257 + "2" on page 1257 + "4" on page 1257 + "5" on page 1257
  – "1" on page 1257 + "2" on page 1257 + "4" on page 1257 + "6" on page 1257
  – "1" on page 1257 + "2" on page 1257 + "3" on page 1257 + "5" on page 1257

  Exceptions:

  – If one of the compared indexes is partitioned and the other of the compared indexes is nonpartitioned, the indexes are not considered duplicates if the indexes have different names, even if other matching index conditions are met.
  – For indexes over XML data, the index descriptions are not considered duplicates if the index names are different, even if the indexed XML column, the XML patterns, and the data type, including its options, are identical.
- Unique indexes on system-maintained MQTs are not supported (SQLSTATE 42809).
- The COLLECT STATISTICS options are not supported if a nickname is specified (SQLSTATE 42601).

- The creation of an index with an expression-based key in a partitioned database environment is supported only from the catalog database partition (SQLSTATE 42997).
- **Restrictions for indexes on column-organized tables:**
  - The following clauses are not supported when creating an index on a column-organized table (SQLSTATE 42858):
    - RANDOM
    - CLUSTER
    - EXTEND USING
    - INCLUDE
    - EXCLUDE NULL KEYS
    - *key-expression*
- **Indexes cannot be created on column-organized temporary tables.**

Notes

- Concurrent read/write access during index creation, and default index creation behavior differs for indexes on nonpartitioned tables, nonpartitioned indexes, partitioned indexes, and indexes in a Db2 pureScale environment:
  - For nonpartitioned indexes, concurrent read/write access to the table is permitted while an index is being created, except when the EXTEND USING clause is specified. Once the index has been built, changes that were made to the table during index creation time are forward-fitted to the new index. Write access to the table is then blocked while index creation completes, after which the new index becomes available.
  - For partitioned indexes, concurrent read/write access to the table is permitted while an index is being created, except when the EXTEND USING clause is specified. Once the index partition has been built, changes that were made to the partition during creation time of that index partition are forward-fitted to the new index partition. Write access to the data partition is then blocked while index creation completes on the remaining data partitions. After the index partition for the last data partition is built and the transaction is committed, all data partitions are available for read and write.
  - Prior to Version 11.5, in a Db2 pureScale environment, concurrent read access during index creation is the default behavior. You can enable concurrent write access by setting the registry variable *DB2_INDEX_CREATE_ALLOW_WRITE* to ON. For more information, see **DB2_INDEX_CREATE_ALLOW_WRITE**. Starting in Version 11.5, in a Db2 pureScale environment, concurrent write access is enabled by default.

  CREATE INDEX tries to forward-fit the concurrent changes before blocking writers. However, if there is a lot of concurrent database activity such that CREATE INDEX is not able to keep up with the changes coming in, it will block new writers sooner so it can complete the forward-fit. CREATE INDEX does as much of the create completion work as it can before writers are blocked, to keep the period of unavailability as short as possible. The size of the index and the amount of current activity impact this period of time.

  To circumvent this default behavior, use the LOCK TABLE statement to explicitly lock the table before issuing a CREATE INDEX statement. (The table can be locked in either SHARE or EXCLUSIVE mode, depending on whether read access is to be allowed.)
- If the named table already contains data, CREATE INDEX creates the index entries for it. If the table does not yet contain data, CREATE INDEX creates a description of the index; the index entries are created when data is inserted into the table.
- Once the index is created and data is loaded into the table, it is advisable to issue the RUNSTATS command. The RUNSTATS command updates statistics collected on the database tables, columns, and indexes. These statistics are used to determine the optimal access path to the tables. By issuing the RUNSTATS command, the database manager can determine the characteristics of the new index. If data has been loaded before the CREATE INDEX statement is issued, it is recommended that the COLLECT

STATISTICS option on the CREATE INDEX statement be used as an alternative to the RUNSTATS command.

- If you collect statistics during index creation, the resulting statistics might be inconsistent. If the table has been modified since you last collected statistics on the table and its existing indexes, you should subsequently run the RUNSTATS command to provide a set of consistent statistics across the table and all of its indexes.
- Creating an index with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- The optimizer can recommend indexes before creating the actual index.
- If an index specification is being defined for a data source table that has an index, the name of the index specification does not have to match the name of the index.
- The explain facility, the Design Advisor, or the EXPLAIN option in Data Server Manager can be used to recommend indexes before creating the actual index. However, none of these methods will recommend indexes containing expression-based keys.
- ***Collecting index statistics***: The UNSAMPLED DETAILED option is available to change the way index statistics are collected. However, it should be used only in cases where it is clear that DETAILED does not yield accurate statistics.
- ***Generated Objects***: If an index is created with expression-based keys, a system-generated statistical view and a system-generated package will also be created and associated with the index.
- ***Syntax alternatives***: The following syntax is tolerated and ignored:
  - CLOSE
  - DEFINE
  - FREEPAGE
  - GBPCACHE
  - PIECESIZE
  - TYPE 2
  - using-block

  The following syntax is accepted as the default behavior:
  - COPY NO
  - DEFER NO

Examples

- *Example 1:* Create an index named UNIQUE_NAM on the PROJECT table. The purpose of the index is to ensure that there are not two entries in the table with the same value for project name (PROJNAME). The index entries are to be in ascending order.

```
CREATE UNIQUE INDEX UNIQUE_NAM
   ON PROJECT(PROJNAME)
```

- *Example 2:* Create an index named JOB_BY_DPT on the EMPLOYEE table. Arrange the index entries in ascending order by job title (JOB) within each department (WORKDEPT).

```
CREATE INDEX JOB_BY_DPT
   ON EMPLOYEE (WORKDEPT, JOB)
```

- *Example 3:* The nickname EMPLOYEE references a data source table called CURRENT_EMP. After this nickname was created, an index was defined on CURRENT_EMP. The columns chosen for the index key were WORKDEBT and JOB. Create an index specification that describes this index. Through this

specification, the optimizer will know that the index exists and what its key is. With this information, the optimizer can improve its strategy to access the table.

```
CREATE UNIQUE INDEX JOB_BY_DEPT
  ON EMPLOYEE (WORKDEPT, JOB)
  SPECIFICATION ONLY
```

- *Example 4:* Create an extended index type named SPATIAL_INDEX on a structured type column location. The description in index extension GRID_EXTENSION is used to maintain SPATIAL_INDEX. The literal is given to GRID_EXTENSION to create the index grid size.

```
CREATE INDEX SPATIAL_INDEX ON CUSTOMER (LOCATION)
  EXTEND USING (GRID_EXTENSION (x'000100100010001000400010'))
```

- *Example 5:* Create an index named IDX1 on a table named TAB1, and collect basic index statistics on index IDX1.

```
CREATE INDEX IDX1 ON TAB1 (col1) COLLECT STATISTICS
```

- *Example 6:* Create an index named IDX2 on a table named TAB1, and collect detailed index statistics on index IDX2.

```
CREATE INDEX IDX2 ON TAB1 (col2) COLLECT DETAILED STATISTICS
```

- *Example 7:* Create an index named IDX3 on a table named TAB1, and collect detailed index statistics on index IDX3 using sampling.

```
CREATE INDEX IDX3 ON TAB1 (col3) COLLECT SAMPLED DETAILED STATISTICS
```

- *Example 8:* Create a unique index named A_IDX on a partitioned table named MYNUMBERDATA in table space IDX_TBSP.

```
CREATE UNIQUE INDEX A_IDX ON MYNUMBERDATA (A) IN IDX_TBSP
```

- *Example 9:* Create a non-unique index named B_IDX on a partitioned table named MYNUMBERDATA in table space IDX_TBSP.

```
CREATE INDEX B_IDX ON MYNUMBERDATA (B)
  NOT PARTITIONED IN IDX_TBSP
```

- *Example 10:* Create an index over XML data on a table named COMPANYINFO, which contains an XML column named COMPANYDOCS. The XML column COMPANYDOCS contains a large number of XML documents similar to the one below:

```
<company name="Company1">
  <emp id="31201" salary="60000" gender="Female">
    <name>
      <first>Laura</first>
      <last>Brown</last>
    </name>
    <dept id="M25">
      Finance
    </dept>
  </emp>
</company>
```

Users of the COMPANYINFO table often need to retrieve employee information using the employee ID. An index like the following one can make that retrieval more efficient.

```
CREATE INDEX EMPINDEX ON COMPANYINFO(COMPANYDOCS)
  GENERATE KEY USING XMLPATTERN '/company/emp/@id'
    AS SQL DOUBLE
```

- *Example 11:* The following index is logically equivalent to the index created in the previous example, except that it uses unabbreviated syntax.

```
CREATE INDEX EMPINDEX ON COMPANYINFO(COMPANYDOCS)
  GENERATE KEY USING XMLPATTERN '/child::company/child::emp/attribute::id'
    AS SQL DOUBLE
```

- *Example 12:* Create an index on a column named DOC, indexing only the book title as a VARCHAR(100). Because the book title should be unique across all books, the index must be unique.

```
CREATE UNIQUE INDEX MYDOCSIDX ON MYDOCS(DOC)
  GENERATE KEY USING XMLPATTERN '/book/title'
    AS SQL VARCHAR(100)
```

- *Example 13:* Create an index on a column named DOC, indexing the chapter number as a DOUBLE. This example includes namespace declarations.

```
CREATE INDEX MYDOCSIDX ON MYDOCS(DOC)
  GENERATE KEY USING XMLPATTERN
    'declare namespace b="http://www.example.com/book/";
     declare namespace c="http://acme.org/chapters";
       /b:book/c:chapter/@number'
    AS SQL DOUBLE
```

- *Example 14:* Create a unique index named IDXPROJEST on table PROJECT and include column PRSTAFF to allow index-only access of the estimated mean staffing information.

```
CREATE UNIQUE INDEX IDXPROJEST ON PROJECT (PROJNO) INCLUDE (PRSTAFF)
```

- *Example 15:* Create a unique index on a column named USER_ID and exclude null keys from that index.

```
CREATE UNIQUE INDEX IDXUSERID ON CUSTOMER (USER_ID) EXCLUDE NULL KEYS
```

- *Example 16:* Create an index with an expression-based key using upper case of employee's name and ID:

```
CREATE INDEX EMP_UPPERNAME ON EMPLOYEE (UPPER(NAME), ID)
```

CREATE INDEX EXTENSION

The CREATE INDEX EXTENSION statement defines an extension object for use with indexes on tables that have structured type or distinct type columns.

Invocation

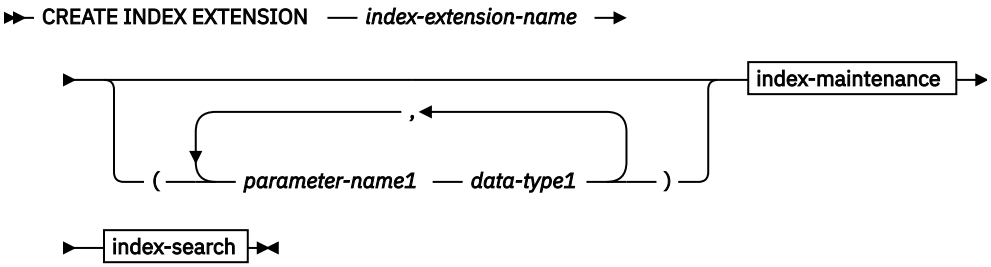
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

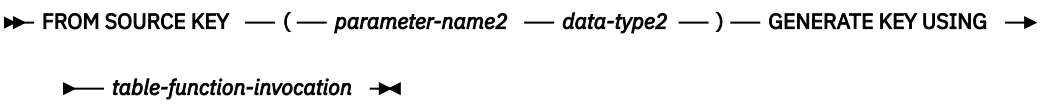
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- IMPLICIT_SCHEMA authority on the database, if the schema name of the index extension does not refer to an existing schema
- CREATEIN privilege on the schema, if the schema name of the index extension refers to an existing schema
- SCHEMAADM authority on the schema, if the schema name of the index extension refers to an existing schema
- DBADM authority

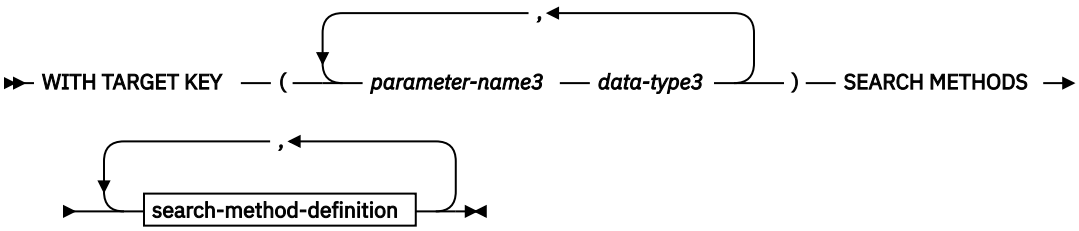
Syntax



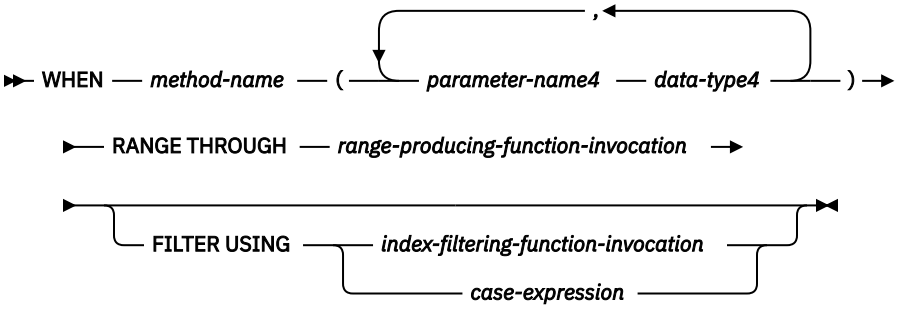
index-maintenance



index-search



search-method-definition



Description

index-extension-name

Names the index extension. The name, including the implicit or explicit qualifier, must not identify an index extension described in the catalog. If a two-part *index-extension-name* is specified, the schema name cannot begin with 'SYS'; otherwise, an error is returned (SQLSTATE 42939).

parameter-name1

Identifies a parameter that is passed to the index extension at CREATE INDEX time to define the actual behavior of this index extension. The parameter that is passed to the index extension is called an *instance parameter*, because that value defines a new instance of an index extension.

parameter-name1 must be unique within the definition of the index extension. No more than 90 parameters are allowed. If this limit is exceeded, an error (SQLSTATE 54023) is returned.

data-type1

Specifies the data type of each parameter. One entry in the list must be specified for each parameter that the index extension will expect to receive. The only SQL data types that can be specified are those that can be used as constants, such as VARCHAR, INTEGER, DECIMAL, DOUBLE, or VARGRAPHIC (SQLSTATE 429B5). The decimal floating-point data type cannot be specified (SQLSTATE 429B5). The parameter value that is received by the index extension at

CREATE INDEX must match *data-type1* exactly, including length, precision, and scale (SQLSTATE 428E0).

Character and graphic string data types cannot specify string units of CODEUNITS32.

index-maintenance
Specifies how the index keys of a structured or distinct type column are maintained. Index maintenance is the process of transforming the source column to a target key. The transformation process is defined using a table function that has previously been defined in the database.

FROM SOURCE KEY (*parameter-name2 data-type2*)
Specifies a structured data type or distinct type for the source key column that is supported by this index extension.

parameter-name2
Identifies the parameter that is associated with the source key column. A source key column is the index key column (defined in the CREATE INDEX statement) with the same data type as *data-type2*.

data-type2
Specifies the data type for *parameter-name2*; *data-type2* must be a user-defined structured type or a distinct type that is not sourced on LOB, XML, or DECFLOAT (SQLSTATE 42997). When the index extension is associated with the index at CREATE INDEX time, the data type of the index key column must:

- Exactly match *data-type2* if it is a distinct type; or
- Be the same type or a subtype of *data-type2* if it is a structured type

Otherwise, an error is returned (SQLSTATE 428E0).

GENERATE KEY USING *table-function-invocation*
Specifies how the index key is generated using a user-defined table function. Multiple index entries may be generated for a single source key data value. An index entry cannot be duplicated from a single source key data value (SQLSTATE 22526). The function can use *parameter-name1*, *parameter-name2*, or a constant as arguments. If the data type of *parameter-name2* is a structured data type, only the observer methods of that structured type can be used in its arguments (SQLSTATE 428E3). The output of the GENERATE KEY function must be specified in the TARGET KEY specification. The output of the function can also be used as input for the index filtering function specified on the FILTER USING clause.

The function used in *table-function-invocation* must:

- Resolve to a table function (SQLSTATE 428E4)
- Not be defined with PARAMETER CCSID UNICODE if this database is not a Unicode database (SQLSTATE 428E4)
- Not be defined with LANGUAGE SQL (SQLSTATE 428E4)
- Not be defined with NOT DETERMINISTIC (SQLSTATE 428E4) or EXTERNAL ACTION (SQLSTATE 428E4)
- Be defined with NO SQL (SQLSTATE 428E4)
- Not have a structured data type, LOB or XML (SQLSTATE 428E3) in the data type of the parameters, with the exception of system-generated observer methods
- Not include a subquery (SQLSTATE 428E3)
- Not include an XMLQUERY or XMLEXISTS expression (SQLSTATE 428E3)
- Return columns with data types that follow the restrictions for data types of columns of an index defined without the EXTEND USING clause

If an argument invokes another operation or routine, it must be an observer method (SQLSTATE 428E3).

The definer of the index extension must have EXECUTE privilege on the function, EXECUTEIN privilege on the schema containing this function, or DATAACCESS authority on the schema containing this function.

index-search
Specifies how searching is performed by providing a mapping of the search arguments to search ranges.

WITH TARGET KEY
Specifies the target key parameters that are the output of the key generation function specified on the GENERATE KEY USING clause.

parameter-name3
Identifies the parameter associated with a given target key. *parameter-name3* corresponds to the columns of the RETURNS table as specified in the table function of the GENERATE KEY USING clause. The number of parameters specified must match the number of columns returned by that table function (SQLSTATE 428E2).

data-type3
Specifies the data type for each corresponding *parameter-name3*. *data-type3* must exactly match the data type of each corresponding output column of the RETURNS table, as specified in the table function of the GENERATE KEY USING clause (SQLSTATE 428E2), including the length, precision, and type.

SEARCH METHODS
Introduces the search methods that are defined for the index.

search-method-definition
Specifies the method details of the index search. It consists of a method name, the search arguments, a range producing function, and an optional index filter function.

WHEN *method-name*
The name of a search method. This is an SQL identifier that relates to the method name specified in the index exploitation rule (found in the PREDICATES clause of a user-defined function). A *search-method-name* can be referenced by only one WHEN clause in the search method definition (SQLSTATE 42713).

parameter-name4
Identifies the parameter of a search argument. These names are for use in the RANGE THROUGH and FILTER USING clauses.

data-type4
The data type associated with a search parameter.

RANGE THROUGH *range-producing-function-invocation*
Specifies an external table function that produces search ranges. This function uses *parameter-name1*, *parameter-name4*, or a constant as arguments and returns a set of search ranges.

The table function used in *range-producing-function-invocation* must:

- Resolve to a table function (SQLSTATE 428E4)
- Not include a subquery (SQLSTATE 428E3) or SQL function (SQLSTATE 428E4) in its arguments
- Not include an XMLQUERY or XMLEXISTS expression in its arguments (SQLSTATE 428E3)
- Not be defined with PARAMETER CCSID UNICODE if this database is not a Unicode database (SQLSTATE 428E4)
- Not be defined with LANGUAGE SQL (SQLSTATE 428E4)
- Not be defined with NOT DETERMINISTIC or EXTERNAL ACTION (SQLSTATE 428E4)
- Be defined with NO SQL (SQLSTATE 428E4)

The number and types of this function's results must relate to the results of the table function specified in the GENERATE KEY USING clause (SQLSTATE 428E1) by:

- Returning up to twice as many columns as returned by the key transformation function

- Having an even number of columns, in which the first half of the return columns defines the start of the range (start key values), and the second half of the return columns defines the end of the range (stop key values)
- Having each start key column with the same type as the corresponding stop key column
- Having the type of each start key column be the same as the corresponding key transformation function column

More precisely, let $a_1:t_1$, ..., $a_n:t_n$ be the function result columns and data types of the key transformation function. The function result columns of the *range-producing-function-invocation* must be $b_1:t_1$, ..., $b_m:t_m$, $c_1:t_1$, ..., $c_m:t_m$, where m <= n and the "b" columns are the start key columns and the "c" columns are the stop key columns.

When the *range-producing-function-invocation* returns a null value as the start or stop key value, the semantics are undefined.

The definer of the index extension must have EXECUTE privilege on the function, EXECUTEIN privilege on the schema containing this function, or DATAACCESS authority on the schema containing this function.

FILTER USING

Allows specification of an external function or a case expression to be used for filtering index entries that were returned after applying the range-producing function.

index-filtering-function-invocation

Specifies an external function to be used for filtering index entries. This function uses the *parameter-name1*, *parameter-name3*, *parameter-name4*, or a constant as arguments (SQLSTATE 42703) and returns an integer (SQLSTATE 428E4). If the value returned is 1, the row corresponding to the index entry is retrieved from the table. Otherwise, the index entry is not considered for further processing.

If not specified, index filtering is not performed.

The function used in the *index-filtering-function-invocation* must:

- Not be defined with PARAMETER CCSID UNICODE if this database is not a Unicode database (SQLSTATE 428E4)
- Not be defined with LANGUAGE SQL (SQLSTATE 429B4)
- Not be defined with NOT DETERMINISTIC or EXTERNAL ACTION (SQLSTATE 42845)
- Be defined with NO SQL (SQLSTATE 428E4)
- Not have a structured data type in the data type of any of the parameters (SQLSTATE 428E3)
- Not include a subquery (SQLSTATE 428E3)
- Not include an XMLQUERY or XMLEXISTS expression (SQLSTATE 428E3)

If an argument invokes another function or method, these rules are also enforced for this nested function or method. However, system-generated observer methods are allowed as arguments to the filter function (or any function or method used as an argument), as long as the argument results in a built-in data type.

The definer of the index extension must have EXECUTE privilege on the function, EXECUTEIN privilege on the schema containing this function, or DATAACCESS authority on the schema containing this function..

case-expression

Specifies a case expression for filtering index entries. Either *parameter-name1*, *parameter-name3*, *parameter-name4*, or a constant (SQLSTATE 42703) can be used in the *searched-when-clause* and *simple-when-clause*. An external function with the rules specified in FILTER USING *index-filtering-function-invocation* may be used in *result-expression*. Any function referenced in the *case-expression* must also conform to the rules listed under *index-filtering-function-invocation*. In addition, subqueries and XMLQUERY or XMLEXISTS expressions cannot be used anywhere else in the *case-expression* (SQLSTATE 428E4). The case expression must return an integer (SQLSTATE

428E4). A return value of 1 in the *result-expression* means that the index entry is kept; otherwise, the index entry is discarded.

Notes

- Creating an index extension with a schema name that does not already exist will result in the implicit creation of that schema, provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.

Example

The following example creates an index extension called *grid_extension* that uses a structured type SHAPE column in a table function called *gridEntry* to generate seven index target keys. This index extension also provides two index search methods to produce search ranges when given a search argument.

```
CREATE INDEX EXTENSION GRID_EXTENSION (LEVELS VARCHAR(20) FOR BIT DATA)
  FROM SOURCE KEY (SHAPECOL SHAPE)
  GENERATE KEY USING GRIDENTRY(SHAPECOL..MBR..XMIN,
                               SHAPECOL..MBR..YMIN,
                               SHAPECOL..MBR..XMAX,
                               SHAPECOL..MBR..YMAX,
                               LEVELS)
  WITH TARGET KEY (LEVEL INT, GX INT, GY INT,
                   XMIN INT, YMIN INT, XMAX INT, YMAX INT)
  SEARCH METHODS
  WHEN SEARCHFIRSTBYSECOND (SEARCHARG SHAPE)
  RANGE THROUGH GRIDRANGE(SEARCHARG..MBR..XMIN,
                               SEARCHARG..MBR..YMIN,
                               SEARCHARG..MBR..XMAX,
                               SEARCHARG..MBR..YMAX,
                               LEVELS)
  FILTER USING
    CASE WHEN (SEARCHARG..MBR..YMIN > YMAX) OR
       SEARCHARG..MBR..YMAX < YMIN) THEN 0
    ELSE CHECKDUPLICATE(LEVEL, GX, GY,
                              XMIN, YMIN, XMAX, YMAX,
                              SEARCHARG..MBR..XMIN,
                              SEARCHARG..MBR..YMIN,
                              SEARCHARG..MBR..XMAX,
                              SEARCHARG..MBR..YMAX,
                              LEVELS)
    END
  WHEN SEARCHSECONDBYFIRST (SEARCHARG SHAPE)
  RANGE THROUGH GRIDRANGE(SEARCHARG..MBR..XMIN,
                               SEARCHARG..MBR..YMIN,
                               SEARCHARG..MBR..XMAX,
                               SEARCHARG..MBR..YMAX,
                               LEVELS)
  FILTER USING
    CASE WHEN (SEARCHARG..MBR..YMIN > YMAX) OR
       SEARCHARG..MBR..YMAX < YMIN) THEN 0
    ELSE MBROVERLAP(XMIN, YMIN, XMAX, YMAX,
                               SEARCHARG..MBR..XMIN,
                               SEARCHARG..MBR..YMIN,
                               SEARCHARG..MBR..XMAX,
                               SEARCHARG..MBR..YMAX)
    END
```

CREATE MASK

The CREATE MASK statement creates a column mask at the current server. A column mask specifies the value to be returned for a specified column.

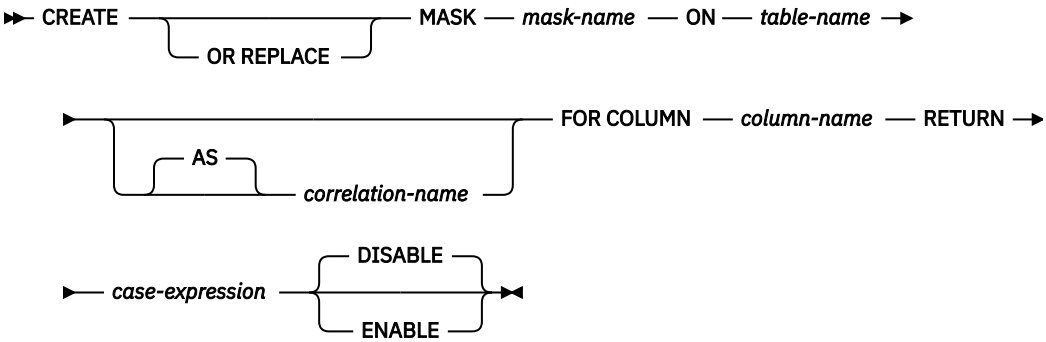
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privileges held by the authorization ID of the statement must include SECADM authority. SECADM authority can create a column mask in any schema. Additional privileges are not needed to reference other objects in the mask definition. For example, the SELECT privilege is not needed to retrieve from a table, and the EXECUTE privilege is not needed to call a user-defined function.

Syntax



Description

OR REPLACE
Specifies to replace the definition for the column mask if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog.

mask-name
Names the column mask. The name, including the implicit or explicit qualifier, must not identify a column mask or a row permission that already exists at the current server (SQLSTATE 42710).

table-name
Identifies the table on which the column mask is created. The name must identify a table that exists at the current server (SQLSTATE 42704). It must not identify a nickname, created or declared temporary table, view, synonym, typed table, alias (SQLSTATE 42809), shadow table or base table of a shadow table (SQLSTATE 428HZ), external table (SQLSTATE 42858), or catalog table (SQLSTATE 42832).

correlation-name
Specifies a correlation name that can be used within *case-expression* to designate the table.

FOR COLUMN *column-name*
Identifies the column to which the mask applies. *column-name* must be an unqualified name that identifies a column of the table (SQLSTATE 42703). A mask must not already exist for the column (SQLSTATE 428HC). The column must not be any of the following columns:

- A LOB column or a distinct type column that is based on a LOB (SQLSTATE 42962).
- An XML column (SQLSTATE 42962).
- A column referenced in an expression that defines a generated column (SQLSTATE 428HB).

RETURN *case-expression*
Specifies a CASE expression to be evaluated to determine the value to return for the column (SQLSTATE 42601). The result of the CASE expression is returned in place of the column value in a row. The result data type, null attribute, and length attribute of the CASE expression must be identical or promotable to those of *column-name* (SQLSTATE 428HB). If the data type of *column-name* is a user-defined data type, the result data type of the CASE expression must be the same user-defined data type. The CASE expression must not reference any of the following objects or elements (SQLSTATE 428HB):

- A created global temporary table or a declared global temporary table.
- A shadow table.

- An external table.
- A nickname.
- A table function.
- A method.
- A parameter marker (SQLSTATE 42601).
- A user-defined function that is defined as not secure.
- A function or expression (such as row change expression, sequence expression) that is non-deterministic or has an external action.
- An XMLQUERY scalar function.
- An XMLEXISTS predicate.
- An OLAP specification.
- A * or `name.*` in a SELECT clause.
- A pseudo-column.
- An aggregate function without specifying the SELECT clause.
- A view that includes any of the previously listed restrictions in its definition.

If the CASE expression references tables for which row or column access control is currently activated, access control from those tables are not cascaded. See the Notes section for details.

ENABLE or DISABLE

Specifies that the column mask is to be enabled or disabled for column access control. The default is DISABLE.

DISABLE

Specifies that the column mask is to be disabled for column access control. If column access control is not currently activated for the table, the column mask will remain ineffective when column access control is activated for the table.

ENABLE

Specifies that the column mask is to be enabled for column access control. If column access control is not currently activated for the table, the column mask will become effective when column access control is activated for the table. If column access control is currently activated for the table, the column mask becomes effective immediately and all packages and dynamic cached statements that reference the table are invalidated.

The application of enabled column masks does not interfere with the operations of other clauses within the statement such as the WHERE, GROUP BY, HAVING, SELECT DISTINCT, and ORDER BY. The rows returned in the final result table remain the same, except that the values in the resulting rows might be masked by the column masks. As such, if the masked column also appears in an ORDER BY *sort-key*, the order is based on the original column values and the masked values in the final result table might not reflect that order. Similarly, the masked values might not reflect the uniqueness enforced by SELECT DISTINCT. If the masked column is embedded in an expression, the result of the expression might become different because the column mask is applied on the column before the expression evaluation can take place. For example, applying a column mask on column SSN might change the result of aggregate function COUNT(DISTINCT SSN) because the DISTINCT operation is performed on the masked values. On the other hand, if the expression in the query is the same as the expression used to mask the column value in the column mask definition, the result of the expression might remain unchanged. For example, the expression in the query is 'XXX-XX-' || SUBSTR(SSN, 8, 4) and the same expression appears in the column mask definition. In this particular example, you can replace the expression in the query with column SSN to avoid the same expression getting evaluated twice.

A column mask is created as a stand alone object without knowing all of the contexts in which it might be used. To mask a column value in the final result table, the column mask definition is merged into a query by the database manager. When the column mask definition is brought into the context of the statement, it might conflict with certain SQL semantics in the statement. Therefore, in some situations, the combination of the statement and the application of the column mask might

return an error (SQLSTATE 428HD). When this happens, either the statement needs to be modified or the column mask must be dropped or recreated with a different definition. See the ALTER TABLE statement description for those situations where a bind time error might be issued for the statement.

If the column is not nullable, its column mask definition will not consider a null value for the column. After column access control is activated for the target table, if the target table is the null-padded table in an outer join operation, the column value in the final result table might be a null. To ensure the column mask has the ability to mask a null value, when the database manager merges the column mask definition into the query, if the target table is the null-padded table in an outer join operation, "WHEN *target-column* IS NULL THEN NULL" will be added as the first WHEN clause to the column mask definition. This forces a null value to be always masked to a null. For a nullable column, this takes away the ability to mask a null value to something else but it is an acceptable restriction from security and usability standpoints.

When a column is used to derive the new value for an INSERT, UPDATE, MERGE, or a SET *transition-variable* assignment statement, the original column value, not the masked value, is used to derive the new value. If the column has a column mask, that column mask is applied to ensure the evaluation of the access control rules at run time masks the column to itself, not to a constant or an expression. This is to ensure the masked values are the same as the original column values. If a column mask does not mask the column to itself, the existing row is not updated or the new row is not inserted and an error is returned at run time (SQLSTATE 428HD). If there is a requirement for masked data to be inserted into a table, the default behavior can be changed by setting the registry variable DB2_ALLOW_WRITE_OF_MASKED_DATA=YES. The rules that are used to apply column masks in order to derive the new values follow the same rules described previously for the final result table of a query.

See the ALTER TABLE statement with the ACTIVATE COLUMN ACCESS CONTROL clause for information about how to activate column access control for the table and how a column mask is applied.

Notes

- **Column masks that are created before column access control is activated for a table:** The CREATE MASK statement is an independent statement that can be used to create a column access control mask before column access control is activated for a table. The only requirement is that the table and the columns exist before the mask is created. Multiple column masks can be created for a table but a column can have one mask only.

  The definition of a mask is stored in the database catalog. Dependency on the table for which the mask is being created and dependencies on other objects referenced in the definition are recorded. No package or dynamic cached statement is invalidated. A column mask can be created as enabled or disabled for column access control. An enabled column mask does not take effect until the ALTER TABLE statement with the ACTIVATE COLUMN ACCESS CONTROL clause is used to activate column access control for the table. SECADM authority is required to issue such an ALTER TABLE statement. A disabled column mask remains ineffective even when column access control is activated for the table. The ALTER MASK statement can be used to alter between ENABLE and DISABLE.

  After column access control is activated for a table, when the table is referenced in a data manipulation statement, all enabled column masks that have been created for the table are implicitly applied by the database manager to mask the values returned for the columns referenced in the final result table of the queries or to determine the new values used in the data change statements.

  Creating column masks before activating column access control for a table is the recommended sequence to avoid multiple invalidations of packages and dynamic cached statements that reference the table.
- **Column masks that are created after column access control is activated for a table:** The enabled column masks become effective as soon as they are committed. All the packages and dynamic cached statements that reference the table are invalidated. Thereafter, when the table is referenced in a data manipulation statement, all enabled column masks are implicitly applied by the database manager to

the statement. Any disabled column masks remain ineffective even when column access control is activated for the table.

- **No cascaded effect when column or row access control enforced tables are referenced in column mask definitions:** A column mask definition can reference tables and columns that are currently enforced by row or column access control. Access control from those tables and columns are ignored when the table for which the column mask is being created is referenced in a data manipulation statement.
- **Consideration for database limits:** If the data manipulation statement already approaches some database limits in the statement, the more enabled column masks and enabled row permissions are created, the more likely they might affect some limits. This is because the enabled column mask and enabled row permission definitions are implicitly merged into the statement when the table is referenced in a data manipulation statement.
- **Column masks that are enabled but in the invalid state:** If a column mask is enabled for column access control but its state is set to invalid, access to the table on which the column mask is defined is blocked until this situation is resolved (SQLSTATE 560D0).
- **Column masks that return data which is not assignable to the column the mask is defined on:** A column mask can be defined so it can return data which is not assignable to the data type of the column the mask is defined on. When this occurs, the CREATE MASK statement is successful but a cast error will be reported when the mask is applied in a user query.

Examples

- *Example 1*: After column access control is activated for table EMPLOYEE, Paul from the payroll department can see the social security number of the employee whose employee number is 123456. Mary who is a manager can see only the last four characters of the social security number. Peter who is neither role cannot see the social security number.

```
CREATE MASK SSN_MASK ON EMPLOYEE
  FOR COLUMN SSN RETURN
    CASE WHEN (VERIFY_GROUP_FOR_USER(SESSION_USER,'PAYROLL') = 1)
                                THEN SSN
         WHEN (VERIFY_GROUP_FOR_USER(SESSION_USER,'MGR') = 1)
                                THEN 'XXX-XX-' || SUBSTR(SSN,8,4)
         ELSE NULL
    END
  ENABLE;
ALTER TABLE EMPLOYEE ACTIVATE COLUMN ACCESS CONTROL;
SELECT SSN FROM EMPLOYEE WHERE EMPNO = 123456;
```

- *Example 2*: In the SELECT statement, column SSN is embedded in an expression that is the same as the expression used in the column mask SSN_MASK. After column access control is activated for table EMPLOYEE, the column mask SSN_MASK is applied to column SSN in the SELECT statement. For this particular expression, the SELECT statement produces the same result as before column access control is activated for all users. The user can replace the expression in the SELECT statement with column SSN to avoid the same expression getting evaluated twice.

```
CREATE MASK SSN_MASK ON EMPLOYEE
  FOR COLUMN SSN RETURN
    CASE WHEN (1 = 1) THEN 'XXX-XX-' || SUBSTR(SSN,8,4)
         ELSE NULL
    END
  ENABLE;
ALTER TABLE EMPLOYEE ACTIVATE COLUMN ACCESS CONTROL;
SELECT 'XXX-XX-' || SUBSTR(SSN,8,4) FROM EMPLOYEE WHERE EMPNO = 123456;
```

- *Example 3*: The California state government conducted a survey for the library usage of the households in each city. Fifty households in each city were sampled in the survey. Each household was given an option, opt-in or opt-out, to show whether their usage in any reports generated from the result of the survey.

  A SELECT statement is used to generate a report to show the average hours used by households in each city. Column mask CITY_MASK is created to mask the city name based on the opt-in or opt-out information chosen by the sampled households. However, after column access control is activated for

table LIBRARY_USAGE, the SELECT statement receives a bind time error. This is because column mask CITY_MASK references another column LIBRARY_OPT and LIBRARY_OPT does not identify a grouping column.

```
CREATE MASK CITY_MASK ON LIBRARY_USAGE
  FOR COLUMN CITY RETURN
    CASE WHEN (LIBRARY_OPT = 'OPT-IN') THEN CITY
         ELSE ' '
    END
  ENABLE;
ALTER TABLE LIBRARY_USAGE ACTIVATE COLUMN ACCESS CONTROL;
SELECT CITY, AVG(LIBRARY_TIME) FROM LIBRARY_USAGE GROUP BY CITY;
```

- *Example 4*: Employee with EMPNO 123456 earns bonus $8000 and salary $80000 in May. When the manager retrieves his salary, the manager receives his salary, not the null value. This is because of no cascaded effect when column mask SALARY_MASK references column BONUS for which column mask BONUS_MASK is defined.

```
CREATE MASK SALARY_MASK ON EMPLOYEE
  FOR COLUMN SALARY RETURN
    CASE WHEN (BONUS < 10000) THEN SALARY
         ELSE NULL
    END
  ENABLE;
CREATE MASK BONUS_MASK ON EMPLOYEE
  FOR COLUMN BONUS RETURN
    CASE WHEN (BONUS > 5000) THEN NULL
         ELSE BONUS
    END
  ENABLE;
ALTER TABLE EMPLOYEE ACTIVATE COLUMN ACCESS CONTROL;
SELECT SALARY FROM EMPLOYEE WHERE EMPNO = 123456;
```

CREATE METHOD

The CREATE METHOD statement is used to associate a method body with a method specification that is already part of the definition of a user-defined structured type.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- CREATEIN privilege on the schema of the structured type referred to in the CREATE METHOD statement
- The owner of the structured type referred to in the CREATE METHOD statement
- SCHEMAADM authority on the schema of the structured type referred to in the CREATE METHOD statement
- DBADM authority

To associate an external method body with its method specification, the privileges held by the authorization ID of the statement must also include at least one of the following authorities:

- CREATE_EXTERNAL_ROUTINE authority on the database
- DBADM authority

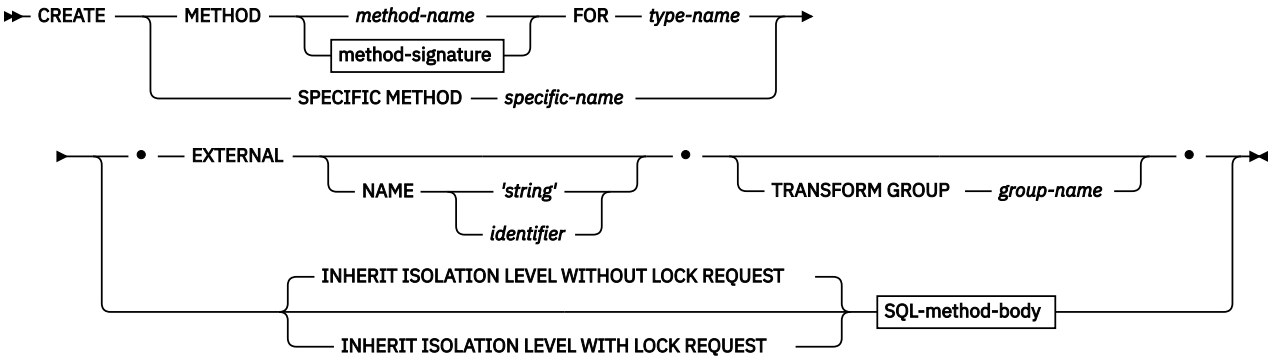
When creating an SQL method, the privileges held by the authorization ID of the statement must also include at least one of the following authorities for each table, view, or nickname identified in any fullselect:

- CONTROL privilege on that table, view, or nickname
- SELECT privilege on that table, view, or nickname
- SELECTIN privilege on the schema containing the table, view, or nickname
- DATAACCESS authority on the schema containing the table, view, or nickname
- DATAACCESS authority on the database

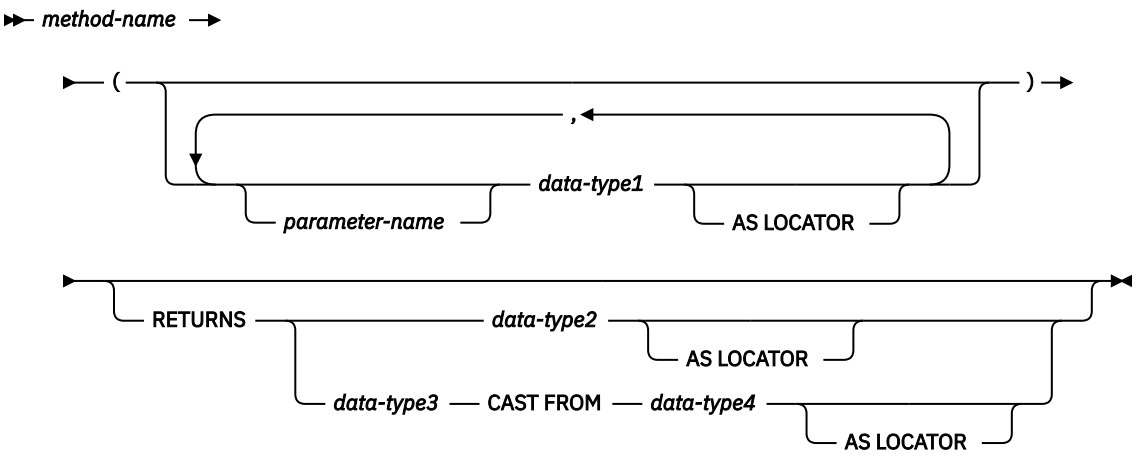
Group privileges other than PUBLIC are not considered for any table or view specified in the CREATE METHOD statement.

Authorization requirements of the data source for the table or view referenced by the nickname are applied when the method is invoked. The authorization ID of the connection can be mapped to a different remote authorization ID.

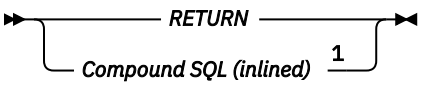
Syntax



method-signature



SQL-method-body



Notes:

[1] The compound SQL (inlined) statement is only supported for an SQL-method-body in an SQL method definition in a non-partitioned database.

Description

METHOD

Identifies an existing method specification that is associated with a user-defined structured type. The method-specification can be identified through one of the following means:

method-name
Names the method specification for which a method body is being defined. The implicit schema is the schema of the subject type (*type-name*). There must be only one method specification for *type-name* that has this *method-name* (SQLSTATE 42725).

method-signature
Provides the method signature which uniquely identifies the method to be defined. The method signature must match the method specification that was provided on the CREATE TYPE or ALTER TYPE statement (SQLSTATE 42883).

method-name
Names the method specification for which a method body is being defined. The implicit schema is the schema of the subject type (*type-name*).

parameter-name
Identifies the parameter name. If parameter names are provided in the method signature, they must be exactly the same as the corresponding parts of the matching method specification. Parameter names are supported in this statement solely for documentation purposes.

data-type1
Specifies the data type of each parameter. Array types are not supported (SQLSTATE 42815).

For a more complete description of each built-in data type, see "CREATE TABLE".

Character and graphic string data types cannot specify string units of CODEUNITS32.

AS LOCATOR
For the LOB types or distinct types which are based on a LOB type, the AS LOCATOR clause can be added.

RETURNS
This clause identifies the output of the method. If a RETURNS clause is provided in the method signature, it must be exactly the same as the corresponding part of the matching method specification on CREATE TYPE. The RETURNS clause is supported in this statement solely for documentation purposes.

data-type2
Specifies the data type of the output. Array types are not supported (SQLSTATE 42815).

AS LOCATOR
For LOB types or distinct types which are based on LOB types, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be returned by the method instead of the actual value.

data-type3* CAST FROM *data-type4
This form of the RETURNS clause is used to return a different data type to the invoking statement from the data type that was returned by the function code.

AS LOCATOR
For LOB types or distinct types which are based on LOB types, the AS LOCATOR clause can be used to indicate that a LOB locator is to be returned from the method instead of the actual value.

FOR *type-name*
Names the type for which the specified method is to be associated. The name must identify a type already described in the catalog (SQLSTATE 42704). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

SPECIFIC METHOD *specific-name*
Identifies the particular method, using the specific name either specified or defaulted to at CREATE TYPE time. The specific-name must identify a method specification in the named or implicit schema; otherwise, an error is raised (SQLSTATE 42704).

EXTERNAL
This clause indicates that the CREATE METHOD statement is being used to register a method, based on code written in an external programming language, and adhering to the documented linkage conventions and interface. The matching method-specification in CREATE TYPE must specify a LANGUAGE other than SQL. When the method is invoked, the subject of the method is passed to the implementation as an implicit first parameter.

If the NAME clause is not specified, "NAME *method-name*" is assumed.

NAME
This clause identifies the name of the user-written code which implements the method being defined.

'*string*'
The *string* option is a string constant with a maximum of 254 bytes. The format used for the string is dependent on the LANGUAGE specified. For more information about the specific language conventions, see "CREATE FUNCTION (External Scalar) statement".

identifier
This identifier specified is an SQL identifier. The SQL identifier is used as the library-id in the string. Unless it is a delimited identifier, the identifier is folded to upper case. If the identifier is qualified with a schema name, the schema name portion is ignored. This form of NAME can only be used with LANGUAGE C (as defined in the method-specification on CREATE TYPE).

TRANSFORM GROUP *group-name*
Indicates the transform group that is used for user-defined structured type transformations when invoking the method. A transform is required since the method definition includes a user-defined structured type.

It is strongly recommended that a transform group name be specified; if this clause is not specified, the default group-name used is DB2_FUNCTION. If the specified (or default) group-name is not defined for a referenced structured type, an error results (SQLSTATE 42741). Likewise, if a required FROM SQL or TO SQL transform function is not defined for the given group-name and structured type, an error results (SQLSTATE 42744).

INHERIT ISOLATION LEVEL WITHOUT LOCK REQUEST or INHERIT ISOLATION LEVEL WITH LOCK REQUEST
Specifies whether or not a lock request can be associated with the isolation-clause of the statement when the method inherits the isolation level of the statement that invokes the method. The default is INHERIT ISOLATION LEVEL WITHOUT LOCK REQUEST.

INHERIT ISOLATION LEVEL WITHOUT LOCK REQUEST
Specifies that, as the method inherits the isolation level of the invoking statement, it cannot be invoked in the context of an SQL statement which includes a lock-request-clause as part of a specified isolation-clause (SQLSTATE 42601).

INHERIT ISOLATION LEVEL WITH LOCK REQUEST
Specifies that, as the method inherits the isolation level of the invoking statement, it also inherits the specified lock-request-clause.

SQL-method-body
The SQL-method-body defines how the method is implemented if the method specification in CREATE TYPE is LANGUAGE SQL.

The SQL-method-body must comply with the following parts of method specification:

- DETERMINISTIC or NOT DETERMINISTIC (SQLSTATE 428C2)
- EXTERNAL ACTION or NO EXTERNAL ACTION (SQLSTATE 428C2)
- CONTAINS SQL or READS SQL DATA (SQLSTATE 42985)

Parameter names can be referenced in the SQL-method-body. The subject of the method is passed to the method implementation as an implicit first parameter named SELF.

For additional details, see "Compound SQL (inlined) statement" and "RETURN statement".

Rules

- The method specification must be previously defined using the CREATE TYPE or ALTER TYPE statement before CREATE METHOD can be used (SQLSTATE 42723).
- If the method being created is an overriding method, those packages that are dependent on the following methods are invalidated:
  - The original method
  - Other overriding methods that have as their subject a supertype of the method being created
- The XML data type cannot be used in a method.

Notes

- If the method allows SQL, the external program must not attempt to access any federated objects (SQLSTATE 55047).
- ***Privileges***: The definer of a method always receives the EXECUTE privilege on the method, as well as the right to drop the method.

  If an EXTERNAL method is created, the definer of the method always receives the EXECUTE privilege WITH GRANT OPTION.

  If an SQL method is created, the definer of the method will only be given the EXECUTE privilege WITH GRANT OPTION on the method when the definer has WITH GRANT OPTION on all privileges required to define the method, or if the definer has SYSADM or DBADM authority. The definer of an SQL method only acquires privileges if the privileges from which they are derived exist at the time the method is created. The definer must have these privileges either directly, or because PUBLIC has the privileges. Privileges held by groups of which the method definer is a member are not considered. When using the method, the connected user's authorization ID must have the valid privileges on the table or view that the nickname references at the data source.
- ***Table access restrictions***: If a method is defined as READS SQL DATA, no statement in the method can access a table that is being modified by the statement which invoked the method (SQLSTATE 57053).

Examples

- *Example 1:*

```
CREATE METHOD BONUS (RATE DOUBLE)
  FOR EMP
  RETURN SELF..SALARY * RATE
```

- *Example 2:*

```
CREATE METHOD SAMEZIP (addr address_t)
  RETURNS INTEGER
  FOR address_t
  RETURN
    (CASE
      WHEN (self..zip = addr..zip)
        THEN 1
      ELSE 0
    END)
```

- *Example 3:*

```
CREATE METHOD DISTANCE (address_t)
  FOR address_t
  EXTERNAL NAME 'addresslib!distance'
  TRANSFORM GROUP func_group
```

CREATE MODULE

The CREATE MODULE statement creates a module at the application server.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

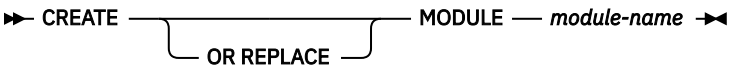
Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the module does not exist.
- CREATEIN privilege on the schema, if the schema name of the module refers to an existing schema.
- SCHEMAADM authority on the schema, if the schema name of the module refers to an existing schema.
- DBADM authority

To replace an existing module, the authorization ID of the statement must be the owner of the existing module (SQLSTATE 42501).

Syntax



Description

OR REPLACE
Specifies replacing the definition for the module if one exists at the current server. The existing module definition is effectively dropped, including all the objects in the module, before the new definition is replaced in the catalog, with the exception that privileges that were granted on the module are not affected. This option is ignored if a definition for the module does not exist at the current server. This option can be specified only by the owner of the object.

module-name
Names the module. The name, including the implicit or explicit qualifier, must not identify an existing module at the current server. The module name and the schema name must not begin with the characters 'SYS' (SQLSTATE 42939) and use of SESSION is not recommended.

Notes

- A module is intended to be a collection of other database objects. Once a module is created, objects in the module are managed using the ALTER MODULE statement. A module can include functions, procedures, types, global variables and conditions. The objects in a module can be published to make them available for reference from outside the module. If an object is not published, it can only be referenced from within the module. A module can be considered to consist of 2 parts:
  - The module specification consists of all the published objects excluding the bodies of any routines.
  - The module body which consists of all objects that are not published and the bodies of any published routines.

  The module management actions include
  - ADD to add an object to the module without publishing it or to replace a routine prototype with the implemented routine definition.

– PUBLISH to add an object to the module and publish it.
– COMMENT on objects in the module.
– DROP to drop an object within the module or drop the module body.

At least one published object should exist in a module in order to have some way to reference the module.

Example

Create a module named *salesModule*

```
CREATE MODULE salesModule
```

CREATE NICKNAME

The CREATE NICKNAME statement defines a nickname for a data source object.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

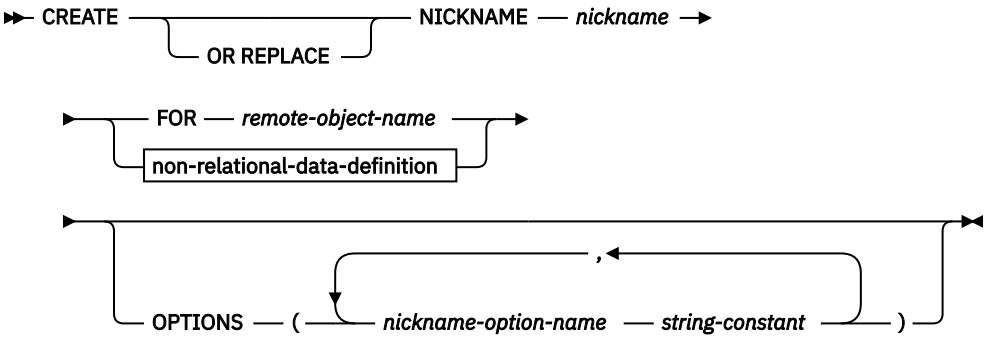
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- CREATETAB authority on the federated database, as well as one of:
  – IMPLICIT_SCHEMA authority on the federated database, if the implicit or explicit schema name of the nickname does not exist
  – CREATEIN privilege on the schema, if the schema name of the nickname refers to an existing schema
  – SCHEMAADM authority on the schema, if the schema name of the nickname refers to an existing schema
- DBADM authority

For data sources that require a user mapping, the privileges held by the authorization ID at the data source must include the privilege to select data from the object that the nickname represents.

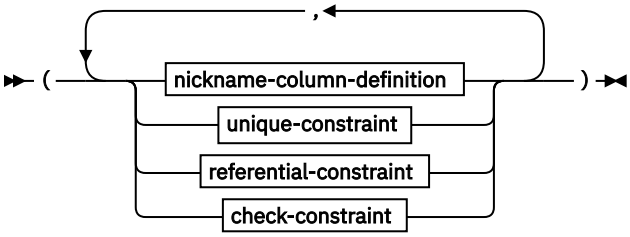
To replace an existing nickname, the authorization ID of the statement must be the owner of the existing nickname (SQLSTATE 42501).

Syntax

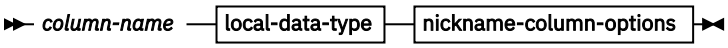


non-relational-data-definition

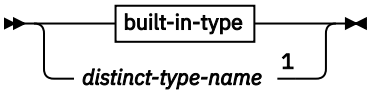
nickname-column-list



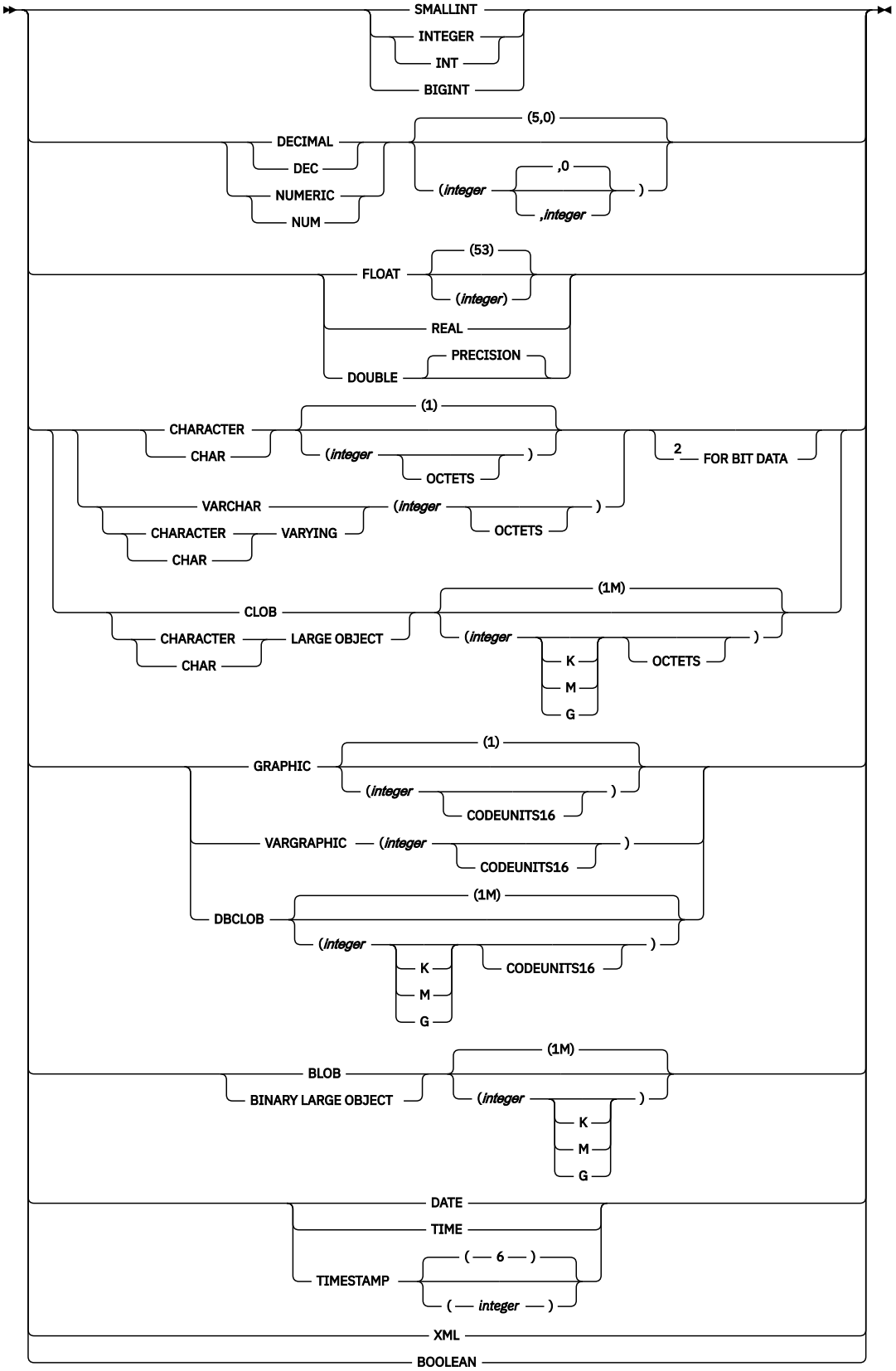
nickname-column-definition



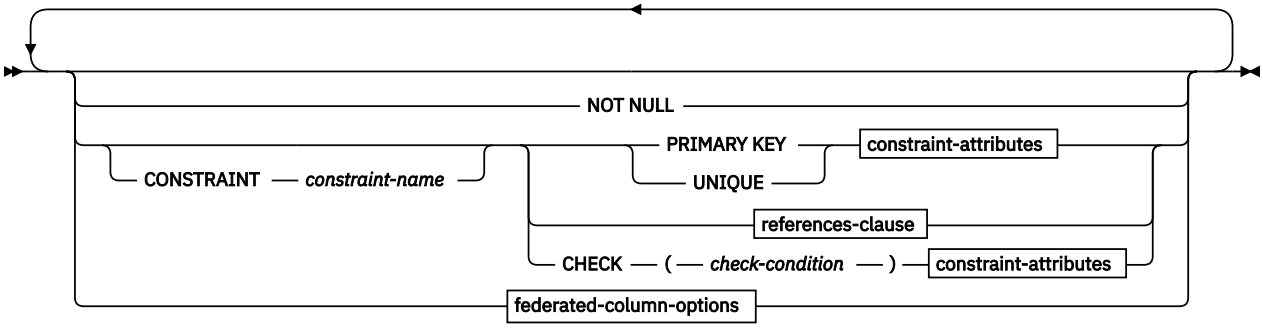
local-data-type



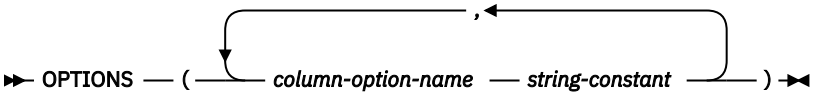
built-in-type



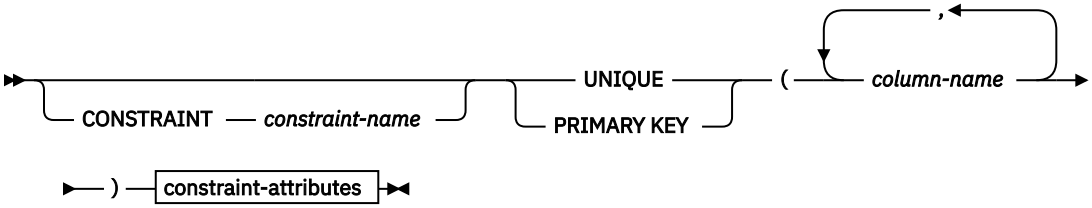
nickname-column-options



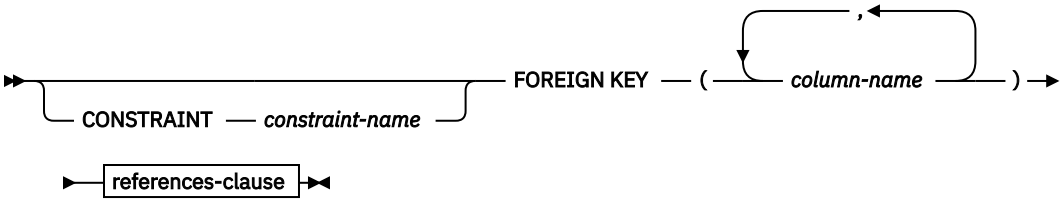
federated-column-options



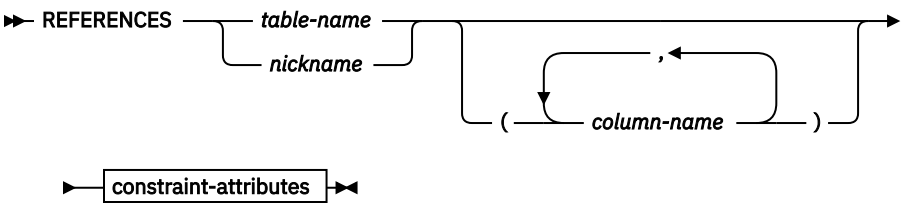
unique-constraint



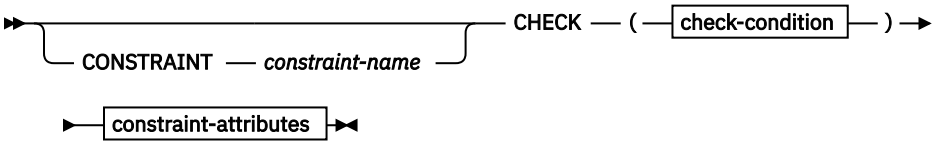
referential-constraint



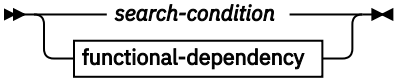
references-clause



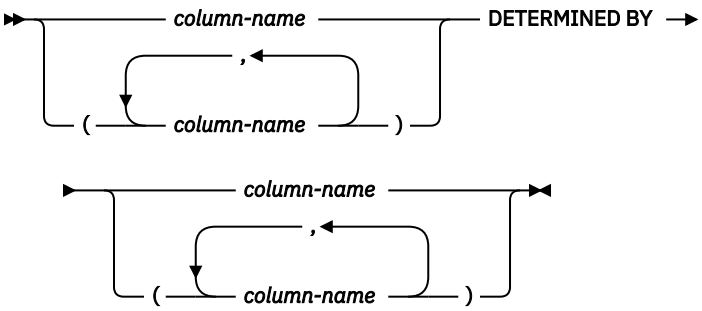
check-constraint



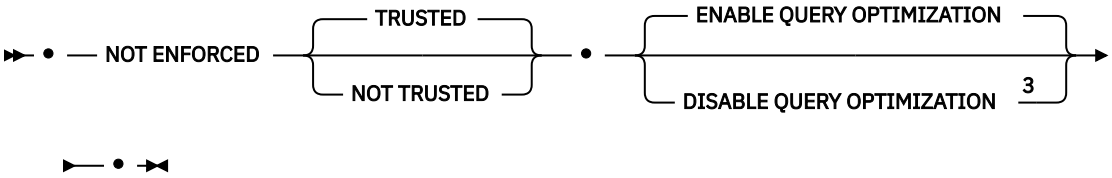
check-condition



functional-dependency



constraint-attributes



Notes:

[1] The specified distinct type cannot have any data type constraints and the source type cannot be an anchored data type (SQLSTATE 428H2).

[2] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow.

[3] DISABLE QUERY OPTIMIZATION is not supported for a unique or primary key constraint.

Description

OR REPLACE
Specifies to replace the definition for the nickname if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog, with the exception that privileges that were granted on the nickname are not affected. This option is ignored if a definition for the nickname does not exist at the current server. This option can be specified only by the owner of the object.

nickname
Specifies a nickname, the identifier used by the federated server for the data source object. The nickname, including the implicit or explicit qualifier, must not identify a table, view, nickname, or alias described in the catalog. The schema name must not begin with 'SYS' (SQLSTATE 42939).

FOR *remote-object-name*
Specifies an identifier. For data sources that support schema names, this is a three-part identifier with the format *data-source-name.remote-schema-name.remote-table-name*. For data sources that do not support schema names, this is a two-part identifier with the format *data-source-name.remote-table-name*.

data-source-name
Names the data source that contains the table or view for which the nickname is being created. The *data-source-name* is the same name that was assigned to the *server-name* in the CREATE SERVER statement.

remote-schema-name
Names the schema to which the table or view belongs. If the remote schema name contains any special or lowercase characters, it must be enclosed by double quotation marks.

remote-table-name
Names the specific data source object (such as a table, alias of a table, or view) for which the nickname is being created. The table cannot be a declared temporary table (SQLSTATE 42995). If the remote table name contains any special or lowercase characters, it must be enclosed by double quotation marks.

For Db2 you can also specify the alias of a table, view, or nickname. For Db2 for z/OS or Db2 for IBM i, you can specify the alias of a table or view.

non-relational-data-definition
Defines the data that is to be accessed through a nonrelational wrapper.

nickname-column-definition
Defines the local attributes of the column for the nickname. Some wrappers require these attributes to be specified, while other wrappers allow the attributes to be determined from the data source.

column-name
Specifies the local name for the column. The name might be different than the corresponding column of the *remote-object-name*.

local-data-type
Specifies the local data type for the column. Some wrappers only support a subset of the SQL data types. For descriptions of specific data types, see "CREATE TABLE" .

built-in-type
See "CREATE TABLE" for the description of built-in data types.

nickname-column-options
Specifies additional options related to columns of the nickname.

NOT NULL
Specifies that the column does not allow null values.

CONSTRAINT *constraint-name*
Names the constraint. A *constraint-name* must not identify a constraint that was already specified within the same CREATE NICKNAME statement (SQLSTATE 42710).

If this clause is omitted, an 18 byte long identifier that is unique among the identifiers of existing constraints defined on the nickname is generated by the system. (The identifier consists of 'SQL' followed by a sequence of 15 numeric characters generated by a timestamp-based function.)

When used with a PRIMARY KEY or UNIQUE constraint, the *constraint-name* can be used as the name of an index specification that is created to support the constraint.

PRIMARY KEY
This provides a shorthand method of defining a primary key composed of a single column. Thus, if PRIMARY KEY is specified in the definition of column C, the effect is the same as if the PRIMARY KEY(C) clause is specified as a separate clause.

See PRIMARY KEY within the description of *unique-constraint*.

UNIQUE
This provides a shorthand method of defining a unique key composed of a single column. Thus, if UNIQUE is specified in the definition of column C, the effect is the same as if the UNIQUE(C) clause is specified as a separate clause.

See UNIQUE within the description of *unique-constraint*.

references-clause
This provides a shorthand method of defining a foreign key composed of a single column. Thus, if a references-clause is specified in the definition of column C, the effect is the same as if that references-clause were specified as part of a FOREIGN KEY clause in which C is the only identified column.

See *references-clause* within the description of *referential-constraint*.

CHECK (*check-condition*)
This provides a shorthand method of defining a check constraint that applies to a single column. See description for CHECK (*check-condition*).

OPTIONS
Indicates the column options that are added when the nickname is created. Some wrappers require that certain column options be specified.

column-option-name
Specifies the name of the option.

string-constant
Specifies the setting for *column-option-name* as a character string constant.

unique-constraint
Defines a unique or primary key constraint.

CONSTRAINT *constraint-name*
Names the primary key or unique constraint.

UNIQUE (*column-name*,...)
Defines a unique key composed of the identified columns. The identified columns must be defined as NOT NULL. Each *column-name* must identify a column of the nickname and the same column must not be identified more than once.

The number of identified columns must not exceed 64, and the sum of their stored lengths must not exceed the index key length limit for the page size. For column stored lengths, see "Byte Counts" in "CREATE TABLE". For key length limits, see "SQL and XQuery limits". No LOB column, distinct type column based on a LOB, or structured type column can be used as part of a unique key, even if the length attribute of the column is small enough to fit within the index key length limit for the page size (SQLSTATE 54008).

The set of columns in the unique key cannot be the same as the set of columns in the primary key or another unique key (SQLSTATE 01543). (If LANGLEVEL is SQL92E or MIA, an error is returned, SQLSTATE 42891.)

The description of the nickname as recorded in the catalog includes the unique key and its index specification. An index specification will automatically be created for the columns in the sequence specified with ascending order for each column. The name of the index specification will be the same as the *constraint-name* if this does not conflict with an existing index or index specification in the schema where the nickname is created. If the name of the index specification conflicts, the name will be 'SQL' followed by a character timestamp (*yymmddhhmmssxxx*), with SYSIBM as the schema name.

PRIMARY KEY (*column-name*,...)
Defines a primary key composed of the identified columns. The clause must not be specified more than once, and the identified columns must be defined as NOT NULL. Each *column-name* must identify a column of the nickname, and the same column must not be identified more than once.

The number of identified columns must not exceed 64, and the sum of their stored lengths must not exceed the index key length limit for the page size. For column stored lengths, see "Byte Counts" in "CREATE TABLE". For key length limits, see "SQL and XQuery limits". No LOB column, distinct type column based on a LOB, or structured type column can be used as part of a primary key, even if the length attribute of the column is small enough to fit within the index key length limit for the page size (SQLSTATE 54008).

The set of columns in the primary key cannot be the same as the set of columns in a unique key (SQLSTATE 01543). (If LANGLEVEL is SQL92E or MIA, an error is returned, SQLSTATE 42891.)

Only one primary key can be defined on a nickname.

The description of the nickname as recorded in the catalog includes the primary key and its index specification. An index specification will automatically be created for the columns in the sequence specified with ascending order for each column. The name of the index specification will be the same as the *constraint-name* if this does not conflict with an existing index or index specification in the schema where the nickname is created. If the name of the index specification conflicts, the name will be 'SQL', followed by a character timestamp (*yymmddhhmmssxxx*), with SYSIBM as the schema name.

referential-constraint
Defines a referential constraint.

CONSTRAINT *constraint-name*
Names the referential constraint.

FOREIGN KEY (*column-name*,...)
Defines a referential constraint with the specified *constraint-name*.

Let N1 denote the object nickname of the statement. The foreign key of the referential constraint is composed of the identified columns. Each name in the list of column names must identify a column of N1, and the same column must not be identified more than once.

The number of identified columns must not exceed 64, and the sum of their stored lengths must not exceed the index key length limit for the page size. For column stored lengths, see "Byte Counts" in "CREATE TABLE". For key length limits, see "SQL and XQuery limits". Foreign keys can be defined on variable length columns whose length is greater than 255 bytes. No LOB column, distinct type column based on a LOB, or structured type column can be used as part of a foreign key (SQLSTATE 42962). There must be the same number of foreign key columns as there are in the parent key, and the data types of the corresponding columns must be compatible (SQLSTATE 42830). Two column descriptions are compatible if they have compatible data types (both columns are numeric, character string, graphic, datetime, or have the same distinct type).

references-clause
Specifies the parent table or the parent nickname, and the parent key for the referential constraint.

REFERENCES *table-name* or *nickname*
The table or nickname specified in a REFERENCES clause must identify a base table or a nickname that is described in the catalog, but must not identify a catalog table.

A referential constraint is a duplicate if its foreign key, parent key, and parent table or parent nickname are the same as the foreign key, parent key, and parent table or parent nickname of a previously specified referential constraint. Duplicate referential constraints are ignored, and a warning is returned (SQLSTATE 01543).

In the following discussion, let N2 denote the identified parent table or parent nickname, and let N1 denote the nickname being created (or altered). N1 and N2 may be the same nickname.

The specified foreign key must have the same number of columns as the parent key of N2, and the description of the *n*th column of the foreign key must be comparable to the description of the *n*th column of that parent key. Datetime columns are not considered to be comparable to string columns for the purposes of this rule.

The referential constraint specified by a FOREIGN KEY clause defines a relationship in which N2 is the parent and N1 is the dependent.

(*column-name*,...)
The parent key of a referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of N2. The same column must not be identified more than once.

The list of column names must match the set of columns (in any order) of the primary key or a unique constraint that exists on N2 (SQLSTATE 42890). If a column name list is not specified, N2 must have a primary key (SQLSTATE 42888). Omission of the column name list is an implicit specification of the columns of that primary key in the sequence originally specified.

constraint-attributes
Defines attributes associated with referential integrity or check constraints.

NOT ENFORCED
The constraint is not enforced by the database manager during normal operations, such as insert, update, or delete.

TRUSTED
The data can be trusted to conform to the constraint. TRUSTED must be used only if the data in the table is independently known to conform to the constraint. Query results might be unpredictable if the data does not actually conform to the constraint. This is the default option.

NOT TRUSTED
The data cannot be trusted to conform to the constraint. NOT TRUSTED is intended for cases where the data conforms to the constraint for most rows, but it is not independently known that all the rows or future additions will conform to the constraint. If a constraint is NOT TRUSTED and enabled for query optimization, then it will not be used to perform optimizations that depend on the data conforming completely to the constraint. NOT TRUSTED can be specified only for referential integrity constraints (SQLSTATE 42613).

ENABLE QUERY OPTIMIZATION
The constraint is assumed to be true and can be used for query optimization under appropriate circumstances.

DISABLE QUERY OPTIMIZATION
The constraint cannot be used for query optimization.

check-constraint
Defines a check constraint. A *check-constraint* is a *search-condition* that must evaluate to not false or that defines a functional dependency between columns.

CONSTRAINT *constraint-name*
Names the check constraint.

CHECK (*check-condition*)
Defines a check constraint. The *check-condition* must be true or unknown for every row of the nickname.

search-condition
The *search-condition* has the following restrictions:

- A column reference must be to a column of the nickname being created.
- The *search-condition* cannot contain a TYPE predicate.
- It cannot contain any of the following elements (SQLSTATE 42621):
  - Subqueries
  - Dereference operations or DEREF functions where the scoped reference argument is other than the object identifier (OID) column
  - CAST specifications with a SCOPE clause
  - Column functions
  - Functions that are not deterministic
  - Functions defined to have an external action
  - User-defined functions defined with either CONTAINS SQL or READS SQL DATA
  - Host variables
  - Parameter markers
  - Special registers and built-in functions that depend on the value of a special register
  - Global variables
  - References to generated columns other than the identity column

functional-dependency
Defines a functional dependency between columns.

The parent set of columns contains the identified columns that immediately precede the DETERMINED BY clause. The child set of columns contains the identified columns that immediately follow the DETERMINED BY clause. All of the restrictions on the *search-condition*

apply to parent set and child set columns, and only simple column references are allowed in the set of columns (SQLSTATE 42621). The same column must not be identified more than once in the functional dependency (SQLSTATE 42709). The data type of the column must not be a LOB data type, a distinct type based on a LOB data type, or a structured type (SQLSTATE 42962). No column in the child set of columns can be a nullable column (SQLSTATE 42621).

If a check constraint is specified as part of a *column-definition*, a column reference can only be made to the same column. Check constraints specified as part of a nickname definition can have column references identifying columns previously defined in the CREATE NICKNAME statement. Check constraints are not checked for inconsistencies, duplicate conditions, or equivalent conditions. Therefore, contradictory or redundant check constraints can be defined, resulting in possible errors at execution time.

FOR SERVER *server-name*

Specifies a server that was registered using the CREATE SERVER statement. This server will be used to access the data for the nickname.

OPTIONS

Specify configuration options for the nickname to be created. Which options you can specify depends on the data source of the object for which a nickname is being created. For a list of data sources and the nickname options that apply to each, see Data source options. Each option value is a character string constant that must be enclosed in single quotation marks.

Notes

- Examples of relational data source objects are: tables and views. Examples of nonrelational data source objects are: Documentum objects or registered tables, text files (.txt), and Microsoft Excel files (.xls).
- The data source object that the nickname references must already exist at the data source denoted by the first qualifier in *remote-object-name*.
- The list of supported data source data types varies from wrapper to wrapper. XML and REF data source data types are not supported by any of the wrappers. DECFLOAT data source data type is supported only by the Db2 wrapper for IBM Db2 Version 9.5 or later. When the CREATE NICKNAME statement specifies a *remote-object-name* that has columns with unsupported data types, an error is returned.

  LONG VARCHAR and LONG VARGRAPHIC data source data types are mapped to CLOB and DBCLOB data types, respectively. LONG VARCHAR FOR BIT DATA is mapped to BLOB.
- The maximum allowable length for index names is 128 bytes. If a nickname is being created for a relational table that has an index whose name exceeds this length, the entire name is not cataloged. Rather, it is truncated to 128 bytes. If the string formed by these characters is not unique within the schema to which the index belongs, an attempt is made to make it unique by replacing the last character with 0. If the result is still not unique, the last character is changed to 1. This process is repeated with numbers 2 through 9 and, if necessary, with numbers 0 through 9 for the name's 127th character, 126th character, and so on, until a unique name is generated. To illustrate: The 130-byte name of an index on a data source table is AREALLY...REALLYLONGNAME. The names AREALLY...REALLYLONGNA and AREALLY...REALLYLONGN0 already exist in the schema to which this index belongs. The new name is over 128 bytes; therefore, it is truncated to AREALLY...REALLYLONGNA. Because this name already exists in the schema, the truncated version is changed to AREALLY...REALLYLONGN0. Because this name also exists, the truncated version is changed to AREALLY...REALLYLONGN1. This name does not already exist in the schema, so it is accepted as a new name.
- When a nickname is created for a data source object, the names of the nickname columns are stored in the catalog. When the data source object is a table or a view, the nickname column names are created to be the same as the table or view column names. If a name exceeds the maximum allowable length for a database column name, the name is truncated to this length. If the truncated version is not unique among the other column names in the table or view, it is made unique by following the procedure described in the preceding paragraph.
- If the data source object has indexes defined, index specifications for each index are created when the nickname is created. Index specifications are not created at the data source for indexes that have:

  - Duplicate column names
  - More than 64 columns
  - More than 1024 bytes in the sum of the length of the index key parts
- If the definition of a remote data source object is changed (for example, a column is deleted or a data type is changed), the nickname should be dropped and recreated; otherwise, errors might occur when the nickname is used in an SQL statement.
- ***Caching and protected objects:*** When a nickname is created, if the data source object is not protected, `ALLOW CACHING` is in effect for the nickname. If the federated server can detect that the data source object is protected, `DISALLOW CACHING` is in effect for the nickname. The `DISALLOW CACHING` option ensures that each time the nickname is used, data for the appropriate authorization ID is returned from the data source at query execution time. This is done by restricting the nickname from being used in the definition of a materialized query table at the federated server, which might be being used to cache the nickname data. The ALTER NICKNAME statement can be used to change between `ALLOW CACHING` and `DISALLOW CACHING`.
- BINARY and VARBINARY types are not supported in a Federated system.
- If the remote data source is Hive, Spark, or Impala, and if the remote data source object contains a column with a large-value character type such as STRING or VARCHAR(65535), the remote column is mapped to local column of type VARCHAR(32672), and any data in excess of 32672 bytes is truncated.
- **Syntax alternatives:** The following syntax is supported for compatibility with previous versions of Db2:
  - ADD can be specified before *nickname-option-name string-constant*.
  - ADD can be specified before *column-option-name string-constant*.

Examples

1. Create a nickname for a view, DEPARTMENT, that is in a schema called HEDGES. This view is stored in a Db2 for z/OS data source called OS390A.

```
CREATE NICKNAME DEPT
  FOR OS390A.HEDGES.DEPARTMENT
```

2. Select all records from the view for which a nickname was created in Example 1. The view must be referenced by its nickname. The remote view can be referenced using the name by which it is known at the data source only in pass-through sessions.

   The following statement is valid after nickname DEPT is created:

```
SELECT * FROM DEPT
```

   The following statement is invalid:

```
SELECT * FROM OS390A.HEDGES.DEPARTMENT
```

3. Create a nickname for the remote table JAPAN that is in a schema called salesdata. Because the schema name and table name on the data source are stored in lowercase, specify the remote schema name and table name with double quotation marks:

```
CREATE NICKNAME JPSALES
  FOR asia."salesdata"."japan"
```

4. Create a nickname for the table-structured file DRUGDATA1.TXT. Include the FILE_PATH, COLUMN DELIMITER, KEY_COLUMN, and VALIDATE_DATA_FILE nickname options in the statement.

```
CREATE NICKNAME DRUGDATA1
  (Dcode         INTEGER,
  DRUG           CHAR(20),
  MANUFACTURER   CHAR(20))
  FOR SERVER biochem_lab
  OPTIONS
    (FILE_PATH '/usr/pat/DRUGDATA1.TXT',
     COLUMN_DELIMITER ',',
```

```
        KEY_COLUMN 'DCODE',
        SORTED 'Y',
        VALIDATE_DATA_FILE 'Y')
```

5. Create the parent nickname CUSTOMERS over multiple XML files under the specified directory path / home/dbuser. Include the following options:
   - Column options:
     - XPATH column option for the VARCHAR(5) column named ID, indicating the element or attribute in the XML file(s) from which the column data is extracted
     - XPATH column option for the VARCHAR(16) column named NAME, indicating the element or attribute in the XML file(s) from which the column data is extracted
     - XPATH column option for the VARCHAR(30) column named ADDRESS, indicating the element or attribute in the XML file(s) from which the column data is extracted
     - PRIMARY_KEY column option for the VARCHAR(16) column named CID, which identifies the customers nickname as a parent nickname in a hierarchy of nicknames
   - Nickname options:
     - DIRECTORY_PATH nickname option to indicate the location of the XML files that provide the data
     - XPATH nickname option to indicate the element in the XML files where the data begins
     - STREAMING nickname option to indicate that the XML source data is separated and processed element by element. In this example, the element is a customer record.

```
CREATE NICKNAME customers
  (id      VARCHAR(5)   OPTIONS(XPATH './@id'),
   name    VARCHAR(16)  OPTIONS(XPATH './/name'),
   address VARCHAR(30)  OPTIONS(XPATH './/address/@street'),
   cid     VARCHAR(16)  OPTIONS(PRIMARY_KEY 'YES'))
   FOR SERVER xml_server
   OPTIONS
     (DIRECTORY_PATH '/home/dbuser',
      XPATH '//customer',
      STREAMING 'YES')
```

6. A Hive table with the name STR_TAB contains a column with the name COL5. COL5 has the type STRING and a column length of 2 GB. When you create a nickname for STR_TAB, the column length of COL5 is reduced to 32672 bytes.

```
CREATE NICKNAME "STRING_NCK" FOR "SERVER10"."STR_TAB"
SQL1812W  Remote column COL5 with length 2147483647 was reduced to 32672. SQLSTATE=0169E
```

CREATE PERMISSION

The CREATE PERMISSION statement creates a row permission at the current server.

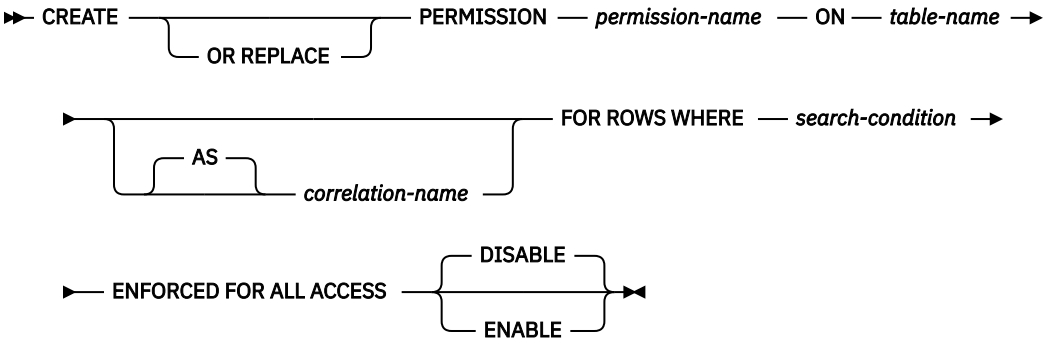
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is implicitly or explicitly specified.

Authorization

The privileges held by the authorization ID of the statement must include SECADM authority. SECADM authority can create a row permission in any schema. Additional privileges are not needed to reference other objects in the permission definition. For example, the SELECT privilege is not needed to retrieve from a table, and the EXECUTE privilege is not needed to call a user-defined function.

Syntax



Description

OR REPLACE
Specifies to replace the definition for the row permission if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog.

permission-name
Names the row permission. The name, including the implicit or explicit qualifier, must not identify a row permission or a column mask that already exists at the current server (SQLSTATE 42710).

table-name
Identifies the table on which the row permission is created. The name must identify a table that exists at the current server (SQLSTATE 42704). It must not identify a nickname, created or declared temporary table, view, synonym, typed table, external table (SQLSTATE 42858), or alias (SQLSTATE 42809). It must not identify a shadow table or a base table of a shadow table (SQLSTATE 428HZ). In releases before Db2 10.5.0.5, *table-name* must not identify a catalog table (SQLSTATE 42832).

correlation-name
Specifies a correlation name that can be used within *search-condition* to designate the table.

FOR ROWS WHERE
Indicates that a row permission is created. A row permission specifies a search condition under which rows of the table can be accessed.

search-condition
Specifies a condition that can be true or false for a row of the table. This follows the same rules used by the search condition in a WHERE clause of a subselect query. In addition, the search condition must not reference any of the following objects or elements (SQLSTATE 428HB):

- A created global temporary table or a declared global temporary table.
- A shadow table.
- An external table.
- A nickname.
- A table function.
- A method.
- A parameter marker (SQLSTATE 42601).
- A user-defined function that is defined as not secure.
- A function or expression (such as row change expression, sequence expression) that is non deterministic or has an external action
- An XMLQUERY scalar function.
- An XMLEXISTS predicate.
- An OLAP specification.
- A `*` or `name.*` in a SELECT clause.

- A pseudocolumn.
- An aggregate function without specifying the SELECT clause.
- A view that includes any of the previously listed restrictions in its definition.

If *search-condition* references tables with currently activated row or column access control, access control from those tables are not cascaded. See "Notes" for details.

ENFORCED FOR ALL ACCESS
Specifies that the row permission applies to all references of the table. If row access control is activated for the table, when the table is referenced in a data manipulation statement, the database manager implicitly applies the row permission to control the access of the table. If the reference of the table is for a fetch operation such as SELECT, the application of the row permission determines what set of rows can be retrieved by the user who requested the fetch operation. If the reference of the table is for a data change operation such as INSERT, the application of the row permission determines whether all rows to be changed can be inserted or updated by the user who requested the data change operation.

ENABLE or DISABLE
Specifies that the row permission is to be enabled or disabled. The default is DISABLE.

DISABLE
Specifies that the row permission is to be disabled. If row access control is not currently activated for the table, the row permission will remain ineffective when row access control is activated for the table.

ENABLE
Specifies that the row permission is to be enabled for row access control. If row access control is not currently activated for the table, the row permission will become effective when row access control is activated for the table. If row access control is currently activated for the table, the row permission becomes effective immediately and all packages and dynamically cached statements that reference the table are invalidated.

See the ACTIVATE ROW ACCESS CONTROL clause in the ALTER TABLE statement for more information about how to activate row access control and how row permissions are applied.

Notes

- **Row permissions that are created before row access control is activated for a table:** The CREATE PERMISSION statement is an independent statement that can be used to create a row permission before row access control is activated for a table. The only requirement is that the table and the columns exist before the permission is created. Multiple row permissions can be created for a table.

  The definition of the row permission is stored in the database catalog. Dependency on the table for which the permission is being created and dependencies on other objects referenced in the definition are recorded. No package or dynamic cached statement is invalidated. A row permission can be created as enabled or disabled for row access control. An enabled row permission does not take effect until the ALTER TABLE statement with the ACTIVATE ROW ACCESS CONTROL clause is used to activate row access control for the table. A disabled row permission remains ineffective even when row access control is activated for the table. The ALTER PERMISSION statement can be used to alter between ENABLE and DISABLE.

  After row access control is activated for a table, when the table is referenced in a data manipulation statement, all enabled row permissions that are defined for the table are implicitly applied by the database manager to control access to the table.

  Creating row permissions before activating row access control for a table is the recommended sequence to avoid multiple invalidations of packages and dynamic cached statements that reference the table.
- **Row permissions that are created after row access control is activated for a table:** An enabled row permission becomes effective as soon as it is committed. All the packages and dynamic cached statements that reference the table are invalidated. Thereafter, when the table is referenced in a data manipulation statement, all enabled row permissions are implicitly applied to the statement. Any disabled row permissions remain ineffective even when row access control is activated for the table.

- **No cascaded effect when row or column access control enforced tables are referenced in row permission definitions:** A row permission definition might reference tables and columns that are currently enforced by row or column access control. Access control from those tables are ignored when the table for which the row permission is being created is referenced in a data manipulation statement.
- **Consideration for database limits:** If the data manipulation statement already approaches some database limits in the statement, the more enabled row permissions and enabled column masks are created, the more likely they might affect some limits. This is because the enabled column mask and enabled row permission definitions are implicitly merged into the statement when the table is referenced in a data manipulation statement. See "SQL and XML Limits" for the limits of a statement.
- **Permissions that are enabled but in the invalid state:** If a permission is enabled for row access control but its state is set to invalid, access to the table on which the permission is defined is blocked until this situation is resolved (SQLSTATE 560D0).

Example

The tellers in a bank can only access customers from their own branch. All tellers are members in role TELLER. The customer service representatives are allowed to access all customers of the bank. All customer service representatives are members in role CSR. A row permission is created accordingly for each group of personnel in the bank by a user with SECADM authority. After row level access control is activated for table CUSTOMER, in the SELECT statement the search conditions of both row permissions are merged into the statement and they are combined with the logical OR operator to control the set of rows accessible by each group.

```
CREATE PERMISSION TELLER_ROW_ACCESS ON CUSTOMER
 FOR ROWS WHERE VERIFY_ROLE_FOR_USER
 (SESSION_USER,'TELLER') = 1 AND
        BRANCH = (SELECT HOME_BRANCH FROM INTERNAL_INFO
             WHERE EMP_ID = SESSION_USER)
ENFORCED FOR ALL ACCESS
ENABLE;

CREATE PERMISSION CSR_ROW_ACCESS ON CUSTOMER
 FOR ROWS WHERE VERIFY_ROLE_FOR_USER(SESSION_USER,'CSR') = 1
 ENFORCED FOR ALL ACCESS
 ENABLE;
```

CREATE PROCEDURE

The CREATE PROCEDURE statement defines a procedure at the current server.

Three different types of procedures can be created using this statement. Each of these types is described separately.

- External. The procedure body is written in a programming language. The external executable is referenced by a procedure defined at the current server, along with various attributes of the procedure.
- Sourced. The procedure body is part of the source procedure, which is referenced by the sourced procedure that is defined at the current server, along with various attributes of the procedure. A sourced procedure whose source procedure is at a data source is also called a *federated procedure*.
- SQL. The procedure body is written in SQL and defined at the current server, along with various attributes of the procedure.

The CREATE PROCEDURE statement can be submitted in obfuscated form. In an obfuscated statement, only the procedure name and its parameters are readable. The rest of the statement is encoded in such a way that is not readable but can be decoded by the database server. Obfuscated statements can be produced by calling the DBMS_DDL.WRAP function.

CREATE PROCEDURE (external)

The CREATE PROCEDURE (external) statement defines an external procedure at the current server.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- CREATE_EXTERNAL_ROUTINE authority on the database and at least one of the following authorities:
  - IMPLICIT_SCHEMA authority on the database, if the schema name of the procedure does not refer to an existing schema
  - CREATEIN privilege on the schema, if the schema name of the procedure refers to an existing schema
  - SCHEMAADM authority on the schema, if the schema name of the procedure refers to an existing schema
- DBADM authority

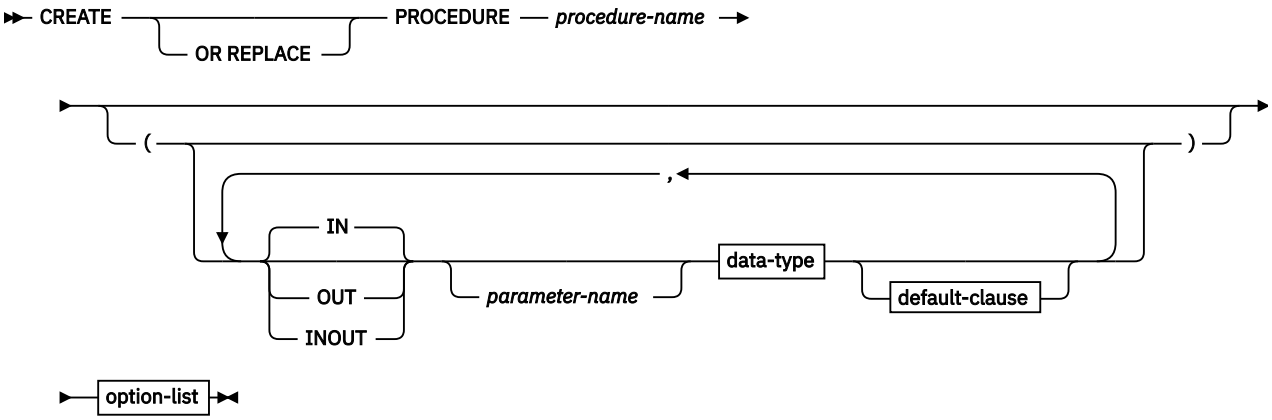
To create a not-fenced procedure, the privileges held by the authorization ID of the statement must also include at least one of the following authorities:

- CREATE_NOT_FENCED_ROUTINE authority on the database
- DBADM authority

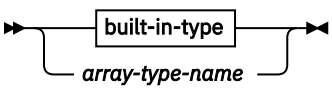
To create a fenced procedure, no additional authorities or privileges are required.

To replace an existing procedure, the authorization ID of the statement must be the owner of the existing procedure (SQLSTATE 42501).

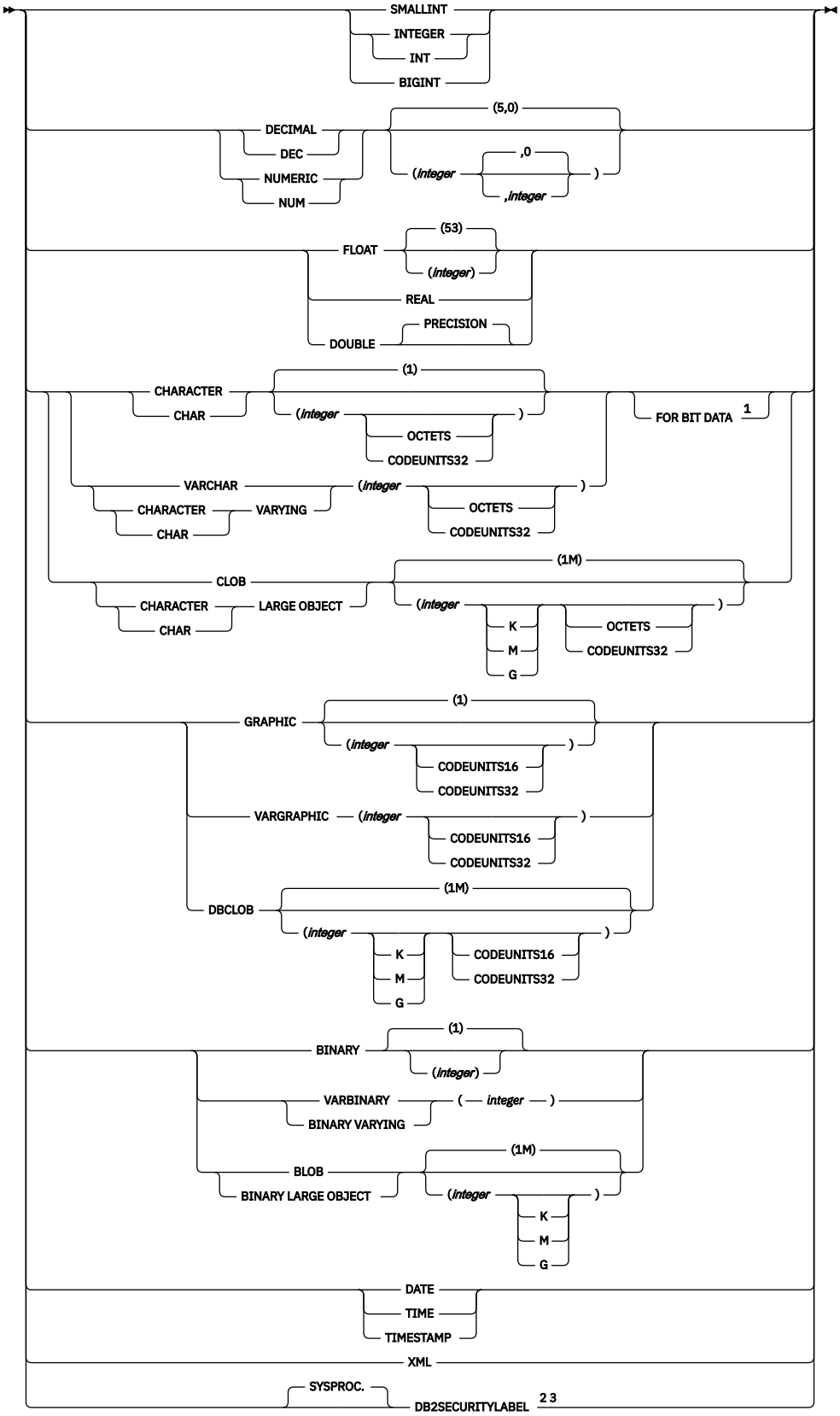
Syntax



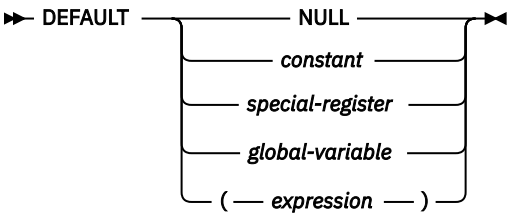
data-type



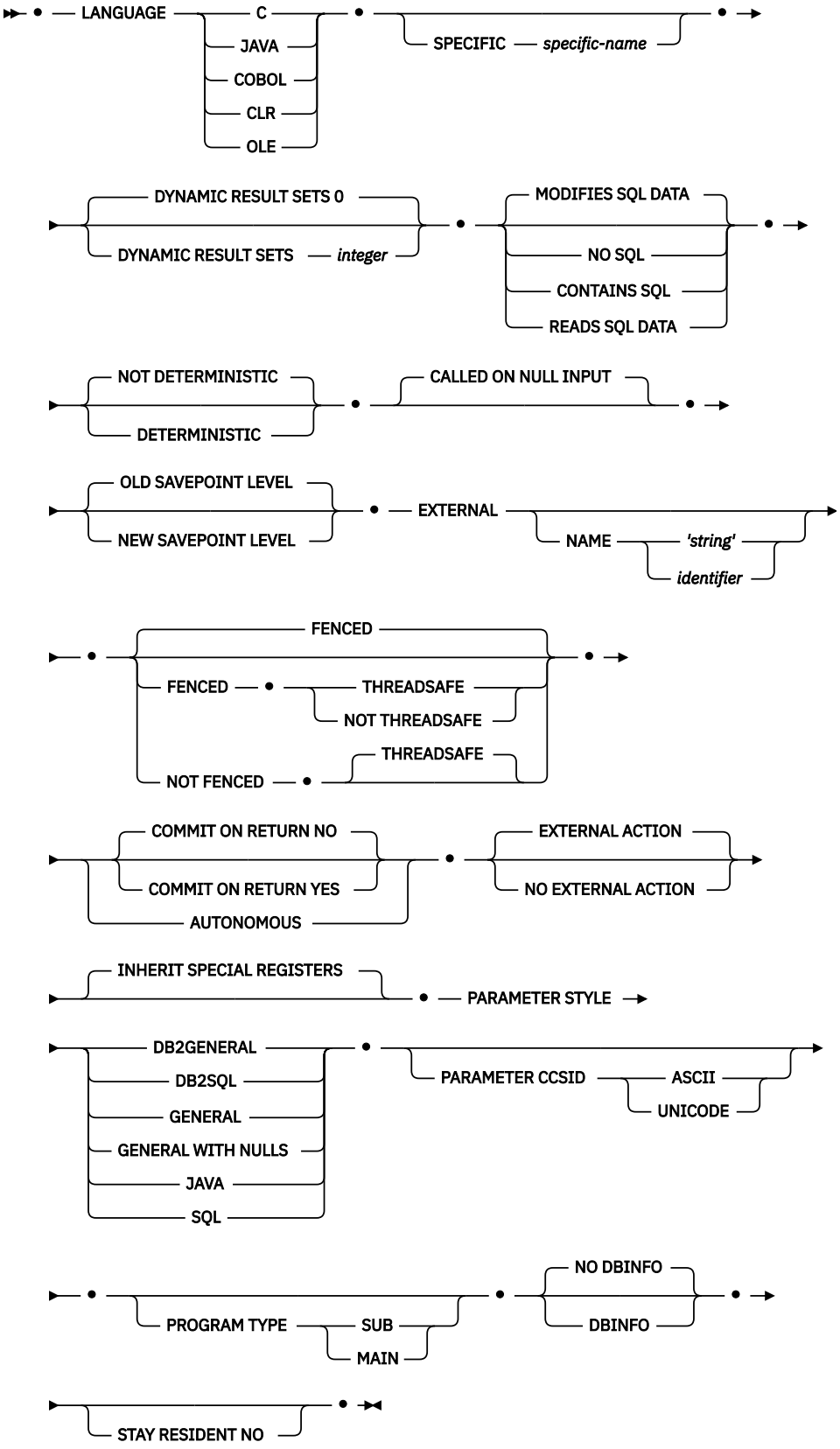
built-in-type



default-clause


DEFAULT
NULL
constant
special-register
global-variable
(
expression
)

option-list



Notes:

[1] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

[2] DB2SECURITYLABEL is the built-in distinct type that must be used to define the row security label column of a protected table.

[3] For a column of type DB2SECURITYLABEL, NOT NULL WITH DEFAULT is implicit and cannot be explicitly specified (SQLSTATE 42842). The default value for a column of type DB2SECURITYLABEL is the session authorization ID's security label for write access.

Description

OR REPLACE

Specifies to replace the definition for the procedure if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog, with the exception that privileges that were granted on the procedure are not affected. This option can be specified only by the owner of the object. This option is ignored if a definition for the procedure does not exist at the current server. To replace an existing procedure, the specific name and procedure name of the new definition must be the same as the specific name and procedure name of the old definition, or the signature of the new definition must match the signature of the old definition. Otherwise, a new procedure is created.

procedure-name

Names the procedure being defined. It is a qualified or unqualified name that designates a procedure. The unqualified form of *procedure-name* is an SQL identifier. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a *schema-name* followed by a period and an SQL identifier.

The name, including the implicit or explicit qualifiers, together with the number of parameters must not identify a procedure described in the catalog (SQLSTATE 42723). The unqualified name, together with the number of the parameters, need not be unique across schemas.

If a two-part name is specified, the *schema-name* cannot begin with 'SYS' (SQLSTATE 42939).

(*IN | OUT | INOUT *parameter-name data-type default-clause,...)

Identifies the parameters of the procedure, and specifies the mode, optional parameter name, data type, and optional default value of each parameter. One entry in the list must be specified for each parameter that the procedure will expect.

No two identically-named procedures within a schema are permitted to have exactly the same number of parameters. A duplicate signature returns an SQL error (SQLSTATE 42723).

For example, given the statements:

```
CREATE PROCEDURE PART (IN NUMBER INT, OUT PART_NAME CHAR(35)) ...
CREATE PROCEDURE PART (IN COST DECIMAL(5,3), OUT COUNT INT) ...
```

the second statement will fail, because the number of parameters in the procedure is the same, even if the data types are not.

If an error is returned by the procedure, OUT parameters are undefined and INOUT parameters are unchanged.

IN

Identifies the parameter as an input parameter to the procedure. Any changes made to the parameter within the procedure are not available to the calling SQL application when control is returned. The default is IN.

OUT

Identifies the parameter as an output parameter for the procedure.

INOUT

Identifies the parameter as both an input and output parameter for the procedure.

parameter-name

Optionally specifies the name of the parameter. The parameter name must be unique for the procedure (SQLSTATE 42734).

data-type
Specifies the data type of the parameter. A structured type cannot be specified (SQLSTATE 429BB).

built-in-type
Specifies a built-in data type. For a more complete description of each built-in data type, see "CREATE TABLE". Only built-in data types that have a correspondence in the language that is being used to write the procedure may be specified.

- A datetime type parameter is passed as a character data type, and the data is passed in the ISO format.
- XML is invalid with LANGUAGE OLE.
- Because the XML value that is seen inside a procedure is a serialized version of the XML value that is passed as a parameter in the procedure call, parameters of type XML must be declared using the syntax `XML AS CLOB(`*n*`)`.
- CLR does not support DECIMAL scale greater than 28 (SQLSTATE 42613).
- Decimal floating-point is not supported with languages C, Java COBOL, CLR, and OLE (SQLSTATE 42613).
- BINARY and VARBINARY data types are invalid with LANGUAGE CLR and OLE (SQLSTATE 42815).

array-type-name
Specifies the name of a user-defined array type. If *array-type-name* is specified without a schema name, the array type is resolved by searching the schemas in the SQL path. The array must be an ordinary array and the procedure must be a Java procedure defined with the PARAMETER STYLE JAVA clause (SQLSTATE 428H2).

DEFAULT
Specifies a default value for the parameter. The default can be a constant, a special register, a global variable, an expression or the keyword NULL. The special registers that can be specified as the default are that same as those that can be specified for a column default (see "*default-clause*" in the "CREATE TABLE" statement). Other special registers can be specified as the default by using an expression.

The *expression* can be any expression of the type described in "Expressions". If a default value is not specified, the parameter has no default and the corresponding argument cannot be omitted on invocation of the procedure. The maximum size of the *expression* is 64K bytes.

The default expression must not modify SQL data (SQLSTATE 428FL or SQLSTATE 429BL). The expression must be assignment compatible to the parameter data type (SQLSTATE 42821).

A default cannot be specified in the following situations:

- For INOUT or OUT parameters (SQLSTATE 42601)
- For a parameter of type ARRAY, ROW, or CURSOR (SQLSTATE 429BB)

SPECIFIC *specific-name*
Provides a unique name for the instance of the procedure that is being defined. This specific name can be used when altering, dropping, or commenting on the procedure. It can never be used to invoke the procedure. The unqualified form of *specific-name* is an SQL identifier. The qualified form is a *schema-name* followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another routine instance that exists at the application server; otherwise an error (SQLSTATE 42710) is raised.

The *specific-name* may be the same as an existing *procedure-name*.

If no qualifier is specified, the qualifier that was used for *procedure-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *procedure-name* or an error (SQLSTATE 42882) is raised.

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is 'SQL' followed by a character timestamp: 'SQL*yymmddhhmmssxxx*'.

DYNAMIC RESULT SETS *integer*
Indicates the estimated upper bound of returned result sets for the procedure.

MODIFIES SQL DATA, NO SQL, CONTAINS SQL, READS SQL DATA
Specifies the classification of SQL statements that can be run by this procedure, or any routine that is called by this procedure. The database manager verifies that the SQL statements issued by the procedure and all routines that are called by the procedure are consistent with this specification.

For the classification of each statement, see "SQL statements that can be executed in routines and triggers" in Developing User-defined Routines (SQL and External).

The default is MODIFIES SQL DATA.

MODIFIES SQL DATA
Specifies that the procedure can run any SQL statement except statements that are not supported in procedures (SQLSTATE 38003).

NO SQL
Specifies that the procedure can run only SQL statements with a data access classification of NO SQL (SQLSTATE 38001).

CONTAINS SQL
Specifies that the procedure can run only statements with a data access classification of CONTAINS SQL or NO SQL (SQLSTATE 38003 or 38004).

READS SQL DATA
Specifies that the procedure can run statements with a data access classification of READS SQL DATA, CONTAINS SQL, or NO SQL (SQLSTATE 38002 or 38003 or 42985).

DETERMINISTIC or NOT DETERMINISTIC
This clause specifies whether the procedure always returns the same results for given argument values (DETERMINISTIC) or whether the procedure depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC procedure must always return the same result from successive invocations with identical inputs.

This clause currently does not impact processing of the procedure.

CALLED ON NULL INPUT
CALLED ON NULL INPUT always applies to procedures. This means that the procedure is called regardless of whether any arguments are null. Any OUT or INOUT parameter can return a null value or a normal (non-null) value. Responsibility for testing for null argument values lies with the procedure.

OLD SAVEPOINT LEVEL or NEW SAVEPOINT LEVEL
Specifies whether or not this procedure establishes a new savepoint level for savepoint names and effects. OLD SAVEPOINT LEVEL is the default behavior. For more information about savepoint levels, see the "Rules" section in the description of the SAVEPOINT statement.

LANGUAGE
This mandatory clause is used to specify the language interface convention to which the procedure body is written.

C
This means the database manager will call the procedure as if it were a C procedure. The procedure must conform to the C language calling and linkage convention as defined by the standard ANSI C prototype.

JAVA
This means the database manager will call the procedure as a method in a Java class.

COBOL
This means the database manager will call the procedure as if it were a COBOL procedure.

For Micro Focus COBOL, NOT THREADSAFE should be specified for procedures defined

CLR
This means the database manager will call the procedure as a method in a .NET class. At this time, LANGUAGE CLR is only supported for procedures running on Windows operating systems. NOT FENCED cannot be specified for a CLR routine (SQLSTATE 42601).

OLE
This means the database manager will call the procedure as if it were a method exposed by an OLE automation object. The stored-procedure must conform with the OLE automation data types and invocation mechanism. Also, the OLE automation object needs to be implemented as an in-process server (DLL). These restrictions are outlined in the *OLE Automation Programmer's Reference*.

LANGUAGE OLE is only supported for procedures stored in databases running on Windows operating systems. THREADSAFE may not be specified for procedures defined with LANGUAGE OLE (SQLSTATE 42613).

EXTERNAL
This clause indicates that the CREATE PROCEDURE statement is being used to register a new procedure based on code written in an external programming language and adhering to the documented linkage conventions and interface.

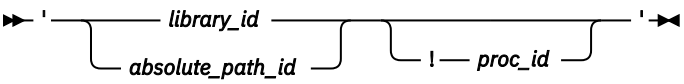
If the NAME clause is not specified, "NAME *procedure-name*" is assumed. If the NAME clause is not formatted correctly, an error is returned (SQLSTATE 42878).

NAME *'string'*
This clause identifies the name of the user-written code which implements the procedure being defined.

The 'string' option is a string constant with a maximum of 254 bytes. The format used for the string is dependent on the LANGUAGE specified.

- For LANGUAGE C:

  The *string* specified is the library name and procedure within the library, which the database manager invokes to execute the procedure being CREATEd. The library (and the procedure within the library) do not need to exist when the CREATE PROCEDURE statement is performed. However, when the procedure is called, the library and procedure within the library must exist and be accessible from the database server machine.



The name must be enclosed by single quotation marks. Extraneous blanks are not permitted.

library_id
Identifies the library name containing the procedure. The database manager will look for the library as follows:

Operating system	Library name location
Linux AIX	If 'myfunc' was given as the *library_id*, and the database manager is being run from `/u/production`, the database manager will look for the procedure in library `/u/production/sqllib/function/myproc` if FENCED is specified, or `/u/production/sqllib/function/unfenced/myproc` if NOT FENCED is specified.
Windows	The database manager will look for the function in a directory path that is specified by the LIBPATH or PATH environment variable.

Stored procedures located in any of these directories do not use any of the registered attributes.

absolute_path_id
Identifies the full path name of the procedure. The format depends on the operating system, as illustrated in the following table:

Operating system	Full path name example
Linux AIX	A value of '/u/jchui/mylib/myproc' would cause the database manager to look in `/u/jchui/mylib` for the `myproc` procedure.
Windows	A value of 'd:\mylib\myproc.dll' would cause the database manager to load the file `myproc.dll` from the `d:\mylib` directory. If an absolute path ID is being used to identify the routine body, be sure to append the `.dll` extension.

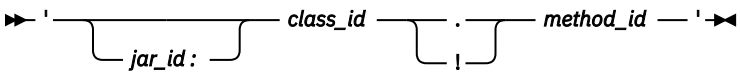
! proc_id
Identifies the entry point name of the procedure to be invoked. The exclamation point (!) serves as a delimiter between the library ID and the procedure ID. '!proc8' would direct the database manager to look for the library in the location specified by *absolute_path_id*, and to use entry point proc8 within that library.

If the string is not properly formed, an error is returned (SQLSTATE 42878).

The body of every procedure should be in a directory that is mounted and available on every database partition.

- For LANGUAGE JAVA:

  The *string* specified contains the optional jar file identifier, class identifier and method identifier, which the database manager invokes to execute the procedure being created. The class identifier and method identifier do not need to exist when the CREATE PROCEDURE statement is performed. If a *jar_id* is specified, it must exist when the CREATE PROCEDURE statement is performed. However, when the procedure is called, the class identifier and the method identifier must exist and be accessible from the database server machine, otherwise an error is returned (SQLSTATE 42884).



The name must be enclosed by single quotation marks. Extraneous blanks are not permitted.

jar_id
Identifies the jar identifier given to the jar collection when it was installed in the database. It can be either a simple identifier or a schema qualified identifier. Examples are 'myJar' and 'mySchema.myJar'.

class_id
Identifies the class identifier of the Java object. If the class is part of a package, the class identifier part must include the complete package prefix, for example, 'myPacks.StoredProcs'. The directory the Java virtual machine will look in for the classes depends on the operating system, as illustrated in the following table:

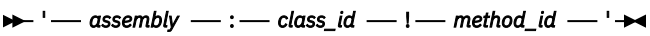
Operating system	Directory the Java virtual machine will look in for the classes
Linux AIX	`'.../myPacks/UserProcs/'`
Windows	`'...\myPacks\UserProcs\'`

method_id
Identifies the method name with the Java class to be invoked.

- For LANGUAGE CLR:

The *string* specified represents the .NET assembly (library or executable), the class within that assembly, and the method within the class that the database manager invokes to execute the procedure being created. The module, class, and method do not need to exist when the CREATE PROCEDURE statement is executed. However, when the procedure is called, the module, class, and method must exist and be accessible from the database server machine, otherwise an error is returned (SQLSTATE 42284).

C++ routines that are compiled with the '/clr' compiler option to indicate that they include managed code extensions must be cataloged as 'LANGUAGE CLR' and not 'LANGUAGE C'. The database manager needs to know that the .NET infrastructure is being utilized in a procedure in order to make necessary runtime decisions. All procedures using the .NET infrastructure must be cataloged as 'LANGUAGE CLR'.



The name must be enclosed by single quotation marks. Extraneous blanks are not permitted.

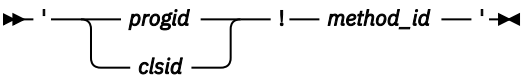
assembly
Identifies the DLL or other assembly file in which the class is located. Any file extensions (such as .dll) must be specified. If the full path name is not given, the file must be in the function directory of the database instance path (for example, `C:\Program Data\IBM\Db2\Copy Name`). If the file is in a subdirectory of the instance function directory, the subdirectory can be given before the file name rather than specifying the full path. For example, if your instance directory is `C:\Program Data\IBM\Db2\Copy Name` and your assembly file is `C:\Program Data\IBM\Db2\Copy Name\function\myprocs\mydotnet.dll`, it is only necessary to specify `'myprocs\mydotnet.dll'` for the assembly. The case sensitivity of this parameter is the same as the case sensitivity of the file system.

class_id
Specifies the name of the class within the given assembly in which the method that is to be invoked resides. If the class resides within a namespace, the full namespace must be given in addition to the class. For example, if the class EmployeeClass is in namespace MyCompany.ProcedureClasses, then MyCompany.ProcedureClasses.EmployeeClass must be specified for the class. Note that the compilers for some .NET languages will add the project name as a namespace for the class, and the behavior may differ depending on whether the command line compiler or the GUI compiler is used. This parameter is case sensitive.

method_id
Specifies the method within the given class that is to be invoked. This parameter is case sensitive.

- For LANGUAGE OLE:

The string specified is the OLE programmatic identifier (*progid*) or class identifier (*clsid*), and method identifier (*method_id*), which the database manager invokes to execute the procedure being created by the statement. The programmatic identifier or class identifier, and the method identifier do not need to exist when the CREATE PROCEDURE statement is executed. However, when the procedure is used in the CALL statement, the method identifier must exist and be accessible from the database server machine, otherwise an error results (SQLSTATE 42724).



The name must be enclosed by single quotation marks. Extraneous blanks are not permitted.

progid
Identifies the programmatic identifier of the OLE object.

A *progid* is not interpreted by the database manager, but only forwarded to the OLE automation controller at run time. The specified OLE object must be creatable and support late binding (also known as IDispatch-based binding). By convention, progids have the following format:

```
<program_name>.<component_name>.<version>
```

Because this is only a convention, and not a rule, *progids* may in fact have a different format.

clsid
Identifies the class identifier of the OLE object to create. It can be used as an alternative for specifying a progid in the case that an OLE object is not registered with a progid. The *clsid* has the form:

```
{nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn}
```

where 'n' is an alphanumeric character. A *clsid* is not interpreted by the database manager, but only forwarded to the OLE APIs at run time.

method_id
Identifies the method name of the OLE object to be invoked.

NAME *identifier*
This *identifier* specified is an SQL identifier. The SQL identifier is used as the *library-id* in the string. Unless it is a delimited identifier, the identifier is folded to upper case. If the identifier is qualified with a schema name, the schema name portion is ignored. This form of NAME can only be used with LANGUAGE C.

FENCED or NOT FENCED
This clause specifies whether the procedure is considered "safe" to run in the database manager operating environment's process or address space (NOT FENCED), or not (FENCED).

If a procedure is registered as FENCED, the database manager protects its internal resources (for example, data buffers) from access by the procedure. All procedures have the option of running as FENCED or NOT FENCED. In general, a procedure running as FENCED will not perform as well as a similar one running as NOT FENCED.

CAUTION: Use of NOT FENCED for procedures that were not adequately coded, reviewed, and tested can compromise the integrity of a Db2 database. Db2 databases take some precautions against many of the common types of inadvertent failures that might occur, but cannot guarantee complete integrity when NOT FENCED stored procedures are used.

Either SYSADM authority, DBADM authority, or a special authority (CREATE_NOT_FENCED) is required to register a procedure as NOT FENCED. Only FENCED can be specified for a procedure with LANGUAGE OLE or NOT THREADSAFE.

LANGUAGE CLR procedures cannot be created when specifying the NOT FENCED clause (SQLSTATE 42601).

THREADSAFE or NOT THREADSAFE
Specifies whether the procedure is considered safe to run in the same process as other routines (THREADSAFE), or not (NOT THREADSAFE).

If the procedure is defined with LANGUAGE other than OLE:

- If the procedure is defined as THREADSAFE, the database manager can invoke the procedure in the same process as other routines. In general, to be threadsafe, a procedure should not use any global or static data areas. Most programming references include a discussion of writing threadsafe routines. Both FENCED and NOT FENCED procedures can be THREADSAFE.
- If the procedure is defined as NOT THREADSAFE, the database manager will never invoke the procedure in the same process as another routine.

For FENCED procedures, THREADSAFE is the default if the LANGUAGE is JAVA or CLR. For all other languages, NOT THREADSAFE is the default. If the procedure is defined with LANGUAGE OLE, THREADSAFE may not be specified (SQLSTATE 42613).

For NOT FENCED procedures, THREADSAFE is the default. NOT THREADSAFE cannot be specified (SQLSTATE 42613).

COMMIT ON RETURN
Indicates whether a commit is to be issued on return from the procedure. The default is NO.

NO
A commit is not issued when the procedure returns.

YES
A commit is issued when the procedure returns if a positive SQLCODE is returned by the CALL statement

The commit operation includes the work that is performed by the calling application process and the procedure.

If the procedure returns result sets, the cursors that are associated with the result sets must have been defined as WITH HOLD to be usable after the commit.

AUTONOMOUS
Indicates the procedure should execute in its own autonomous transaction scope.

EXTERNAL ACTION or NO EXTERNAL ACTION
Specifies whether the procedure takes some action that changes the state of an object not managed by the database manager (EXTERNAL ACTION), or not (NO EXTERNAL ACTION). The default is EXTERNAL ACTION. If NO EXTERNAL ACTION is specified, the system can use certain optimizations that assume the procedure has no external impact.

INHERIT SPECIAL REGISTERS
This optional clause specifies that updatable special registers in the procedure will inherit their initial values from the environment of the invoking statement.

No changes to the special registers are passed back to the caller of the procedure.

Non-updatable special registers, such as the datetime special registers, reflect a property of the statement currently executing, and are therefore set to their default values.

PARAMETER STYLE
This clause is used to specify the conventions used for passing parameters to and returning the value from procedures.

DB2GENERAL
This means that the procedure will use a parameter passing convention that is defined for use with Java methods. This can only be specified when LANGUAGE JAVA is used.

DB2SQL
In addition to the parameters on the CALL statement, the following arguments are passed to the procedure:

- A vector containing a null indicator for each parameter on the CALL statement
- The SQLSTATE to be returned to the database manager
- The qualified name of the procedure
- The specific name of the procedure
- The SQL diagnostic string to be returned to the database manager

This can only be specified when LANGUAGE C, COBOL, CLR, or OLE is used.

GENERAL
This means that the procedure will use a parameter passing mechanism by which the procedure receives the parameters specified on the CALL. The parameters are passed directly, as expected by the language; the SQLDA structure is not used. This can only be specified when LANGUAGE C, COBOL, or CLR is used.

Null indicators are *not* directly passed to the program.

GENERAL WITH NULLS
In addition to the parameters on the CALL statement specified under GENERAL, another argument is passed to the procedure. This additional argument is a vector of null indicators, one for each of the parameters on the CALL statement. In C, this would be an array of short integers. This can only be specified when LANGUAGE C, COBOL, or CLR is used.

JAVA
This means that the procedure will use a parameter passing convention that conforms to the Java language and SQLJ Routines specification. IN/OUT and OUT parameters will be passed as single entry arrays to facilitate returning values. This can only be specified when LANGUAGE JAVA is used.

PARAMETER STYLE JAVA procedures do not support the DBINFO or PROGRAM TYPE clauses.

SQL
In addition to the parameters on the CALL statement, the following arguments are passed to the procedure:

- A null indicator for each parameter on the CALL statement
- The SQLSTATE to be returned to the database manager
- The qualified name of the procedure
- The specific name of the procedure
- The SQL diagnostic string to be returned to the database manager

This can only be specified when LANGUAGE C, COBOL, CLR, or OLE is used.

PARAMETER CCSID
Specifies the encoding scheme to use for all string data passed into and out of the procedure. If the PARAMETER CCSID clause is not specified, the default is PARAMETER CCSID UNICODE for Unicode databases, and PARAMETER CCSID ASCII for all other databases.

ASCII
Specifies that string data is encoded in the database code page. If the database is a Unicode database, PARAMETER CCSID ASCII cannot be specified (SQLSTATE 56031). When the procedure is invoked, the application code page for the procedure is the database code page.

UNICODE
Specifies that string data is encoded in Unicode. If the database is a Unicode database, character data is in UTF-8, and graphic data is in UCS-2. If the database is not a Unicode database, character data is in UTF-8. In either case, when the procedure is invoked, the application code page for the procedure is 1208.

If the database is not a Unicode database, and a procedure with PARAMETER CCSID UNICODE is created, the procedure cannot have any graphic types, the XML type, or user-defined types (SQLSTATE 560C1).

If the database is not a Unicode database, and the alternate collating sequence has been specified in the database configuration, procedures can be created with either PARAMETER CCSID ASCII or PARAMETER CCSID UNICODE. All data passed into and out of the procedure will be converted to the appropriate code page.

This clause cannot be specified with LANGUAGE OLE, LANGUAGE JAVA, or LANGUAGE CLR (SQLSTATE 42613).

PROGRAM TYPE
Specifies whether the procedure expects parameters in the style of a main routine or a subroutine. The default is SUB.

SUB
The procedure expects the parameters to be passed as separate arguments.

MAIN
The procedure expects the parameters to be passed as an argument counter, and a vector of arguments (argc, argv). The name of the procedure to be invoked must also be "main". Stored procedures of this type must still be built in the same fashion as a shared library, rather than a stand-alone executable. PROGRAM TYPE MAIN is only valid when the LANGUAGE clause specifies one of: C, COBOL, or CLR.

DBINFO or NO DBINFO
Specifies whether specific information known by the database manager is passed to the procedure when it is invoked as an additional invocation-time argument (DBINFO) or not (NO DBINFO). NO DBINFO is the default. DBINFO is not supported for LANGUAGE OLE (SQLSTATE 42613). It is also not supported for PARAMETER STYLE JAVA or DB2GENERAL.

If DBINFO is specified, a structure containing the following information is passed to the procedure:

- Data base name - the name of the currently connected database.
- Application ID - unique application ID which is established for each connection to the database.
- Application Authorization ID - the authorization ID of the user that connected to the database (the SYSTEM_USER special register).
- Code page - identifies the database code page.
- Database version/release - identifies the version, release and modification level of the database server invoking the procedure.
- Platform - contains the server's platform type.

The DBINFO structure is common for all external routines and contains additional fields that are not relevant to procedures.

If you change session authorization ID (the SESSION_USER special register) using the SET SESSION AUTHORIZATION statement, the Application Authorization ID still returns the value of the SYSTEM_USER special register.

STAY RESIDENT NO
Specifies that the library that is loaded for the function is not to remain resident in memory after the function ends. This clause is ignored when:

- The NOT FENCED clause is specified.
- The LANGUAGE option is set to JAVA or CLR.

Rules

- ***Autonomous routine restrictions:*** Autonomous routines cannot return result sets and do not support the following parameter data types (SQLSTATE 428H2):
  - Cursor types
  - Structured types
  - XML

  Global variables of cursor types cannot be referenced within the autonomous scope.

Notes

- Creating a procedure with a schema name that does not already exist results in the implicit creation of that schema, provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- A Java routine defined as NOT FENCED will be invoked as if it had been defined as FENCED THREADSAFE.
- A procedure that is called from within a compound SQL (inlined) statement will execute as if it were created specifying NEW SAVEPOINT LEVEL, even if OLD SAVEPOINT LEVEL was specified or defaulted to when the procedure was created.

- XML parameters are only supported in LANGUAGE JAVA external procedures when the PARAMETER STYLE DB2GENERAL clause is specified.
- ***Setting of the default value:*** Parameters of a procedure that are defined with a default value are set to their default value when the procedure is invoked, but only if a value is not supplied for the corresponding argument, or is specified as DEFAULT, when the procedure is invoked.
- ***Privileges:*** The definer of a procedure always receives the EXECUTE privilege WITH GRANT OPTION on the procedure, as well as the right to drop the procedure. When the procedure is used in an SQL statement, the procedure definer must have the EXECUTE privilege on any packages used by the procedure or EXECUTEIN privilege on the schema containing the packages used by the procedure.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - RESULT SETS can be specified in place of DYNAMIC RESULT SETS.
  - NULL CALL can be specified in place of CALLED ON NULL INPUT.
  - DB2GENRL can be specified in place of DB2GENERAL.
  - SIMPLE CALL can be specified in place of GENERAL.
  - SIMPLE CALL WITH NULLS can be specified in place of GENERAL WITH NULLS.
  - PARAMETER STYLE DB2DARI is supported.

  The following syntax is accepted as the default behavior:

  - ASUTIME NO LIMIT
  - NO COLLID
  - STAY RESIDENT NO
  - CCSID UNICODE in a Unicode database
  - CCSID ASCII in a non-Unicode database if PARAMETER CCSID UNICODE is not specified

Examples

- *Example 1:* Create the procedure definition for a procedure, written in Java, that is passed a part number and that returns the cost of the part and the quantity that is currently available.

```
CREATE PROCEDURE PARTS_ON_HAND (IN PARTNUM INTEGER,
    OUT COST DECIMAL(7,2),
    OUT QUANTITY INTEGER)
EXTERNAL NAME 'parts.onhand'
LANGUAGE JAVA PARAMETER STYLE JAVA
```

- *Example 2:* Create the procedure definition for a procedure, written in C, that is passed an assembly number and returns the number of parts that make up the assembly, total part cost, and a result set that lists the part numbers, quantity, and unit cost of each part.

```
CREATE PROCEDURE ASSEMBLY_PARTS (IN ASSEMBLY_NUM INTEGER,
    OUT NUM_PARTS INTEGER,
    OUT COST DOUBLE)
  EXTERNAL NAME 'parts!assembly'
  DYNAMIC RESULT SETS 1 NOT FENCED
  LANGUAGE C PARAMETER STYLE GENERAL
```

CREATE PROCEDURE (sourced)

The CREATE PROCEDURE (sourced) statement defines a procedure (the *sourced procedure*) that is based on another procedure (the *source procedure*). In a federated system, a *federated procedure* is a sourced procedure whose source procedure is at a supported data source.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

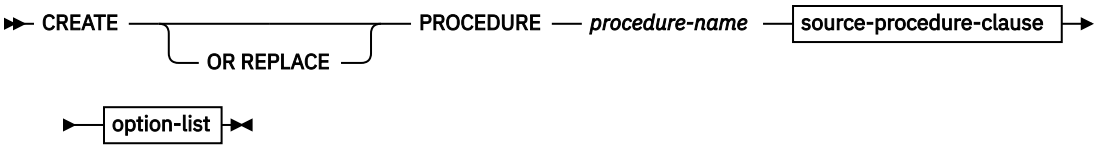
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- IMPLICIT_SCHEMA authority on the database, if the schema name of the procedure does not refer to an existing schema
- CREATEIN privilege on the schema, if the schema name of the procedure refers to an existing schema
- SCHEMAADM authority on the schema, if the schema name of the procedure refers to an existing schema
- DBADM authority

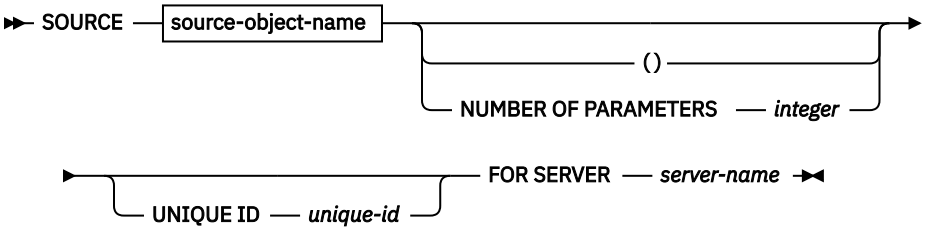
For data sources that require a user mapping, the privileges held at the data source by the authorization ID of the statement must include the privilege to select the procedure's description from the remote catalog tables.

To replace an existing procedure, the authorization ID of the statement must be the owner of the existing procedure (SQLSTATE 42501).

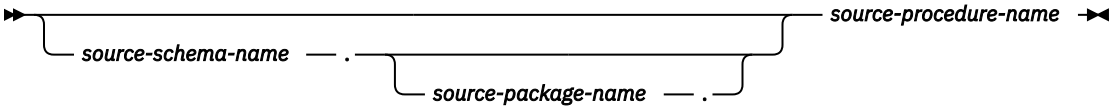
Syntax



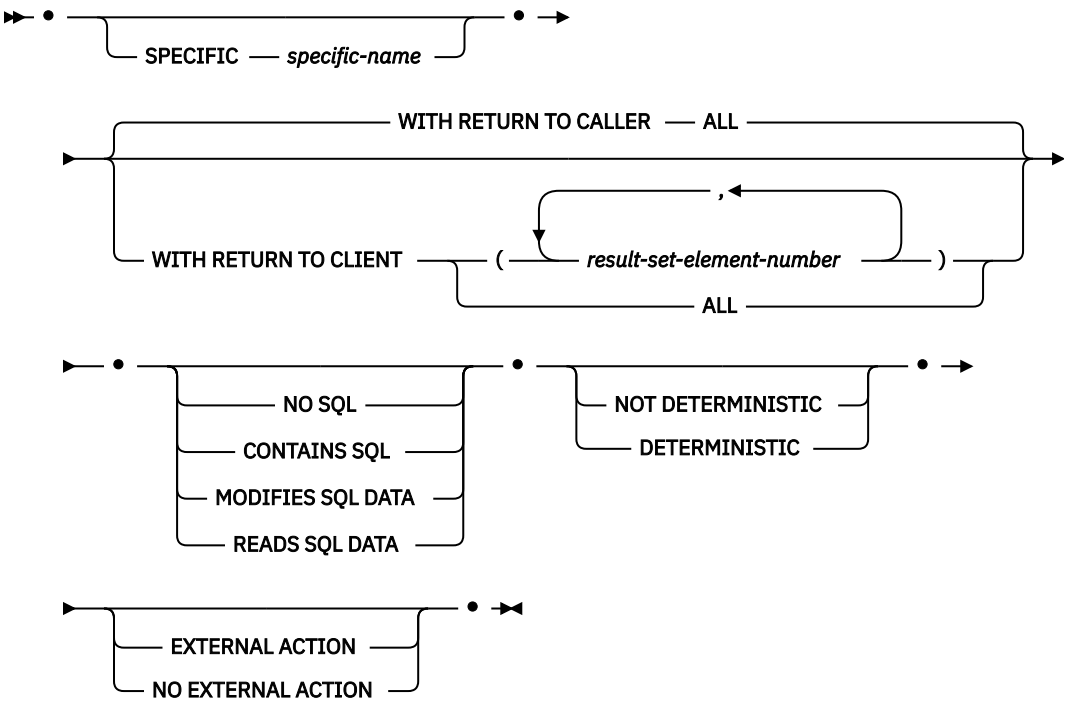
source-procedure-clause



source-object-name



option-list



Description

OR REPLACE
Specifies to replace the definition for the procedure if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog, with the exception that privileges that were granted on the procedure are not affected. This option can be specified only by the owner of the object. This option is ignored if a definition for the procedure does not exist at the current server. To replace an existing procedure, the specific name and procedure name of the new definition must be the same as the specific name and procedure name of the old definition, or the signature of the new definition must match the signature of the old definition. Otherwise, a new procedure is created.

procedure-name
Names the sourced procedure being defined. It is a qualified or unqualified name that designates a procedure. The unqualified form of *procedure-name* is an SQL identifier. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile or bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a *schema-name* followed by a period and an SQL identifier.

The name, including the implicit or explicit qualifiers, together with the number of parameters, must not identify a procedure that is described in the catalog (SQLSTATE 42723). The unqualified name, together with the number of parameters, need not be unique across schemas.

If a two-part name is specified, the *schema-name* cannot begin with 'SYS' (SQLSTATE 42939).

In a federated system, *procedure-name* is the name of the procedure on the federated server.

SOURCE *source-object-name*
Specifies the source procedure that is used by the procedure being defined. In a federated system, the source procedure is a procedure that is located at a supported data source.

source-schema-name
Identifies the schema name of the source procedure. If a schema name is used to identify the source procedure, the *source-schema-name* must be specified in the CREATE PROCEDURE (Sourced) statement. If the *source-schema-name* contains any special or lowercase characters, it must be enclosed by double quotation marks.

source-package-name
Identifies the package name of the source procedure. The *source-package-name* applies only to Oracle data sources. If a package name is used to identify the source procedure, the *source-package-name* must be specified in the CREATE PROCEDURE (Sourced) statement. If the *source-package-name* contains any special or lowercase characters, it must be enclosed by double quotation marks.

source-procedure-name
Identifies the procedure name of the source procedure. If the *source-procedure-name* contains any special or lowercase characters, it must be enclosed by double quotation marks.

( )
Indicates that the number of parameters is zero.

NUMBER OF PARAMETERS *integer*
Specifies the number of parameters for the source procedure. The minimum value for *integer* is 0, and the maximum value is 32 767.

UNIQUE ID *string-constant*
Provides a way to uniquely identify the source procedure when there are multiple procedures at the data source with the identical name, schema, and number of parameters. The *string-constant* value, which has a maximum length of 128, is interpreted uniquely by each data source.

FOR SERVER *server-name*
Specifies a server definition that was registered using the CREATE SERVER statement.

SPECIFIC *specific-name*
Provides a unique name for the instance of the sourced procedure that is being defined. This specific name can be used when altering, dropping, or commenting on the sourced procedure. This name can never be used to invoke the sourced procedure. The unqualified form of *specific-name* is an SQL identifier. The qualified form of *specific-name* is a *schema-name* followed by a period and an SQL identifier. The *specific-name* value, including the implicit or explicit qualifier, must not identify another procedure instance that exists at the application server; otherwise an error is returned (SQLSTATE 42710).

The *specific-name* can be the same as an existing *procedure-name*.

If no qualifier is specified, the qualifier that was used for *procedure-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier for *procedure-name*, or an error is returned (SQLSTATE 42882).

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is 'SQL' followed by a character timestamp: 'SQL*yymmddhhmmssxxx*'.

WITH RETURN TO CALLER or WITH RETURN TO CLIENT
Indicates where the result sets from the source procedure are handled. If the source procedure is not from an Oracle data source, the only one result set is returned to the caller or client; and if the source procedure is coded to return more than one result set, only the first result set is returned to the caller or client. The default is WITH RETURN TO CALLER.

WITH RETURN TO CALLER ALL
Specifies that all result sets from the source procedure are returned to the caller.

WITH RETURN TO CLIENT
Indicates which result sets from the source procedure are returned directly to the client application. The dynamic result set value at the data source must be greater than 0 for a result set to be returned.

(result-set-element-number, ...)
Specifies a non-empty list of result sets to return to the client application (SQLSTATE 42601). A *result-set-element-number* identifies a result set based on the order the result sets are returned, where 1 identifies the first result set, 2 the second result set, and so on. A *result-set-element-number* greater than the total number of result sets returned is ignored. Each *result-set-element-number* must be an integer value greater than zero (SQLSTATE 42815), and must not exceed the value of a small integer constant (SQLSTATE 42820). The list of result

sets to return to the client application must not contain duplicate values and must be specified in ascending order (SQLSTATE 42815). Result sets are always processed in the order they are returned from the source procedure.

Result sets that are not identified in the list to return to client application are returned to the caller.

Note: This list of result sets to return to the client application must only be used with source procedures that are known to consistently return result sets that are intended for the client in the same position in the list of result sets each time they are executed. It is possible for a source procedure to return different sets of result sets each time it is executed, depending on the internal logic of the procedure. If this is the case, then specify either WITH RETURN TO CALLER ALL or WITH RETURN TO CLIENT ALL instead, and code the application to handle this case.

ALL
Specifies all result sets from the source procedure are returned to the client.

NO SQL, CONTAINS SQL, MODIFIES SQL DATA, READS SQL DATA
Specifies the classification of SQL statements that can be run by this procedure, or any routine that is called by this procedure. The database manager verifies that the SQL statements issued by the procedure and all routines that are called by the procedure are consistent with this specification.

For the classification of each statement, see "SQL statements that can be executed in routines and triggers" in Developing User-defined Routines (SQL and External).

Because the source procedure for the sourced procedure is not on the federated server, the specified level is not enforced during execution of the source procedure at the data source. If there is discrepancy between what is specified for the sourced procedure and what the source procedure actually does at the data source, data inconsistency might occur.

If this option is not explicitly specified, the value for the source procedure is used.

If this option is explicitly specified but does not match the value for the source procedure, an error is returned (SQLSTATE 428GS).

If this option is not available at the data source, the default is MODIFIES SQL DATA.

NO SQL
Specifies that the procedure can run only SQL statements with a data access classification of NO SQL. (SQLSTATE 38001).

CONTAINS SQL
Specifies that the procedure can run only statements with a data access classification of CONTAINS SQL or NO SQL (SQLSTATE 38003 or 38004).

MODIFIES SQL DATA
Specifies that the procedure can run any SQL statement except statements that are not supported in procedures (SQLSTATE 38003).

READS SQL DATA
Specifies that the procedure can run statements with a data access classification of READS SQL DATA, CONTAINS SQL, or NO SQL (SQLSTATE 38002 or 38003 or 42985).

DETERMINISTIC or NOT DETERMINISTIC
Specifies whether the sourced procedure always returns the same results for given argument values (DETERMINISTIC), or whether the sourced procedure depends on some stated values that affect the results (NOT DETERMINISTIC). A DETERMINISTIC sourced procedure must always return the same result from successive invocations with identical inputs. This clause currently does not impact the processing of the procedure. If this option is not explicitly specified, the value for the source procedure is used. If this option is not available at the data source, the default is NOT DETERMINISTIC. If this option is explicitly specified, but does not match the value for the source procedure, an error is returned (SQLSTATE 428GS).

EXTERNAL ACTION or NO EXTERNAL ACTION
Specifies whether the sourced procedure takes some action that changes the state of an object that is not managed by the database manager (EXTERNAL ACTION), or does not (NO EXTERNAL ACTION). If the NO EXTERNAL ACTION clause is specified, the federated database uses optimization that assumes that the sourced procedure has no external impact. If this option is not explicitly specified, the value for the source procedure is used. If this option is not available at the data source, the default is EXTERNAL ACTION. If this option is explicitly specified but does not match the value for the source procedure, an error is returned (SQLSTATE 428GS).

Rules

- If the *source-object-name*, along with the NUMBER OF PARAMETERS and UNIQUE ID clauses do not identify a procedure at the data source, an error is returned (SQLSTATE 42883); if more than one procedure is identified, an error is returned (SQLSTATE 42725).
- If the UNIQUE ID clause is specified and the data source does not support unique IDs, an error is returned (SQLSTATE 42883).

Notes

- Before a federated procedure can be registered for a data source, the federated server must be configured to access that data source. This configuration includes: registering the wrapper for the data source, creating the server definition for the data source, and creating the user mappings between the federated server and the data source server for the data sources that require user mapping.
- ***Creating procedures that are initially invalid:*** If an object referenced in the procedure body does not exist or is marked invalid, or the definer temporarily doesn't have privileges to access the object, and if the database configuration parameter **auto_reval** is not set to DISABLED, then the procedure will still be created successfully. The procedure will be marked invalid and will be revalidated the next time it is invoked.
- Unlike SQL and external procedures defined at the federated server, federated procedures do not inherit the special registers of the caller, even those whose *remote-object-name* refers to a procedure on a Db2 data source.
- If the definition of the source procedure is changed (for example, a parameter data type is changed), the federated procedure should be dropped and recreated; otherwise, errors might occur when the federated procedure is invoked.
- If the length of the source procedure parameter is longer than 128, the parameter name of the federated procedure is truncated to 128 bytes.
- ***Compatibilities***: The DataJoiner syntax for Create Stored Procedure Nickname is not supported. Parameter type mapping is handled similarly to nicknames: A catalog look-up determines the remote data type. The local parameter type is determined through forward type mapping.

Examples

- *Example 1:* Create a federated procedure named FEDEMPLOYEE for an Oracle procedure named EMPLOYEE, using the remote schema name USER1, the remote package name P1 at the federated server S1, and returning the result set to the client.

```
CREATE PROCEDURE FEDEMPLOYEE SOURCE USER1.P1.EMPLOYEE
  FOR SERVER S1 WITH RETURN TO CLIENT ALL
```

- *Example 2:* Create a federated procedure named FEDSALARYSTAT for an Oracle procedure named SALARYSTAT, using the remote schema name USER1, the remote package name P1 at the federated server S1, and returning the first and the third result set to the client, and remaining result sets to the caller.

```
CREATE OR REPLACE PROCEDURE FEDSALARYSTAT SOURCE USER1.P1.SALARYSTAT
FOR SERVER S1 WITH RETURN TO CLIENT(1,3)
```

CREATE PROCEDURE (SQL)

The CREATE PROCEDURE (SQL) statement defines an SQL procedure at the current server.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

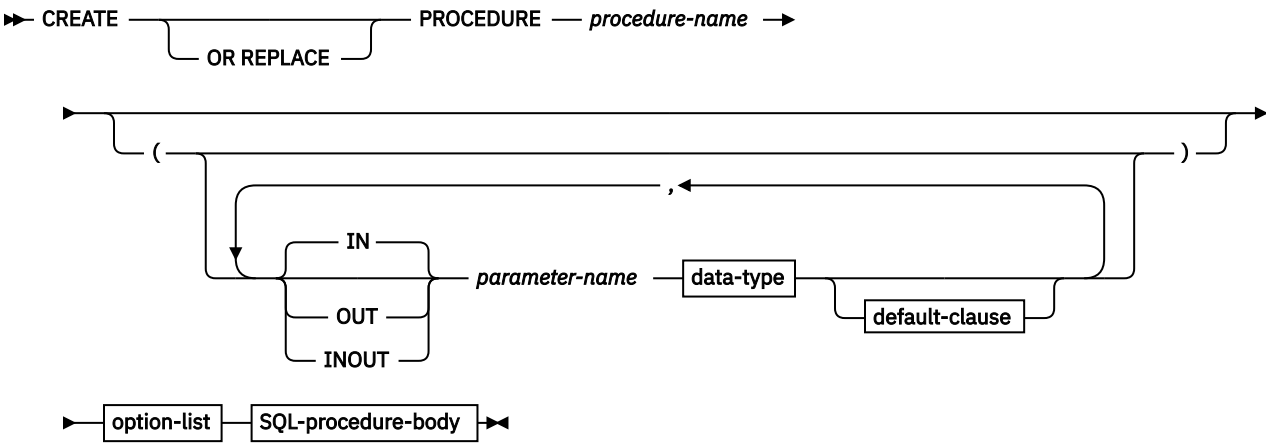
- If the implicit or explicit schema name of the procedure does not exist, IMPLICIT_SCHEMA authority on the database.
- If the schema name of the procedure refers to an existing schema, CREATEIN privilege on the schema.
- If the schema name of the procedure refers to an existing schema, SCHEMAADM authority on the schema.
- DBADM authority

The privileges held by the authorization ID of the statement must also include all of the privileges necessary to invoke the SQL statements that are specified in the procedure body.

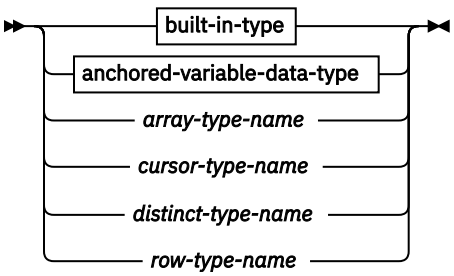
To replace an existing procedure, the authorization ID of the statement must be the owner of the existing procedure (SQLSTATE 42501).

Group privileges are not considered for any table or view specified in the CREATE PROCEDURE (SQL) statement.

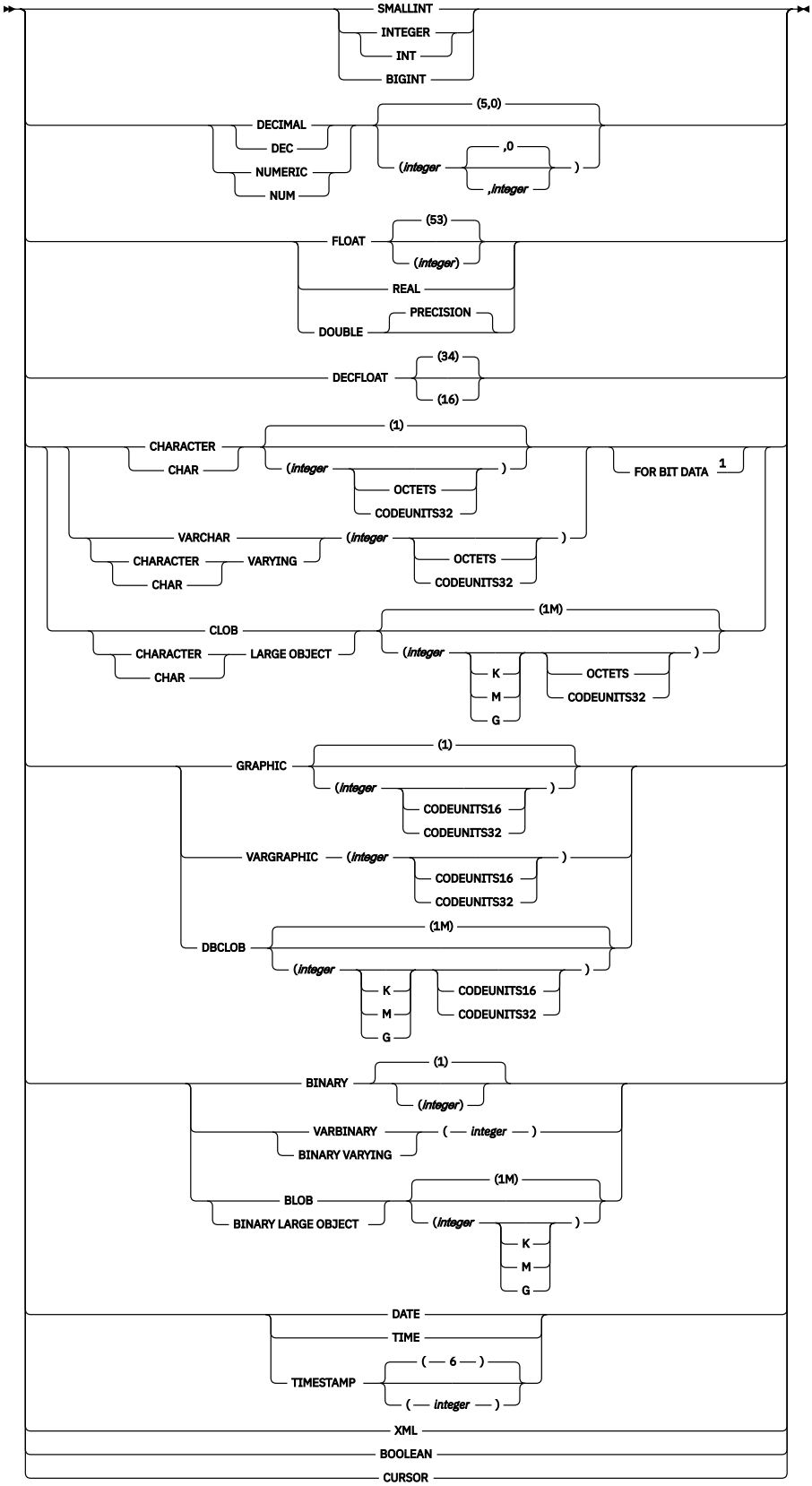
Syntax



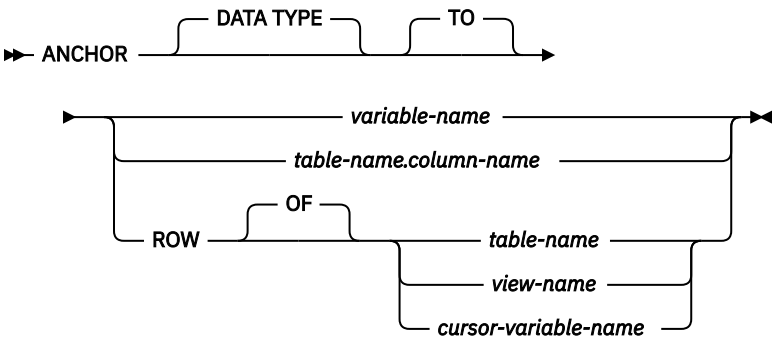
data-type



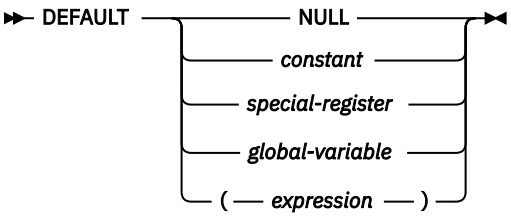
built-in-type



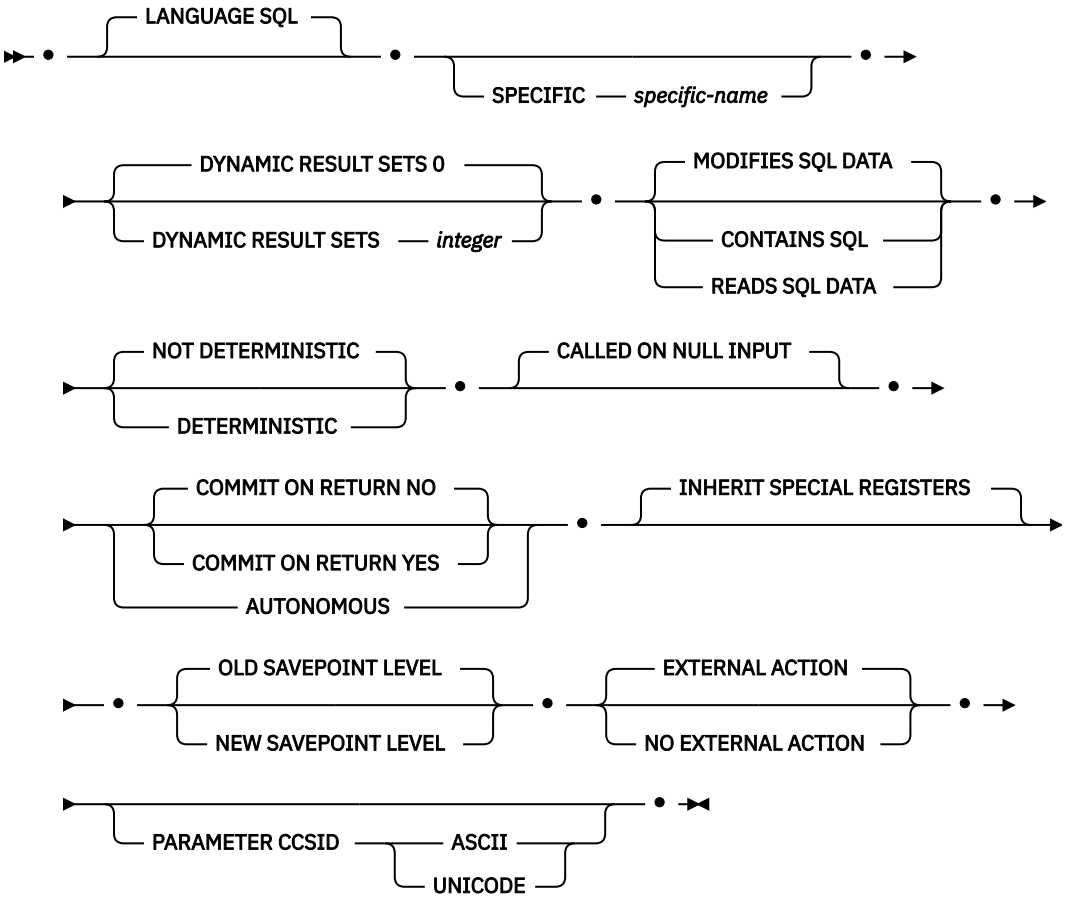
anchored-data-type



default-clause



option-list



SQL-procedure-body

Notes:

[1] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

Description

OR REPLACE
Specifies to replace the definition for the procedure if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog, with the exception that privileges that were granted on the procedure are not affected. This option can be specified only by the owner of the object. This option is ignored if a definition for the procedure does not exist at the current server. To replace an existing procedure, the specific name and procedure name of the new definition must be the same as the specific name and procedure name of the old definition, or the signature of the new definition must match the signature of the old definition. Otherwise, a new procedure is created.

procedure-name
Names the procedure being defined. It is a qualified or unqualified name that designates a procedure. The unqualified form of *procedure-name* is an SQL identifier. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a *schema-name* followed by a period and an SQL identifier.

The name, including the implicit or explicit qualifiers, together with the number of parameters, must not identify a procedure described in the catalog (SQLSTATE 42723). The unqualified name, together with the number of parameters, is unique within its schema, but does not need to be unique across schemas.

If a two-part name is specified, the *schema-name* cannot begin with 'SYS'; otherwise, an error is returned (SQLSTATE 42939).

(*IN | OUT | INOUT *parameter-name data-type default-clause,...)
Identifies the parameters of the procedure, and specifies the mode, name, data type, and optional default value of each parameter. One entry in the list must be specified for each parameter that the procedure will expect.

It is possible to register a procedure that has no parameters. In this case, the parentheses must still be coded, with no intervening data types. For example:

```
CREATE PROCEDURE SUBWOOFER() ...
```

No two identically-named procedures within a schema are permitted to have exactly the same number of parameters. A duplicate signature raises an SQL error (SQLSTATE 42723).

For example, given the statements:

```
CREATE PROCEDURE PART (IN NUMBER INT, OUT PART_NAME CHAR(35)) ...
CREATE PROCEDURE PART (IN COST DECIMAL(5,3), OUT COUNT INT) ...
```

the second statement will fail because the number of parameters in the procedure is the same, even if the data types are not.

IN | OUT | INOUT
Specifies the mode of the parameter.

If an error is returned by the procedure, OUT parameters are undefined and INOUT parameters are unchanged.

IN
Identifies the parameter as an input parameter to the procedure. Any changes made to the parameter within the procedure are not available to the calling SQL application when control is returned. The default is IN.

OUT
Identifies the parameter as an output parameter for the procedure.

INOUT
Identifies the parameter as both an input and output parameter for the procedure.

parameter-name
Specifies the name of the parameter. The parameter name must be unique for the procedure (SQLSTATE 42734).

data-type
Specifies the data type of the parameter. A structured type or reference type cannot be specified (SQLSTATE 429BB).

built-in-type
Specifies a built-in data type. For a more complete description of each built-in data type except BOOLEAN and CURSOR, which cannot be specified for a table, see "CREATE TABLE".

BOOLEAN
For a Boolean.

CURSOR
For a reference to an underlying cursor.

anchored-data-type
Identifies another object used to define the data type. The data type of the anchor object has the same limitations that apply to specifying the data type directly, or in the case of a row, to creating a row type.

ANCHOR DATA TYPE TO
Indicates an anchored data type is used to specify the data type.

variable-name
Identifies a global variable. The data type of the global variable is used as the data type for *parameter-name*.

table-name.column-name
Identifies a column name of an existing table or view. The data type of the column is used as the data type for *parameter-name*.

ROW OF *table-name* or *view-name*
Specifies a row of fields with names and data types that are based on the column names and column data types of the table identified by *table-name* or the view identified by *view-name*. The data type of *parameter-name* is an unnamed row type.

ROW OF *cursor-variable-name*
Specifies a row of fields with names and data types that are based on the field names and field data types of the cursor variable identified by *cursor-variable-name*. The specified cursor variable must be one of the following elements (SQLSTATE 428HS):

- A global variable with a strongly typed cursor data type
- A global variable with a weakly typed cursor data type that was created or declared with a CONSTANT clause specifying a *select-statement* where all the result columns are named.

If the cursor type of the cursor variable is not strongly-typed using a named row type, the data type of *parameter-name* is an unnamed row type.

array-type-name
Specifies the name of a user-defined array type. If *array-type-name* is specified without a schema name, the array type is resolved by searching the schemas in the SQL path.

cursor-type-name
Specifies the name of a cursor type. If *cursor-type-name* is specified without a schema name, the cursor type is resolved by searching the schemas in the SQL path.

distinct-type-name
Specifies the name of a distinct type. The length, precision, and scale of the parameter are, respectively, the length, precision, and scale of the source type of the distinct type. A distinct type parameter is passed as the source type of the distinct type. If *distinct-type-name* is specified without a schema name, the distinct type is resolved by searching the schemas in the SQL path.

row-type-name
Specifies the name of a user-defined row type. The fields of the parameter are the fields of the row type. If *row-type-name* is specified without a schema name, the row type is resolved by searching the schemas in the SQL path.

DEFAULT
Specifies a default value for the parameter. The default can be a constant, a special register, a global variable, an expression or the keyword NULL. The special registers that can be specified as the default are that same as those that can be specified for a column default (see *default-clause* in the CREATE TABLE statement). Other special registers can be specified as the default by using an expression.

The *expression* can be any expression of the type described in "Expressions". If a default value is not specified, the parameter has no default and the corresponding argument cannot be omitted on invocation of the procedure. The maximum size of the *expression* is 64K bytes.

The default expression must not modify SQL data (SQLSTATE 428FL or SQLSTATE 429BL). The expression must be assignment compatible to the parameter data type (SQLSTATE 42821).

A default cannot be specified in the following situations:

- For INOUT or OUT parameters (SQLSTATE 42601)
- For a parameter of type ARRAY, ROW, or CURSOR (SQLSTATE 429BB)

SPECIFIC *specific-name*
Provides a unique name for the instance of the procedure that is being defined. This specific name can be used when altering, dropping, or commenting on the procedure. It can never be used to invoke the procedure. The unqualified form of *specific-name* is an SQL identifier. The qualified form is a *schema-name* followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another procedure instance that exists at the application server; otherwise an error (SQLSTATE 42710) is raised.

The *specific-name* can be the same as an existing *procedure-name*.

If no qualifier is specified, the qualifier that was used for *procedure-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier for *procedure-name*, or an error (SQLSTATE 42882) is raised.

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is 'SQL' followed by a character timestamp: 'SQL*yymmddhhmmssxxx*'.

If you intend to archive the procedure by using the **GET ROUTINE** command, ensure the specific-name has a maximum length of 18 characters.

DYNAMIC RESULT SETS *integer*
Indicates the estimated upper bound of returned result sets for the procedure.

MODIFIES SQL DATA, CONTAINS SQL, READS SQL DATA
Specifies the classification of SQL statements that can be run by this procedure or any routine that is called by this procedure. The database manager verifies that the SQL statements issued by the procedure and all routines that are called by the procedure are consistent with this specification.

For the classification of each statement, see "SQL statements that can be executed in routines and triggers" in Developing User-defined Routines (SQL and External).

The default is MODIFIES SQL DATA.

MODIFIES SQL DATA
Specifies that the procedure can run any SQL statement except statements that are not supported in procedures (SQLSTATE 38003 or 42985).

CONTAINS SQL
Specifies that the procedure can run only statements with a data access classification of CONTAINS SQL (SQLSTATE 38003 or 38004 or 42985).

READS SQL DATA
Specifies that the procedure can run statements with a data access classification of READS SQL DATA or CONTAINS SQL (SQLSTATE 38002 or 38003 or 42985).

If the BEGIN ATOMIC clause is used in a compound SQL procedure, the procedure can be created only if it is defined as MODIFIES SQL DATA.

DETERMINISTIC or NOT DETERMINISTIC
This clause specifies whether the procedure always returns the same results for given argument values (DETERMINISTIC) or whether the procedure depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC procedure must always return the same result from successive invocations with identical inputs.

This clause currently does not impact processing of the procedure.

CALLED ON NULL INPUT
CALLED ON NULL INPUT always applies to procedures. This means that the procedure is called regardless of whether any arguments are null. Any OUT or INOUT parameter can return a null value or a normal (non-null) value. Responsibility for testing for null argument values lies with the procedure.

COMMIT ON RETURN
Indicates whether a commit is to be issued on return from the procedure. The default is NO.

NO
A commit is not issued when the procedure returns.

YES
A commit is issued when the procedure returns if a positive SQLCODE is returned by the CALL statement

The commit operation includes the work that is performed by the calling application process and the procedure.

If the procedure returns result sets, the cursors that are associated with the result sets must have been defined as WITH HOLD to be usable after the commit.

AUTONOMOUS
Indicates the procedure should execute in its own autonomous transaction scope.

INHERIT SPECIAL REGISTERS
This optional clause specifies that updatable special registers in the procedure will inherit their initial values from the environment of the invoking statement. For a routine invoked in a nested object (for example a trigger or view), the initial values are inherited from the runtime environment (not inherited from the object definition).

No changes to the special registers are passed back to the caller of the procedure.

Non-updatable special registers, such as the datetime special registers, reflect a property of the statement currently executing, and are therefore set to their default values.

OLD SAVEPOINT LEVEL or NEW SAVEPOINT LEVEL
Specifies whether or not this procedure establishes a new savepoint level for savepoint names and effects. OLD SAVEPOINT LEVEL is the default behavior. For more information about savepoint levels, see "Rules" in "SAVEPOINT".

LANGUAGE SQL
This clause is used to specify that the procedure body is written in the SQL language.

EXTERNAL ACTION or NO EXTERNAL ACTION
Specifies whether the procedure takes some action that changes the state of an object not managed by the database manager (EXTERNAL ACTION), or not (NO EXTERNAL ACTION). The default is EXTERNAL ACTION. If NO EXTERNAL ACTION is specified, the system can use certain optimizations that assume the procedure has no external impact.

PARAMETER CCSID
Specifies the encoding scheme to use for all string data passed into and out of the procedure. If the PARAMETER CCSID clause is not specified, the default is PARAMETER CCSID UNICODE for Unicode databases, and PARAMETER CCSID ASCII for all other databases.

ASCII
Specifies that string data is encoded in the database code page. If the database is a Unicode database, PARAMETER CCSID ASCII cannot be specified (SQLSTATE 56031).

UNICODE
Specifies that character data is in UTF-8, and that graphic data is in UCS-2. If the database is not a Unicode database, PARAMETER CCSID UNICODE cannot be specified (SQLSTATE 56031).

SQL-procedure-body
Specifies the SQL statement that is the body of the SQL procedure.

See *SQL-procedure-statement* in "Compound SQL (Compiled)" statement.

Rules

- ***Autonomous routine restrictions:*** Autonomous routines cannot return result sets and do not support the following data types (SQLSTATE 428H2):
  - User-defined cursor types
  - User-defined structured types
  - XML as IN, OUT, and INOUT parameters

  Session variables of cursor types cannot be referenced within the autonomous scope.
- ***Use of anchored data types:*** An anchored data type cannot refer to the following objects (SQLSTATE 428HS): a nickname, typed table, typed view, statistical view that is associated with an expression-based index, declared temporary table, row definition that is associated with a weakly typed cursor, object with a code page or collation that is different from the database code page or database collation.
- ***Use of cursor and row types:*** A procedure that uses a cursor type or row type for a parameter can be invoked only from within a compound SQL (compiled) statement (SQLSTATE 428H2), except for Java applications using JDBC, which can invoke a procedure with OUT parameters that have a cursor type. Invocation from Java external procedures is not supported.

Notes

- Creating a procedure with a schema name that does not already exist will result in the implicit creation of that schema, provided that the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- A procedure that is called from within a compound SQL (inlined) statement will execute as if it were created specifying NEW SAVEPOINT LEVEL, even if OLD SAVEPOINT LEVEL was specified or defaulted to when the procedure was created.
- ***Creating procedures that are initially invalid:*** If an object referenced in the procedure body does not exist or is marked invalid, or the definer temporarily doesn't have privileges to access the object, and if the database configuration parameter **auto_reval** is not set to DISABLED, then the procedure will still be created successfully. The procedure will be marked invalid and will be revalidated the next time it is invoked.
- ***Setting of the default value:*** Parameters of a procedure that are defined with a default value are set to their default value when the procedure is invoked, but only if a value is not supplied for the corresponding argument, or is specified as DEFAULT, when the procedure is invoked.
- ***Privileges:*** The definer of a procedure always receives the EXECUTE privilege WITH GRANT OPTION on the procedure, as well as the right to drop the procedure.
- ***Rebinding dependent packages:*** Every SQL procedure has a dependent package. The package can be rebound at any time by running the REBIND_ROUTINE_PACKAGE procedure. Explicitly rebinding the dependent package does not revalidate an invalid procedure. An invalid procedure should be

revalidated with automatic revalidation or by explicitly running the ADMIN_REVALIDATE_DB_OBJECTS procedure. Procedure revalidation automatically rebinds the dependent package.

- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - RESULT SETS can be specified in place of DYNAMIC RESULT SETS.
  - NULL CALL can be specified in place of CALLED ON NULL INPUT.

  The following syntax is accepted as the default behavior:
  - ASUTIME NO LIMIT
  - NO COLLID
  - STAY RESIDENT NO

Example

Create an SQL procedure that returns the median staff salary. Return a result set containing the name, position, and salary of all employees who earn more than the median salary.

```
CREATE PROCEDURE MEDIAN_RESULT_SET (OUT medianSalary DOUBLE)
  RESULT SETS 1
  LANGUAGE SQL
BEGIN
  DECLARE v_numRecords INT DEFAULT 1;
  DECLARE v_counter INT DEFAULT 0;

  DECLARE c1 CURSOR FOR
    SELECT CAST(salary AS DOUBLE)
      FROM staff
      ORDER BY salary;
  DECLARE c2 CURSOR WITH RETURN FOR
    SELECT name, job, CAST(salary AS INTEGER)
      FROM staff
      WHERE salary > medianSalary
      ORDER BY salary;

  DECLARE EXIT HANDLER FOR NOT FOUND
    SET medianSalary = 6666;

  SET medianSalary = 0;
  SELECT COUNT(*) INTO v_numRecords
    FROM STAFF;
  OPEN c1;
  WHILE v_counter < (v_numRecords / 2 + 1)
  DO
    FETCH c1 INTO medianSalary;
    SET v_counter = v_counter + 1;
  END WHILE;
  CLOSE c1;
  OPEN c2;
END
```

CREATE ROLE

The CREATE ROLE statement defines a role at the current server.

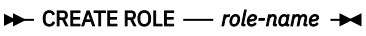
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include SECADM authority.

Syntax



Description

role-name

Names the role. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The name must not identify an existing role at the current server (SQLSTATE 42710). The name must not begin with the characters 'SYS' and must not be 'ACCESSCTRL', 'DATAACCESS', 'DBADM', 'NONE', 'NULL', 'PUBLIC', 'SECADM', 'SQLADM', 'SCHEMAADM', or 'WLMADM' (SQLSTATE 42939).

Example

Create a role named DOCTOR.

```
CREATE ROLE DOCTOR
```

CREATE SCHEMA

The CREATE SCHEMA statement defines a schema. It is also possible to create some objects and grant privileges on objects within the statement.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

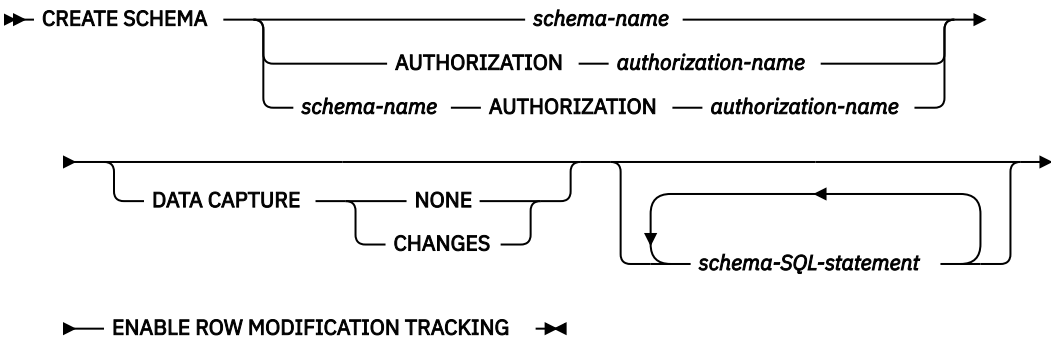
An authorization ID that holds DBADM authority can create a schema with any valid *schema-name* or *authorization-name*.

An authorization ID that does not hold DBADM authority can only create a schema with a *schema-name* or *authorization-name* that matches the authorization ID of the statement.

If the statement includes a *schema-SQL-statement*, the privileges held by the *authorization-name* (which, if not specified, defaults to the authorization ID of the statement) must include at least one of the following authorities:

- The privileges required to perform each *schema-SQL-statement*
- DBADM authority

Syntax



Description

schema-name
An identifier that names the schema. The name must not identify a schema already described in the catalog (SQLSTATE 42710). The name cannot begin with 'SYS' (SQLSTATE 42939). The owner of the schema is the authorization ID that issued the statement.

AUTHORIZATION *authorization-name*
Identifies the user who is the owner of the schema. The value of *authorization-name* is also used to name the schema. The *authorization-name* must not identify a schema already described in the catalog (SQLSTATE 42710).

schema-name* AUTHORIZATION *authorization-name
Identifies a schema called *schema-name*, whose owner is *authorization-name*. The *schema-name* must not identify a schema already described in the catalog (SQLSTATE 42710). The *schema-name* cannot begin with 'SYS' (SQLSTATE 42939).

DATA CAPTURE
Indicates whether extra information for data replication is to be written to the log. The default is determined based on the value of database configuration parameter **dft_schemas_dcc**. If the value is "Yes" the default is CHANGES, otherwise the default is NONE.

NONE
Indicates that no extra information for data replication will be logged.

CHANGES
Indicates that extra information regarding SQL changes to this schema will be written to the log. This option is required if this schema will be replicated and a replication capture program is used to capture changes for this schema from the log.

schema-SQL-statement
SQL statements that can be included as part of the CREATE SCHEMA statement are:

- CREATE TABLE statement, excluding typed tables and materialized query tables
- CREATE VIEW statement, excluding typed views
- CREATE INDEX statement
- COMMENT statement
- GRANT statement

ENABLE ROW MODIFICATION TRACKING
Indicates tables created in the schema are to be enabled for logical backup. Applies only to columnar organized tables. For a list of restrictions, see Schema enabled for row modification tracking.

Notes

- The owner of the schema is determined as follows:
  - If an AUTHORIZATION clause is specified, the specified *authorization-name* is the schema owner
  - If an AUTHORIZATION clause is not specified, the authorization ID that issued the CREATE SCHEMA statement is the schema owner.
- The schema owner is assumed to be a user (not a group).
- When the schema is explicitly created with the CREATE SCHEMA statement, the schema owner is granted CREATEIN, DROPIN, and ALTERIN privileges on the schema with the ability to grant these privileges to other users.
- The definer of any object created as part of the CREATE SCHEMA statement is the schema owner. The schema owner is also the grantor for any privileges granted as part of the CREATE SCHEMA statement.
- Unqualified object names in any SQL statement within the CREATE SCHEMA statement are implicitly qualified by the name of the created schema.
- Schema names that are shorter than 8-bytes are padded with blanks and stored in the catalog as 8-byte names.

- If the CREATE statement contains a qualified name for the object being created, the schema name specified in the qualified name must be the same as the name of the schema being created (SQLSTATE 42875). Any other objects referenced within the statements may be qualified with any valid schema name.
- It is recommended not to use "SESSION" as a schema name. Since declared temporary tables must be qualified by "SESSION", it is possible to have an application declare a temporary table with a name identical to that of a persistent table. An SQL statement that references a table with the schema name "SESSION" will resolve (at statement compile time) to the declared temporary table rather than a persistent table with the same name. Since an SQL statement is compiled at different times for static embedded and dynamic embedded SQL statements, the results depend on when the declared temporary table is defined. If persistent tables, views or aliases are not defined with a schema name of "SESSION", these issues do not require consideration.
- Setting the DATA CAPTURE attribute at the schema level causes newly created tables to inherit the DATA CAPTURE attribute from the schema if one is not specified at the table level.

Examples

- *Example 1:* As a user with DBADM authority, create a schema called RICK with the user RICK as the owner.

```
CREATE SCHEMA RICK AUTHORIZATION RICK
```

- *Example 2:* Create a schema that has an inventory part table and an index over the part number. Give authority on the table to user JONES.

```
CREATE SCHEMA INVENTRY

  CREATE TABLE PART (PARTNO   SMALLINT NOT NULL,
                     DESCR    VARCHAR(24),
                     QUANTITY INTEGER)

  CREATE INDEX PARTIND ON PART (PARTNO)

  GRANT ALL ON PART TO JONES
```

- *Example 3:* Create a schema called PERS with two tables that each have a foreign key that references the other table. This is an example of a feature of the CREATE SCHEMA statement that allows such a pair of tables to be created without the use of the ALTER TABLE statement.

```
CREATE SCHEMA PERS

  CREATE TABLE ORG (DEPTNUMB  SMALLINT NOT NULL,
                    DEPTNAME VARCHAR(14),
                    MANAGER  SMALLINT,
                    DIVISION VARCHAR(10),
                    LOCATION VARCHAR(13),
                    CONSTRAINT PKEYDNO
                      PRIMARY KEY (DEPTNUMB),
                    CONSTRAINT FKEYMGR
                      FOREIGN KEY (MANAGER)
                      REFERENCES STAFF (ID) )

  CREATE TABLE STAFF (ID        SMALLINT NOT NULL,
                      NAME     VARCHAR(9),
                      DEPT     SMALLINT,
                      JOB      VARCHAR(5),
                      YEARS    SMALLINT,
                      SALARY   DECIMAL(7,2),
                      COMM     DECIMAL(7,2),
                      CONSTRAINT PKEYID
                        PRIMARY KEY (ID),
                      CONSTRAINT FKEYDNO
                        FOREIGN KEY (DEPT)
                        REFERENCES ORG (DEPTNUMB) )
```

Related information

Schema enabled for row modification tracking

CREATE SECURITY LABEL COMPONENT

The CREATE SECURITY LABEL COMPONENT statement defines a component that is to be used as part of a security policy.

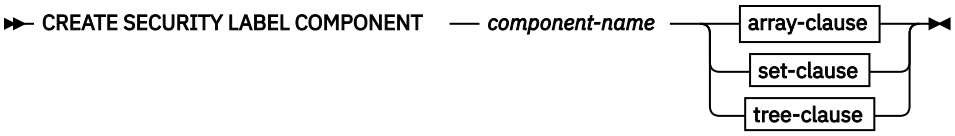
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

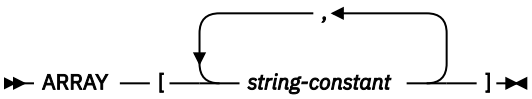
Authorization

The privileges held by the authorization ID of the statement must include SECADM authority.

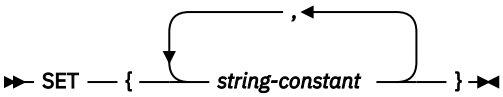
Syntax



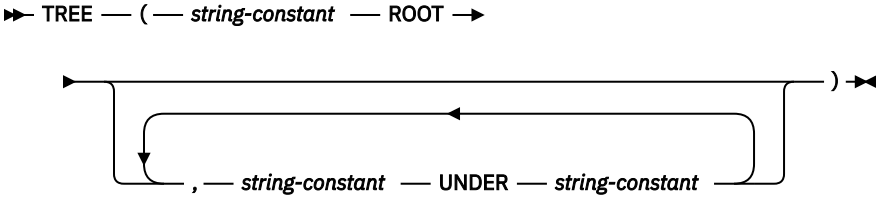
array-clause



set-clause



tree-clause



Description

component-name
Names the security label component. This is a one-part name. The name must not identify an existing security label component at the current server (SQLSTATE 42710).

ARRAY
Specifies an ordered set of elements.

***string-constant*,...**
One or more string constant values that make up the set of valid values for this security label component. The order in which the array elements appear is important. The first element ranks higher than the second element. The second element ranks higher than the third element and so on.

SET
Specifies an unordered set of elements.

***string-constant*,...**
One or more string constant values that make up the set of valid values for this security label component. The order of the elements is not important.

TREE
Specifies a tree structure of node elements.

string-constant
One or more string constant values that make up the set of valid values for this security label component.

ROOT
Specifies that the *string-constant* that follows the keyword is the root node element of the tree.

UNDER
Specifies that the *string-constant* before the UNDER keyword is a child of the *string-constant* that follows the UNDER keyword. An element must be defined as either being the root element or as being the child of another element before it can be used as a parent, otherwise an error (SQLSTATE 42704) is returned.

Rules

These rules apply to all three types of component (ARRAY, SET, and TREE):

- Element names cannot contain any of these characters:
  - Opening parenthesis - (
  - Closing parenthesis -)
  - Comma - ,
  - Colon - :
- An element name can have no more than 32 bytes (SQLSTATE 42622).
- If a security label component is a set or a tree, no more than 64 elements can be part of that component.
- A CREATE SECURITY LABEL COMPONENT statement can specify at most 65 535 elements for a security label component of type array.
- No element name can be used more than once in the same component (SQLSTATE 42713).

Examples

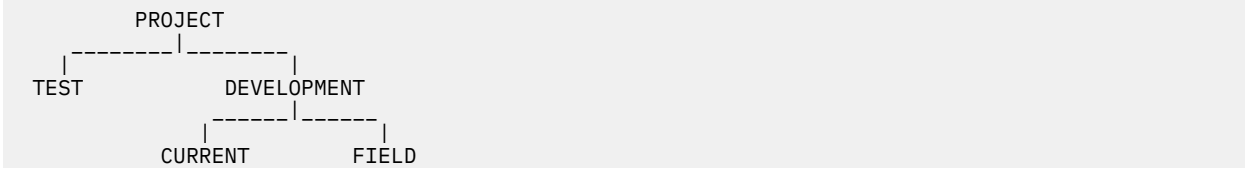
- *Example 1:* Create an ARRAY type security label component named LEVEL. The component has the following four elements, listed in order of decreasing rank: Top Secret, Secret, Classified, and Unclassified.

```
CREATE SECURITY LABEL COMPONENT LEVEL
  ARRAY ['Top Secret', 'Secret', 'Classified', 'Unclassified']
```

- *Example 2:* Create a SET type security label component named COMPARTMENTS. The component has the following three elements: Research, Analysis, and Collection.

```
CREATE SECURITY LABEL COMPONENT COMPARTMENTS
  SET {'Collection', 'Research', 'Analysis'}
```

- *Example 3:* Create a TREE type security label component named GROUPS. GROUPS has five elements: PROJECT, TEST, DEVELOPMENT, CURRENT, AND FIELD. The following diagram shows the relationship of these elements to one another:



```
CREATE SECURITY LABEL COMPONENT GROUPS
  TREE (
    'PROJECT' ROOT,
    'TEST' UNDER 'PROJECT',
    'DEVELOPMENT' UNDER 'PROJECT',
    'CURRENT' UNDER 'DEVELOPMENT',
    'FIELD' UNDER 'DEVELOPMENT'
  )
```

CREATE SECURITY LABEL

The CREATE SECURITY LABEL statement defines a security label.

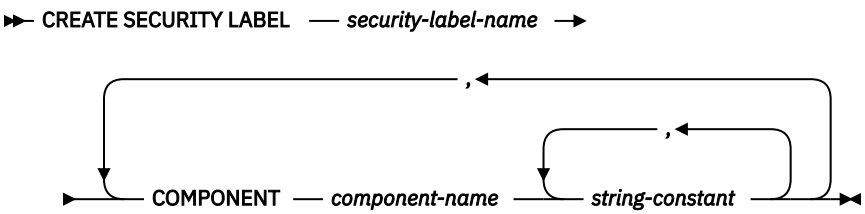
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include SECADM authority.

Syntax



Description

security-label-name
Names the security label. The name must be qualified with a security policy (SQLSTATE 42704), and must not identify an existing security label for this security policy (SQLSTATE 42710).

COMPONENT *component-name*
Specifies the name of a security label component. If the component is not part of the security policy *security-policy-name*, an error is returned (SQLSTATE 4274G). If a component is specified twice in the same statement, an error is returned (SQLSTATE 42713).

***string-constant*,...**
Specifies a valid element for the security component. A valid element is one that was specified when the security component was created. If the element is invalid, an error is returned (SQLSTATE 4274F).

Examples

- *Example 1:* Create a security label named EMPLOYEESECLABEL that is part of the DATA_ACCESS security policy, and that has the element Top Secret for the LEVEL component and the elements Research and Analysis for the COMPARTMENTS component.

```
CREATE SECURITY LABEL DATA_ACCESS.EMPLOYEESECLABEL
  COMPONENT LEVEL 'Top Secret',
  COMPONENT COMPARTMENTS 'Research', 'Analysis'
```

- *Example 2:* Create a security label named EMPLOYEESECLABELREAD that has the element Top Secret for the LEVEL component and the element Research for the COMPARTMENTS component.

```
CREATE SECURITY LABEL DATA_ACCESS.EMPLOYEESECLABELREAD
  COMPONENT LEVEL 'Top Secret',
  COMPONENT COMPARTMENTS 'Research'
```

- *Example 3:* Create a security label named EMPLOYEESECLABELWRITE that has the element Analysis for the COMPARTMENTS component and a null value for the LEVEL component. Assume that the security policy named DATA_ACCESS is the same security policy that is used in examples 1 and 2.

```
CREATE SECURITY LABEL DATA_ACCESS.EMPLOYEESECLABELWRITE
  COMPONENT COMPARTMENTS 'Analysis'
```

- *Example 4:* Create a security label named BEGINNER that is part of an existing CLASSPOLICY security policy, and that has the element Trainee for the TRUST component and the element Morning for the SECTIONS component.

```
CREATE SECURITY LABEL CLASSPOLICY.BEGINNER
  COMPONENT TRUST 'Trainee',
  COMPONENT SECTIONS 'Morning'
```

CREATE SECURITY POLICY

The CREATE SECURITY POLICY statement defines a security policy.

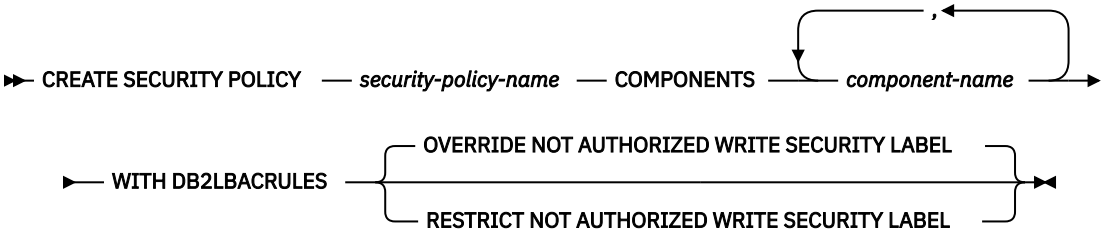
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include SECADM authority.

Syntax



Description

security-policy-name
Names the security policy. This is a one-part name. The name must not identify an existing security policy at the current server (SQLSTATE 42710).

COMPONENTS *component-name,...*
Identifies a security label component. The name must identify a security label component that already exists at the current server (SQLSTATE 42704). The same security component must not be specified more than once for the security policy (SQLSTATE 42713). No more than 16 security label components can be specified for a security policy (SQLSTATE 54062).

WITH DB2LBACRULES
Indicates what rule set that will be used when comparing security labels that are part of this security policy. There is currently only one rule set: DB2LBACRULES.

OVERRIDE NOT AUTHORIZED WRITE SECURITY LABEL or RESTRICT NOT AUTHORIZED WRITE SECURITY LABEL

Specifies the action that is to be taken when a user is not authorized to write the explicitly specified security label that is provided in the INSERT or UPDATE statement issued against a table that is protected with this security policy. A user's security label and exemption credentials determine the user's authorization to write an explicitly provided security label. The default is OVERRIDE NOT AUTHORIZED WRITE SECURITY LABEL.

OVERRIDE NOT AUTHORIZED WRITE SECURITY LABEL

Indicates that the value of the user's security label, rather than the explicitly specified security label, is to be used for write access during an insert or update operation.

RESTRICT NOT AUTHORIZED WRITE SECURITY LABEL

Indicates that the insert or update operation will fail if the user is not authorized to write the explicitly specified security label that is provided in the INSERT or UPDATE statement (SQLSTATE 42519).

Notes

- ***DB2LBACRULES rule set:*** DB2LBACRULES is a predefined set of rules that includes the following rules: DB2LBACREADARRAY, DB2LBACREADSET, DB2LBACREADTREE, DB2LBACWRITEARRAY, DB2LBACWRITESET, DB2LBACWRITETREE.
- Group and role authorizations are not considered by default when a security policy is created. Use the ALTER SECURITY POLICY statement to change this behavior and have them considered.

Examples

- *Example 1:* Create a security policy named DATA_ACCESS that uses the DB2LBACRULES rule set and has two components: LEVEL and COMPARTMENTS, in that order. Assume that both components already exist.

```
CREATE SECURITY POLICY DATA_ACCESS
  COMPONENTS LEVEL, COMPARTMENTS
  WITH DB2LBACRULES
```

- *Example 2:* Create a security policy named CONTRIBUTIONS that has the components MEMBER and BADGE, which are assumed to already exist.

```
CREATE SECURITY POLICY CONTRIBUTIONS
  COMPONENTS MEMBER, BADGE
  WITH DB2LBACRULES
```

CREATE SEQUENCE

The CREATE SEQUENCE statement defines a sequence at the application server.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

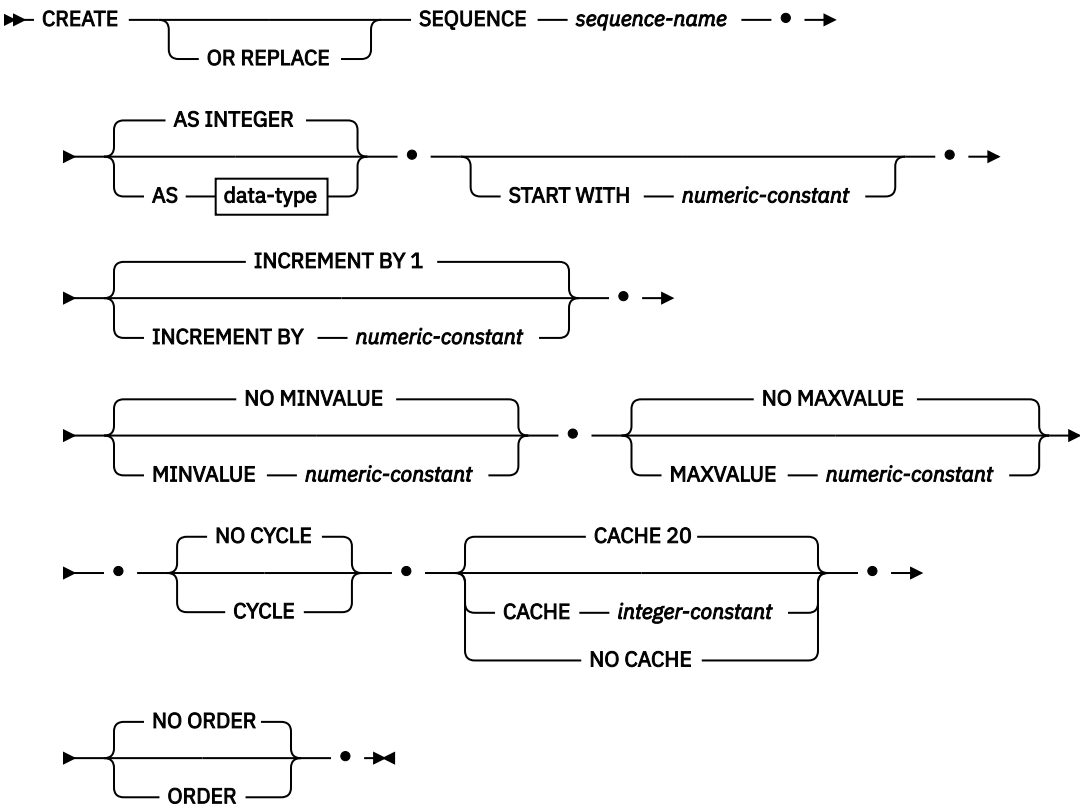
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the sequence does not exist
- CREATEIN privilege on the schema, if the schema name of the sequence refers to an existing schema

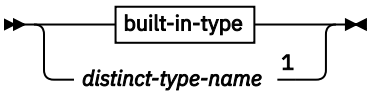
- SCHEMAADM authority on the schema, if the schema name of the sequence refers to an existing schema
- DBADM authority

To replace an existing sequence, the authorization ID of the statement must be the owner of the existing sequence (SQLSTATE 42501).

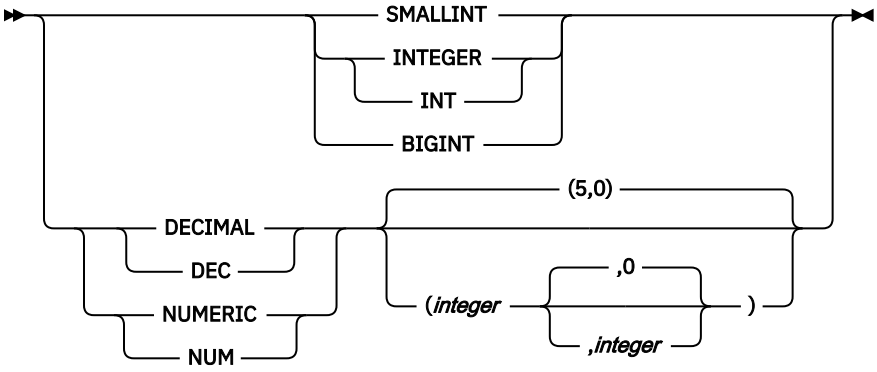
Syntax



data-type



built-in type



Notes:

[1] The specified distinct type cannot have any data type constraints and the source type cannot be an anchored data type.

Description

OR REPLACE
Specifies to replace the definition for the sequence if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog, with the exception that privileges that were granted on the sequence are not affected. This option is ignored if a definition for the sequence does not exist at the current server. This option can be specified only by the owner of the object.

sequence-name
Names the sequence. The combination of name, and the implicit or explicit schema name must not identify an existing sequence at the current server (SQLSTATE 42710).

The unqualified form of sequence-name is an SQL identifier. The qualified form is a qualifier followed by a period and an SQL identifier. The qualifier is a schema name.

If the sequence name is explicitly qualified with a schema name, the schema name cannot begin with 'SYS' or an error (SQLSTATE 42939) is raised.

AS *data-type*
Specifies the data type to be used for the sequence value. The data type can be any exact numeric type (SMALLINT, INTEGER, BIGINT or DECIMAL) with a scale of zero, or a user-defined distinct type or reference type for which the source type is an exact numeric type with a scale of zero (SQLSTATE 42815). The specified distinct type cannot have any data type constraints and the source type cannot be an anchored data type (SQLSTATE 428H2). The default is INTEGER.

START WITH *numeric-constant*
Specifies the first value for the sequence. This value can be any positive or negative value that could be assigned to a column of the data type associated with the sequence (SQLSTATE 42815), without nonzero digits existing to the right of the decimal point (SQLSTATE 428FA). The default is MINVALUE for ascending sequences and MAXVALUE for descending sequences.

This value is not necessarily the value that a sequence would cycle to after reaching the maximum or minimum value of the sequence. The START WITH clause can be used to start a sequence outside the range that is used for cycles. The range used for cycles is defined by MINVALUE and MAXVALUE.

INCREMENT BY *numeric-constant*
Specifies the interval between consecutive values of the sequence. This value can be any positive or negative value that could be assigned to a column of the data type associated with the sequence (SQLSTATE 42815). The value must not exceed the value of a large integer constant (SQLSTATE 42820) and must not contain nonzero digits to the right of the decimal point (SQLSTATE 428FA).

If this value is negative, this is a descending sequence. If this value is 0 or positive, this is an ascending sequence. The default is 1.

MINVALUE or NO MINVALUE
Specifies the minimum value at which a descending sequence either cycles or stops generating values, or an ascending sequence cycles to after reaching the maximum value.

MINVALUE *numeric-constant*
Specifies the numeric constant that is the minimum value. This value can be any positive or negative value that could be assigned to a column of the data type associated with the sequence (SQLSTATE 42815), without nonzero digits existing to the right of the decimal point (SQLSTATE 428FA), but the value must be less than or equal to the maximum value (SQLSTATE 42815).

NO MINVALUE
For an ascending sequence, the value is the START WITH value, or 1 if START WITH is not specified. For a descending sequence, the value is the minimum value of the data type associated with the sequence. This is the default.

MAXVALUE or NO MAXVALUE
Specifies the maximum value at which an ascending sequence either cycles or stops generating values, or a descending sequence cycles to after reaching the minimum value.

MAXVALUE *numeric-constant*
Specifies the numeric constant that is the maximum value. This value can be any positive or negative value that could be assigned to a column of the data type associated with the sequence (SQLSTATE 42815), without nonzero digits existing to the right of the decimal point (SQLSTATE 428FA), but the value must be greater than or equal to the minimum value (SQLSTATE 42815).

NO MAXVALUE
For an ascending sequence, the value is the maximum value of the data type associated with the sequence. For a descending sequence, the value is the START WITH value, or -1 if START WITH is not specified.

CYCLE or NO CYCLE
Specifies whether the sequence should continue to generate values after reaching either its maximum or minimum value. The boundary of the sequence can be reached either with the next value landing exactly on the boundary condition, or by overshooting it.

CYCLE
Specifies that values continue to be generated for this sequence after the maximum or minimum value has been reached. If this option is used, after an ascending sequence reaches its maximum value it generates its minimum value; after a descending sequence reaches its minimum value it generates its maximum value. The maximum and minimum values for the sequence determine the range that is used for cycling.

When CYCLE is in effect, then duplicate values can be generated for the sequence.

NO CYCLE
Specifies that values will not be generated for the sequence once the maximum or minimum value for the sequence has been reached. This is the default.

CACHE or NO CACHE
Specifies whether to keep some preallocated values in memory for faster access. This is a performance and tuning option.

CACHE *integer-constant*
Specifies the maximum number of sequence values that are preallocated and kept in memory. Preallocating and storing values in the cache reduces synchronous I/O to the log when values are generated for the sequence.

In the event of a system failure, all cached sequence values that have not been used in committed statements are lost (that is, they will never be used). The value specified for the CACHE option is the maximum number of sequence values that could be lost in case of system failure.

The minimum value is 2 (SQLSTATE 42815). The default value is CACHE 20.

Use the CACHE and NO ORDER options to allow multiple caches of sequence values simultaneously. In a multi-partition or Db2 pureScale environment, multiple members can cache them.

In a Db2 pureScale environment, if both CACHE and ORDER are specified, the specification of ORDER overrides the specification of CACHE and instead NO CACHE will be in effect.

NO CACHE
Specifies that values of the sequence are not to be preallocated. It ensures that there is not a loss of values in the case of a system failure, shutdown or database deactivation. When this option is specified, the values of the sequence are not stored in the cache. In this case, every request for a new value for the sequence results in synchronous I/O to the log.

NO ORDER or ORDER
Specifies whether the sequence numbers must be generated in order of request.

ORDER
Specifies that the sequence numbers are generated in order of request.

NO ORDER
Specifies that the sequence numbers do not need to be generated in order of request. This is the default.

Notes

- It is possible to define a constant sequence, that is, one that would always return a constant value. This could be done by specifying an INCREMENT value of zero and a START WITH value that does not exceed MAXVALUE, or by specifying the same value for START WITH, MINVALUE and MAXVALUE. For a constant sequence, each time NEXT VALUE is invoked for the sequence, the same value is returned. A constant sequence can be used as a numeric global variable. ALTER SEQUENCE can be used to adjust the values that will be generated for a constant sequence.
- A sequence can be cycled manually by using the ALTER SEQUENCE statement. If NO CYCLE is implicitly or explicitly specified, the sequence can be restarted or extended using the ALTER SEQUENCE statement to cause values to continue to be generated once the maximum or minimum value for the sequence has been reached.
- A sequence can be explicitly defined to cycle by specifying the CYCLE keyword. Use the CYCLE option when defining a sequence to indicate that the generated values should cycle once the boundary is reached. When a sequence is defined to automatically cycle (that is, CYCLE was explicitly specified), the maximum or minimum value generated for a sequence might not be the actual MAXVALUE or MINVALUE specified, if the increment is a value other than 1 or -1. For example, the sequence defined with START `WITH=1, INCREMENT=2, MAXVALUE=10` will generate a maximum value of 9, and will not generate the value 10. When defining a sequence with CYCLE, carefully consider the impact of the values for MINVALUE, MAXVALUE and START WITH.
- Caching sequence numbers implies that a range of sequence numbers can be kept in memory for fast access. When an application accesses a sequence that can allocate the next sequence number from the cache, the sequence number allocation can happen quickly. However, if an application accesses a sequence that cannot allocate the next sequence number from the cache, the sequence number allocation may require having to wait for I/O operations to persistent storage. The choice of the value for CACHE should be done keeping in mind the performance and application requirements tradeoffs.
- ***Gaps in a sequence:*** Consecutive values in a sequence differ by the constant INCREMENT BY value specified for the sequence. However, gaps can occur in the values that are assigned to a sequence object by Db2.

  The following situations are some examples of how gaps can be introduced in the sequence values:

  – A transaction has advanced the sequence and then rolls back.
  – The SQL statement leading to the generation of the next value fails after the value was generated.
  – The NEXT VALUE expression is used in the SELECT statement of a cursor in a DRDA environment where the client uses block-fetch and not all retrieved rows are fetched by the application.
  – The sequence is altered and then the alteration is rolled back.
  – The sequence (or an identity column table) is dropped and then the drop is rolled back.
  – The SYSIBM.SYSSEQ table space is stopped or closed for any reason (including when DSMAX is reached).
  – The Db2 subsystem is stopped or goes down.

  Values of such gaps are not available for the current cycle, unless the sequence is altered and restarted in a specific way to make them available.

  A sequence is incremented independently of a transaction. Thus, a given transaction increments the sequence two times might see a gap in the two numbers that it receives if other transactions concurrently increment the same sequence. Most applications can tolerate these instances as these are not really gaps.
- The definer of a sequences is granted ALTER and USAGE privileges with the grant option. The owner of the sequence can drop the sequence.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.

  – A comma can be used to separate multiple sequence options

  - NOMINVALUE, NOMAXVALUE, NOCYCLE, NOCACHE, and NOORDER can be specified in place of NO MINVALUE, NO MAXVALUE, NO CYCLE, NO CACHE, and NO ORDER, respectively
- ***Considerations for a multi-partition or Db2 pureScale environment***:
  - If the CACHE and NO ORDER options are in effect, multiple caches can be active simultaneously. This can happen at each member in a multi-partition or Db2 pureScale environment. The requests for next value assignments from different members might not result in the assignment of values in strict numeric order. Assume, for example, in a multi-partition or Db2 pureScale environment, that members DB1A and DB1B are using the same sequence, and DB1A gets the cache values 1 to 20 and DB1B gets the cache values 21 to 40. In this scenario, if DB1A requested the next value first, then DB1B requested, and then DB1A requested again, the actual order of values assigned would be 1,21,2. Therefore, to guarantee that sequence numbers are generated in strict numeric order among multiple members using the same sequence concurrently, specify the ORDER option.
  - In a Db2 pureScale environment, using the ORDER or NO CACHE option ensures that the values assigned to a sequence which is shared by applications across multiple members are in strict numeric order. If ORDER is specified, then NO CACHE is implied even if CACHE *n* is specified.

Example

Create a sequence called ORG_SEQ that starts at 1, increments by 1, does not cycle, and caches 24 values at a time:

```
CREATE SEQUENCE ORG_SEQ
  START WITH 1
  INCREMENT BY 1
  NO MAXVALUE
  NO CYCLE
  CACHE 24
```

CREATE SERVICE CLASS

The CREATE SERVICE CLASS statement defines a service class.

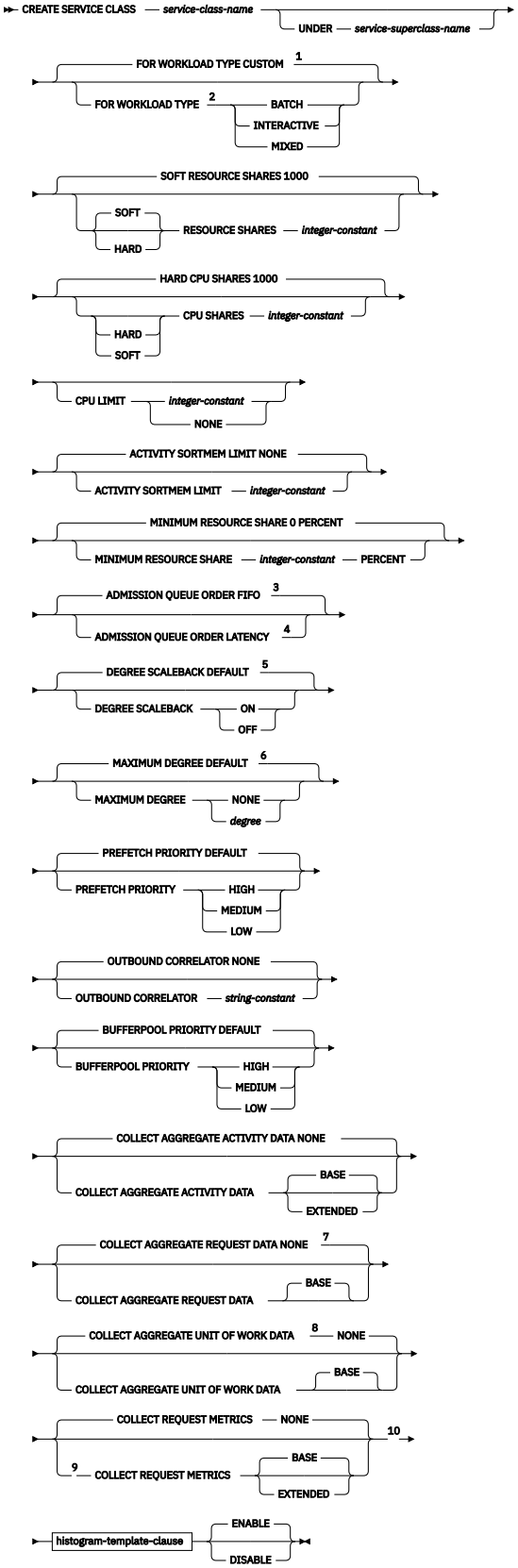
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

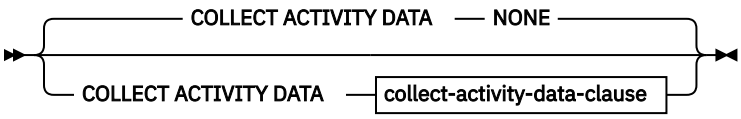
Authorization

The privileges held by the authorization ID of the statement must include WLMADM or DBADM authority.

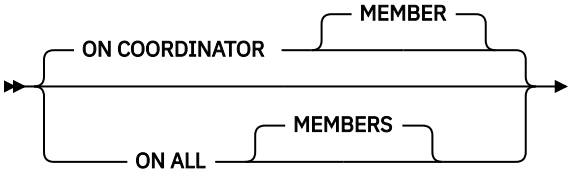
Syntax

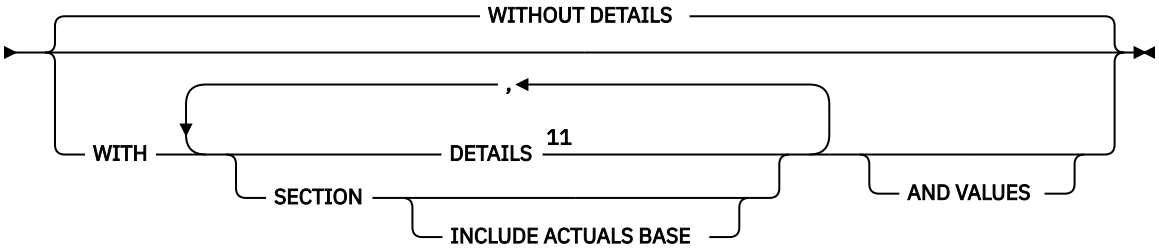


collect-activity-clause

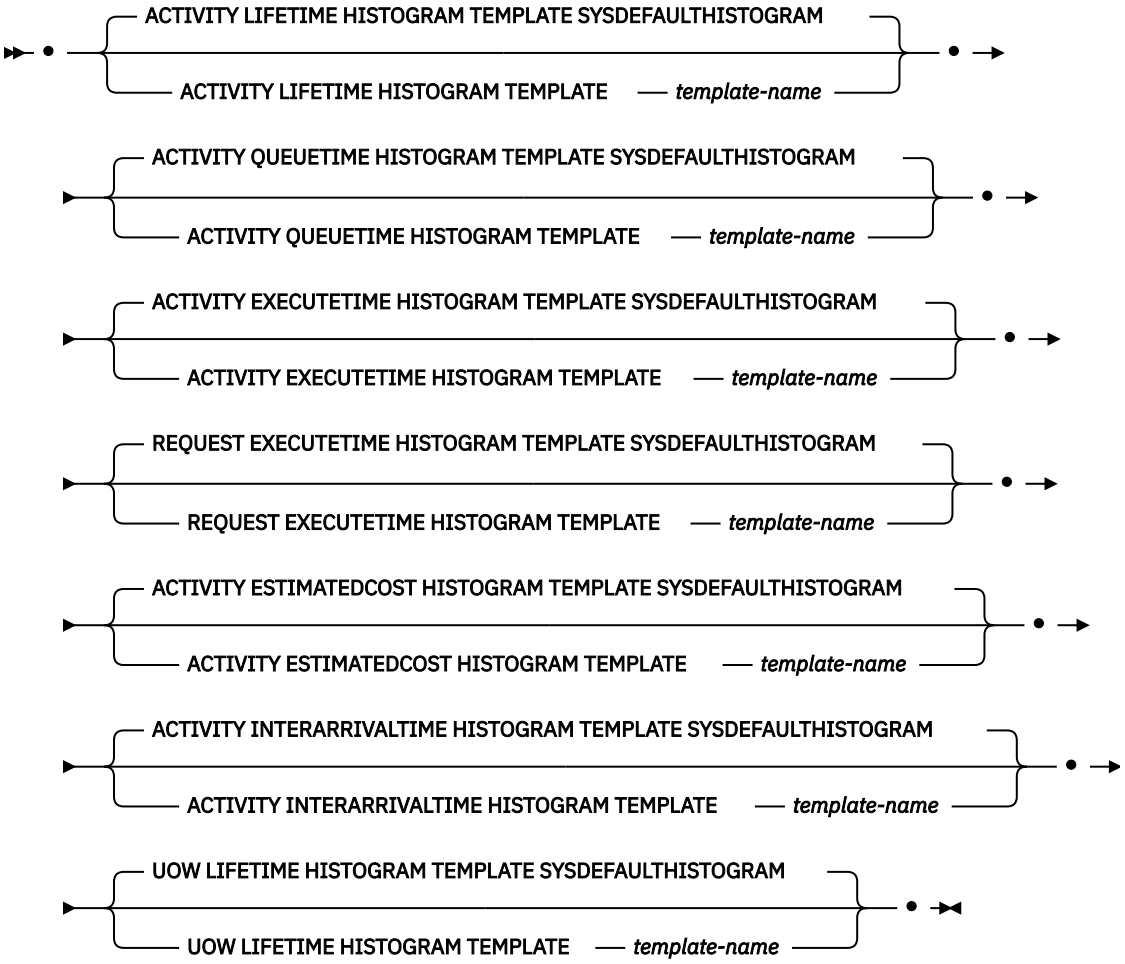


collect-activity-data-clause





histogram-template-clause



Notes:

[1] The FOR WORKLOAD TYPE clause is valid only for a service superclass (SQLSTATE 5U044).
[2] The FOR WORKLOAD TYPE clause is valid only for a service superclass (SQLSTATE 5U044).
[3] The ADMISSION QUEUE ORDER clause is valid only for a service subclass (SQLSTATE 5U043).

[4] The ADMISSION QUEUE ORDER clause is valid only for a service subclass (SQLSTATE 5U043).
[5] The DEGREE SCALEBACK DEFAULT option is valid only for a service subclass (SQLSTATE 5U043).
[6] The MAXIMUM DEGREE DEFAULT option is valid only for a service subclass (SQLSTATE 5U043).
[7] The COLLECT AGGREGATE REQUEST DATA clause is valid only for service subclasses.
[8] The COLLECT AGGREGATE UNIT OF WORK DATA clause is valid only for service subclasses.
[9] The COLLECT REQUEST METRICS clause is valid only for a service superclass.
[10] The REQUEST EXECUTETIME AND UOW LIFETIME HISTOGRAM TEMPLATE clauses are valid only for a service subclass.
[11] The DETAILS keyword is the minimum to be specified, followed by the option separated by a comma.

Description

service-class-name
Name of the service class to be created. This is a one-part name that is also an SQL identifier (either ordinary or delimited). The following restrictions apply:

- If the service class is a service superclass, the service class name must not identify a service superclass that already exists in the catalog (SQLSTATE 42710).
- If the service class is a service subclass, the service class name must not be the same as its service superclass, and must not identify a service subclass that already exists under the service superclass (SQLSTATE 42710).
- The name must not begin with the characters 'SYS' (SQLSTATE 42939).

UNDER *service-superclass-name*
Specifies that the service class is a subclass of service superclass *service-superclass-name*. If UNDER is not specified, the service class is a service superclass. The *service-superclass-name* must identify a service superclass that exists for the database (SQLSTATE 42704). The service superclass cannot be a default service class (SQLSTATE 5U029).

FOR WORKLOAD TYPE
Specifies the type of workload that is expected to run in the service superclass. This dictates how the service superclass is configured.

CUSTOM
The service superclass attributes are set to their default values. This is the default.

BATCH
The service superclass is configured to optimize it for large, batch-oriented activities.

INTERACTIVE
The service superclass is configured to optimize its response time for short activities.

MIXED
The service superclass is configured to handle a mixed set of activities of varying types and complexity.

RESOURCE SHARES
Specifies the number of shares of resources to which this service class is entitled, and whether the service class is allowed to exceed this number when other service classes in the same scope are not using their full entitlements. This value affects the amount of work the workload manager (WLM) adaptive admission control allows into the system.

HARD
The service class is not allowed to exceed its resource share entitlement.

SOFT
The service class is allowed to exceed its resource share entitlement when other service classes are not using their full entitlements.

Valid values are integers `1` - `65535`. The default is `SOFT RESOURCE SHARES 1000`.

Note: To use resource shares with WLM, you must enable the **wlm_admission_ctrl** configuration parameter.

CPU SHARES
Specifies the number of CPU shares that the workload manager (WLM) dispatcher allocates to this service class when work is executing within this service class, and whether the service class is allowed to exceed this number when other service classes in the same scope are not using their full entitlement.

HARD
The service class is not allowed to exceed its CPU share entitlement.

SOFT
The service class is allowed to exceed its CPU share entitlement when other service classes are not using their full entitlements.

Valid values are integers 1 - 65535. The default is `HARD CPU SHARES 1000`.

Note: To use CPU shares with WLM dispatcher, you must enable the **wlm_disp_cpu_shares** database manager configuration parameter.

CPU LIMIT
Specifies the maximum percentage of the CPU resources that the WLM dispatcher can assign to this service class. Valid values for the *integer-constant* are integers between 1 and 100. You can also specify **CPU LIMIT NONE** to indicate that there is no CPU limit.

ACTIVITY SORTMEM LIMIT

Specifies the maximum percentage of the configured shared sort memory (**SHEAPTHRES_SHR**) that individual queries executing in the service class are allowed to consume. Queries requiring more memory than the configured limit will have their individual per-operator **SORTHEAP** values reduced at runtime and memory requests will be throttled if they exceed the limit. Valid values for the integer-constant are integers between 10 and 100. You can also specify NONE to indicate there is no activity sort memory limit. The default is NONE.

The effective sort memory limit for a query will be the most restrictive of the limit defined at the subclass, superclass and database via the **ACT_SORTMEM_LIMIT** database configuration parameter. The sort memory limit applied to an activity is determined when the activity is first admitted for execution. The applied sort memory limit will not change if a query is remapped at runtime to a different service subclass.

The activity sort memory limit will only be enforced for queries that are managed by the adaptive workload manager. If the adaptive workload manager is disabled (**WLM_ADMISSION_CTRL** database config parameter is set to NO) or a query bypasses the adaptive workload manager, no sort memory limit will be applied to the query regardless of the service class it runs in.

Note: Setting an activity sort memory limit too low may result in reduced performance for queries.

MINIMUM RESOURCE SHARE *integer-constant* PERCENT
Specifies the percentage of entitled resources used by WLM adaptive admission control that is to be held in reserve for the service class when other service classes exceed their admission resource entitlement. Valid values for *integer-constant* are integers 0 - 100. The default is 0.

ADMISSION QUEUE ORDER
Specifies the queue order for activities queued by WLM adaptive admission control.

FIFO
Requests are queued in a first-in first-out order. This is the default.

LATENCY
The position of a request in the queue is based on its estimated execution time (that is, its latency) relative to the amount of time that has elapsed since it joined the queue.

DEGREE SCALEBACK
Specifies whether work running in this service class may have its degree scaled back. Queries set to DEGREE ANY may have their actual runtime degree scaled back by the database manager based on current CPU loads.

Scaling back the degree for service classes running simple queries may result in less contention and improved throughput. Disabling degree scale back for service classes with complex queries can help ensure more consistent and predictable response times. A setting of DEFAULT means a service subclass inherits its DEGREE SCALEBACK setting from the parent superclass. The DEFAULT setting is only applicable to service subclasses. The default setting for a service superclass is ON. The default value for a service subclass is DEFAULT.

MAXIMUM DEGREE
Specifies the maximum runtime degree of parallelism for activities that are running in this service class.

DEFAULT
This service subclass inherits its maximum degree value from its parent superclass. This value is the default for a service subclass. This setting is applicable only to service subclasses.

NONE
This service class does not specify a maximum runtime degree for assigned applications. The actual runtime degree is determined as the lower of the value of max_querydegree configuration parameter, the value set by SET RUNTIME DEGREE command, the SQL statement compilation degree and the MAXIMUM DEGREE value set on the Workload. This is the default for a service superclass.

degree
Specifies the maximum degree of parallelism for this service class. Valid values are 1 to 32767. The actual runtime degree is determined as the lower of this degree, the value of max_querydegree configuration parameter, the value set by SET RUNTIME DEGREE command, the SQL statement compilation degree and the MAXIMUM DEGREE set on the Workload.

PREFETCH PRIORITY
This parameter controls the priority with which agents in the service class can submit their prefetch requests. Valid values are HIGH, MEDIUM, LOW, or DEFAULT (SQLSTATE 42615). HIGH, MEDIUM, and LOW mean that prefetch requests will be submitted to the high, medium, and low priority queues. Prefetchers empty the priority queue in order from high to low. Agents in the service class submit their prefetch requests at the prefetch priority level when the next activity begins. If the prefetch priority is altered after a prefetch request is submitted, the request priority does not change. The default value is DEFAULT, which is internally mapped to MEDIUM for service superclasses. If DEFAULT is specified for a service subclass, it inherits the prefetch priority of its parent superclass.

The prefetch priority cannot be altered for a default subclass (SQLSTATE 5U032).

OUTBOUND CORRELATOR
Specifies whether or not to associate threads from this service class to an external workload manager service class.

If OUTBOUND CORRELATOR is set to a *string-constant* for the service superclass and OUTBOUND CORRELATOR NONE is set for a service subclass, the service subclass inherits the OUTBOUND CORRELATOR of its parent.

OUTBOUND CORRELATOR NONE
For a service superclass, specifies that there is no external workload manager service class association with this service class, and for a service subclass, specifies that the external workload manager service class association is the same as its parent. This is the default.

OUTBOUND CORRELATOR *string-constant*
Specifies the *string-constant* that is to be used as a correlator to associate threads from this service class to an external workload manager service class. The external workload manager must be active (SQLSTATE 5U030). The external workload manager should be set up to recognize the value of the specified string constant.

BUFFERPOOL PRIORITY
This parameter controls the bufferpool priority of pages fetched by activities in this service class. Valid values are HIGH, MEDIUM, LOW or DEFAULT (SQLSTATE 42615). Pages fetched by activities in a service class with higher bufferpool priority are less likely to be swapped out than pages fetched by activities in a service class with lower bufferpool priority. The default value is DEFAULT, which is internally mapped to LOW for service superclasses. If DEFAULT is specified for a service subclass, it inherits the bufferpool priority from its parent superclass.

The bufferpool priority cannot be altered for a default subclass (SQLSTATE 5U032).

COLLECT ACTIVITY DATA
Specifies that information about each activity that executes in this service class is to be sent to any active activities event monitor when the activity completes.

NONE
Specifies that activity data should not be collected for each activity that executes in this service class. This is the default.

ON COORDINATOR MEMBER
Specifies that activity data is to be collected only at the coordinator member of the activity.

ON ALL MEMBERS
Specifies that activity data is to be collected at all members where the activity is processed. On remote members, a record for the activity may be captured multiple times as the activity comes and goes on those members. If the AND VALUES clause is specified, activity input values will be collected only for the members of the coordinator.

WITHOUT DETAILS
Specifies that data about each activity that executes in the service class is to be sent to any active activities event monitor, when the activity completes execution. Details about statement, compilation environment, and section environment data are not sent.

WITH DETAILS
Specifies that statement and compilation environment data is to be sent to any active activities event monitor, for those activities that have them. Section environment data is not sent.

SECTION
Specifies that statement, compilation environment, section environment data, and section actuals are to be sent to any active activities event monitor for those activities that have them. DETAILS must be specified if SECTION is specified. Section actuals will be collected on any member where the activity data is collected.

INCLUDE ACTUALS BASE
Specifies that section actuals should also be collected on any partition where the activity data is collected. For section actuals to be collected, either INCLUDE ACTUALS clause must be specified or the **section_actuals** database configuration parameter must be set.

The effective setting for the collection of section actuals is the combination of the INCLUDE ACTUALS clause, the **section_actuals** database configuration parameter, and the <collectsectionactuals> setting specified on the WLM_SET_CONN_ENV routine. For example, if INCLUDE ACTUALS BASE is specified, yet the **section_actuals** database configuration parameter value is NONE and <collectsectionactuals> is set to NONE, then the effective setting for the collection of section actuals is BASE.

BASE specifies that the following should be enabled and collected during the activity's execution:

- Basic operator cardinality counts
- Statistics for each object referenced (DML statements only)

AND VALUES
Specifies that input data values are to be sent to any active activities event monitor, for those activities that have them. This data does not include SQL statements that are compiled by using the REOPT ALWAYS bind option.

COLLECT AGGREGATE ACTIVITY DATA
Specifies that aggregate activity data should be captured for this service class and sent to the statistics event monitor, if one is active. This information is collected periodically on an interval that is specified by the **wlm_collect_int** database configuration parameter.

BASE
Specifies that basic aggregate activity data should be captured for this service class and sent to the statistics event monitor, if one is active. Basic aggregate activity data includes:

- Estimated activity cost high watermark
- Rows returned high watermark
- Temporary table space usage high watermark

  Note: Only activities that have an SQLTEMPSPACE threshold applied to them participate in this high watermark.
- Activity life time histogram
- Activity queue time histogram
- Activity execution time histogram

This is the default when COLLECT AGGREGATE ACTIVITY DATA is specified, but without a value.

EXTENDED
Specifies that all aggregate activity data should be captured for this service class and sent to the statistics event monitor, if one is active. This includes all basic aggregate activity data plus:

- Activity data manipulation language (DML) estimated cost histogram
- Activity DML inter-arrival time histogram

NONE
Specifies that no aggregate activity data should be captured for this service class. This is the default when COLLECT AGGREGATE ACTIVITY DATA is not specified.

COLLECT AGGREGATE REQUEST DATA
Specifies that aggregate request data should be captured for this service class and sent to the statistics event monitor, if one is active. This information is collected periodically on an interval specified by the **wlm_collect_int** database configuration parameter. The COLLECT AGGREGATE REQUEST DATA clause is valid only for a service subclass.

BASE
Specifies that basic aggregate request data should be captured for this service class and sent to the statistics event monitor, if one is active.

NONE
Specifies that no aggregate request data should be captured for this service class. This is the default.

COLLECT AGGREGATE UNIT OF WORK DATA
Specifies that aggregate unit of work data is to be captured for this service class and sent to the statistics event monitor, if one is active. This information is collected periodically on an interval specified by the **wlm_collect_int** database configuration parameter. The COLLECT AGGREGATE UNIT OF WORK DATA clause is valid only for a service subclass.

BASE
Specifies that basic aggregate unit of work data is to be captured for this service class and sent to the statistics event monitor, if one is active. Basic aggregate unit of work includes:

- Unit of work lifetime histogram

NONE
Specifies that no aggregate unit of work data is to be collected for this service class. This is the default.

COLLECT REQUEST METRICS
Specifies that monitor metrics should be collected for any request submitted by a connection that is associated with the specified service superclass and sent to the statistics and unit of work event monitors, if active. The COLLECT REQUEST METRICS clause is valid only for a service superclass (SQLSTATE 50U44).

Note: The effective request metrics collection setting is the combination of the attribute specified by the COLLECT REQUEST METRICS clause on the service superclass associated with the connection submitting the request, and the **mon_req_metrics** database configuration parameter. If either the service superclass attribute or the configuration parameter has a value other than NONE, metrics will be collected for the request.

NONE
Specifies that no metrics will be collected for any request submitted by a connection associated with the service superclass. This is the default.

BASE
Specifies that basic metrics will be collected for any request submitted by a connection associated with the service superclass.

EXTENDED
Specifies that basic aggregate request data should be captured for this service class and sent to the statistics event monitor, if one is active. In addition, specifies that the values for the following monitor elements should be determined with additional granularity:

- **total_section_time**
- **total_section_proc_time**
- **total_routine_user_code_time**
- **total_routine_user_code_proc_time**
- **total_routine_time**

histogram-template-clause
Specifies the histogram templates to use when collecting aggregate activity data for activities executing in the service class.

ACTIVITY LIFETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the duration, in milliseconds, of database activities running in the service class during a specific interval. This time includes both time queued and time executing. The default is SYSDEFAULTHISTOGRAM. This information is collected only when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

ACTIVITY QUEUETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, that database activities running in the service class are queued during a specific interval. The default is SYSDEFAULTHISTOGRAM. This information is collected only when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

ACTIVITY EXECUTETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, that database activities running in the service class are executing during a specific interval. This time does not include the time spent queued. Activity execution time is collected in this histogram at the coordinator member only. The time does not include idle time. Idle time is the time between the execution of requests belonging to the same activity when no work is being done. An example of idle time is the time between the end of opening a cursor and the start of fetching from that cursor. The default is SYSDEFAULTHISTOGRAM. This information is collected only when the COLLECT AGGREGATE ACTIVITY DATA clause is specified,

with either the BASE or EXTENDED option. Only activities at nesting level 0 are considered for inclusion in the histogram.

REQUEST EXECUTETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, that database requests running in the service class are executing during a specific interval. This time does not include the time spent queued. Request execution time is collected in this histogram on each member where the request executes. The default is SYSDEFAULTHISTOGRAM. This information is collected only when the COLLECT AGGREGATE REQUEST DATA clause is specified with the BASE option.

ACTIVITY ESTIMATEDCOST HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the estimated cost, in timerons, of DML activities running in the service class. The default is SYSDEFAULTHISTOGRAM. This information is collected only when the COLLECT AGGREGATE ACTIVITY DATA clause is specified with the EXTENDED option. Only activities at nesting level 0 are considered for inclusion in the histogram.

ACTIVITY INTERARRIVALTIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, between the arrival of one DML activity and the arrival of the next DML activity. The default is SYSDEFAULTHISTOGRAM. This information is collected only when the COLLECT AGGREGATE ACTIVITY DATA clause is specified with the EXTENDED option.

UOW LIFETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the duration, in milliseconds, of units of work running in the service class during a specific interval. The default is SYSDEFAULTHISTOGRAM. This information is collected only when the COLLECT AGGREGATE UNIT OF WORK DATA clause is specified with the BASE option.

ENABLE or DISABLE
Specifies whether or not connections and activities can be mapped to the service class.

ENABLE
Connections and activities can be mapped to the service class. This is the default.

DISABLE
Connections and activities cannot be mapped to the service class. New connections or activities that are mapped to a disabled service class will be rejected (SQLSTATE 5U028). When a service superclass is disabled, its service subclasses are also disabled. When the service superclass is re-enabled, its service subclasses return to states that are defined in the system catalog. A default service class cannot be disabled (SQLSTATE 5U032).

Rules

- The maximum number of service subclasses that can be created under a service superclass is 61 (SQLSTATE 5U027).
- The maximum number of service superclasses that can be created for a database is 64 (SQLSTATE 5U027).
- A workload management (WLM)-exclusive SQL statement must be followed by a COMMIT or a ROLLBACK statement (SQLSTATE 5U027). WLM-exclusive SQL statements are:
  - CREATE HISTOGRAM TEMPLATE, ALTER HISTOGRAM TEMPLATE, or DROP (histogram template)
  - CREATE SERVICE CLASS, ALTER SERVICE CLASS, or DROP (service class)
  - CREATE THRESHOLD, ALTER THRESHOLD, or DROP (threshold)
  - CREATE WORK ACTION SET, ALTER WORK ACTION SET, or DROP (work action set)
  - CREATE WORK CLASS SET, ALTER WORK CLASS SET, or DROP (work class set)
  - CREATE WORKLOAD, ALTER WORKLOAD, or DROP (workload)
  - GRANT (workload privileges) or REVOKE (workload privileges)

- A WLM-exclusive SQL statement cannot be issued within a global transaction (SQLSTATE 51041) such as, for example, an XA transaction.

Notes

- A default subclass, SYSDEFAULTSUBCLASS, is automatically created for every service superclass.
- Only one uncommitted WLM-exclusive SQL statement at a time is allowed across all members. If an uncommitted WLM-exclusive SQL statement is executing, subsequent WLM-exclusive SQL statements will wait until the current WLM-exclusive SQL statement commits or rolls back.
- Changes are written to the system catalog, but do not take effect until after a COMMIT statement, even for the connection that issues the statement.
- ***Syntax alternatives***: The following are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - DATABASE PARTITION can be specified in place of MEMBER, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
  - DATABASE PARTITIONS can be specified in place of MEMBERS, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.

Examples

- *Example 1:* Create a service superclass named PETSALES. The default subclass for PETSALES is automatically created.

```
CREATE SERVICE CLASS PETSALES
```

- *Example 2:* Create a service subclass named DOGSALES under service superclass PETSALES. Set service class DOGSALES as disabled.

```
CREATE SERVICE CLASS DOGSALES UNDER PETSALES DISABLE
```

- *Example 3:* Create a service superclass named BARNSALES with a prefetcher priority of LOW. The default subclass for BARNSALES is automatically created. Prefetch requests submitted by agents in the BARNSALES service class will go to the low priority prefetch queue.

```
CREATE SERVICE CLASS BARNSALES PREFETCH PRIORITY LOW
```

CREATE SERVER

The CREATE SERVER statement defines a data source to a federated database.

In this statement, the term SERVER and the parameter names that start with *server-* refer only to data sources in a federated system. They do not refer to the federated server in such a system, or to DRDA application servers.

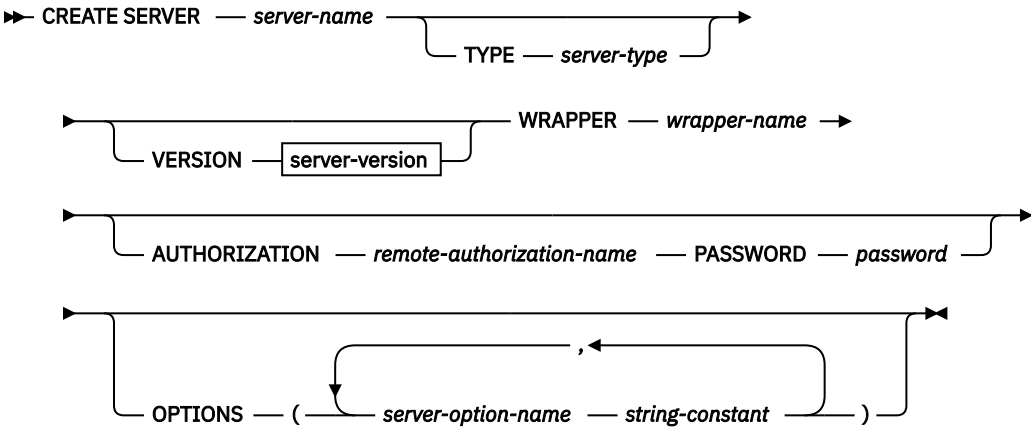
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

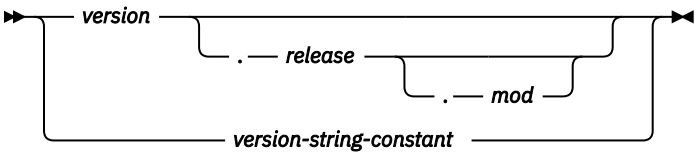
Authorization

The privileges held by the authorization ID of the statement must include DBADM authority.

Syntax



server-version



Description

server-name

Specifies the name of the data source that is being defined to the federated database. The name must not identify a data source that is described in the catalog. The server name must not be the same as the name of any of the table spaces in the federated database.

A server definition for relational data sources usually represents a remote database. Some relational database management systems, such as Oracle, do not allow multiple databases within each instance. Instead, each instance represents a server within a federated system.

For non-relational data sources, the purpose of a server definition varies from data source to data source. Some server definitions map to a search type and daemon, a website, or a web server. For other non-relational data sources, a server definition is created because the hierarchy of federated objects requires that data source files (identified by nicknames) are associated with a specific server object.

TYPE *server-type*

Specifies the type of the data source that is being defined to the federated database, and determines the default wrapper that is used.

Table 136. Server types and default wrappers

Data source	Type	Default Wrapper
Amazon Redshift	REDSHIFT	ODBC
Apache Hive	HIVE	ODBC
Apache Spark	SPARK	JDBC
Apache Spark SQL	SPARK_ODBC	ODBC
Cloudera Impala	IMPALA	ODBC
CouchDB	COUCHDB	NoSQL
database.com	DATABASE.COM	ODBC

Table 136. Server types and default wrappers (continued)		
Data source	**Type**	**Default Wrapper**
force.com	FORCE.COM	ODBC
HDFS parquet	HDFSPARQUET	NoSQL
IBM BigInsights	BIGSQL	DRDA
IBM Db2 Warehouse on Cloud	DASHDB	DRDA
IBM Db2 on Cloud	DASHDB	DRDA
IBM Db2 Warehouse	DASHDB	DRDA
IBM Db2	DB2/LUW	DRDA
Db2 for z/OS	DB2/ZOS	DRDA
Db2 for IBM i	DB2/ISERIES	DRDA
IBM Db2 Hosted	DB2/LUW	DRDA
IBM Db2 Server for VSE and VM	DB2/VM	DRDA
IBM PureData System for Analytics (formerly Netezza)	PDA (the type NETEZZA can also be used but is deprecated)	ODBC
IBM PureData System for Operational Analytics	DB2/LUW	DRDA
IBM PureData System for Transactions	DB2/LUW	DRDA
Informix® (with INFORMIX wrapper)	INFORMIX	INFORMIX
Informix (with ODBC wrapper)	INFORMIX_ODBC	ODBC
JDBC	JDBC	JDBC
MariaDB	MARIADB	ODBC
Microsoft Azure	AZURE	ODBC
Microsoft SQL Server (with MSSQLODBC3 wrapper)	MSSQLSERVER	MSSQLODBC3
Microsoft SQL Server (with ODBC wrapper)	MSSQL_ODBC	ODBC
MongoDB	MONGODBREST[1], MONGODRIVER[2], RESTHEART[3]	NoSQL
ODBC	ODBC	ODBC
Oracle (with NET8 wrapper)	ORACLE	NET8
Oracle (with ODBC wrapper)	ORACLE_ODBC	ODBC
Oracle Cloud	ORACLE_CLOUD	ODBC
Oracle MySQL	MYSQL	ODBC
Pivotal Greenplum	GREENPLUM	ODBC
Pivotal HAWQ	HAWQ	ODBC
PostgreSQL	POSTGRESQL	ODBC

Table 136. Server types and default wrappers (continued)		
Data source	**Type**	**Default Wrapper**
Progress OpenEdge	OPENEDGE	ODBC
Salesforce	SALESFORCE	ODBC
SAP HANA	HANA	ODBC
SAP Sybase	SYBASE	CTLIB
SAP Sybase IQ	SYBASEIQ	ODBC
SAP Sybase ASE	SYBASE_ODBC	ODBC
Teradata (with TERADATA wrapper)	TERADATA	TERADATA
Teradata (with ODBC wrapper)	TERADATA_ODBC	ODBC

This parameter is required by some wrappers. For example, it is required by an ODBC wrapper that is operating in DSN-less connection mode.

[1] Server type MONGODBREST uses RESTAPI to connect to MongoDB. The MONGODBREST option depends on an HTTP interface being configured correctly and running on MongoDB version 3.4, and earlier versions have a built-in HTTP interface.

[2] Server type MONGODRIVER uses the native MongoDB driver to connect to MongoDB. This approach has less dependency compared to other approaches and provides more performance advantages because this method leverages MongoDB full native API.

[3] Server type RESTHEART uses RESTAPI to get data. However, RestHeart is a third-party tool that must be installed separately. RestHeart supports more features than the MongoDB HTTP interface, and provides better performance than using MONGODBREST server type because it supports more filter functionality.

VERSION
Specifies the version of the data source denoted by *server-name*. This parameter is required by some wrappers.

version
Specifies the version number. The value must be an integer.

release
Specifies the number of the release of the version denoted by *version*. The value must be an integer.

mod
Specifies the number of the modification of the release denoted by *release*. The value must be an integer.

version-string-constant
Specifies the complete designation of the version. The *version-string-constant* can be a single value (for example, '8i'); or it can be the concatenated values of *version*, *release* and, if applicable, *mod* (for example, '8.0.3').

WRAPPER *wrapper-name*
Specifies the name of the wrapper that the federated server is to use to interact with the newly created server object. The default depends on the type of the remote server (see Table 136 on page 1344.

If the default wrapper does not already exist, it is created with the SET DB2_FENCED 'Y' option. A default wrapper is created only once. For example, if a wrapper with the name DRDA does not already exist, and if you issue two CREATE SERVER statements (one to create a BIGSQL server and another to create a DB2/ZOS server), and if you do not specify a wrapper name for either server explicitly, the first statement causes a default wrapper with the name DRDA to be created and both new servers use that wrapper.

Note: If a wrapper with the same name as a not-yet-created default wrapper already exists (for example, because it was created by a CREATE WRAPPER statement), that wrapper is not used as a default for a CREATE SERVER statement. In this situation, no default is available and the CREATE SERVER statement must specify a wrapper name explicitly.

AUTHORIZATION ***remote-authorization-name***
Required only for Db2 family data sources. Specifies the authorization ID under which any necessary actions are performed at the data source when the CREATE SERVER statement is processed. This authorization ID is not used when establishing subsequent connections to the server.

This ID must hold the authority (BINDADD or its equivalent) that the necessary actions require. If the *remote-authorization-name* is specified in mixed or lowercase characters (and the remote data source has case sensitive authorization names), the *remote-authorization-name* should be enclosed by double quotation marks.

PASSWORD ***password***
Required only for Db2 family data sources. Specifies the password associated with the authorization ID represented by *remote-authorization-name*. If the *password* is specified in mixed or lowercase characters (and the remote data source has case sensitive passwords), the *password* should be enclosed by double quotation marks.

OPTIONS
Specify configuration options for the for the server to be created. Which options you can specify depends on the data source of the object for which a server is being created. For a list of data sources and the server options that apply to each, see Data source options. Each option value is a character string constant that must be enclosed in single quotation marks.

Notes

- The *password* should be specified when the data source requires a password. If any letters in *password* must be in lowercase, enclose *password* in quotation marks.
- If the CREATE SERVER statement is used to define a Db2 family instance as a data source, Db2 may need to bind certain packages to that instance. If binding is required, the *remote-authorization-name* in the statement must have BIND authority. The time required for the bind operation to complete is dependent on data source speed and network connection speed.
- No verification occurs to ensure that the specified server version matches the remote server version. Specifying an incorrect server version can result in SQL errors when you access nicknames that belong to the database server definition. This is most likely when you specify a server version that is later than the remote server version. In that case, when you access nicknames that belong to the server definition, the database server might send SQL that the remote server does not recognize.
- The **AUTHORIZATION** keyword and **PASSWORD** keyword in the **CREATE SERVER** statement become optional if following conditions are true:
  - Db2 family data source.
  - Native wrapper or Db2 JDBC wrapper.
  - Server option **SSO_AUTH** value set to 'Y'.
- **Syntax alternatives:** The following syntax is supported for compatibility with previous product versions:
  - ADD can be specified before *server-option-name string-constant*.
- The TYPE server-type is optional for DSN connection mode but mandatory for DSN-less connection mode.

Examples

1. Register a server definition to access a Db2 for z/OS and OS/390®, Version 7.1 data source. CRANDALL is the name assigned to the Db2 for z/OS and OS/390 server definition. DRDA is the name of the wrapper used to access this data source. In addition, specify that:

- GERALD and drowssap are the authorization ID and password under which packages are bound at CRANDALL when this statement is processed.
- The alias for the Db2 for z/OS and OS/390 database that was specified with the CATALOG DATABASE statement is CLIENTS390.
- The authorization IDs and passwords under which CRANDALL can be accessed are to be sent to CRANDALL in uppercase.
- CLIENTS390 and the federated database use the same collating sequence.

```
CREATE SERVER CRANDALL
  TYPE DB2/ZOS
  VERSION 7.1
  WRAPPER DRDA
  AUTHORIZATION "GERALD"
  PASSWORD drowssap
  OPTIONS
      (DBNAME 'CLIENTS390',
      FOLD_ID 'U',
      FOLD_PW 'U',
      COLLATING_SEQUENCE 'Y')
```

2. Register a server definition to access an Oracle 9 data source. CUSTOMERS is the name assigned to the Oracle server definition. NET8 is the name of the wrapper used to access this data source. In addition, specify that:
   - ABC is the name of the node where the Oracle database server resides.
   - The CPU for the federated server runs twice as fast as the CPU that supports CUSTOMERS.
   - The I/O devices at the federated server process data one and a half times as fast as the I/O devices at CUSTOMERS.

```
CREATE SERVER CUSTOMERS
  TYPE ORACLE
  VERSION 9
  WRAPPER NET8
  OPTIONS
      (NODE 'ABC',
      CPU_RATIO '2.0',
      IO_RATIO '1.5')
```

3. Register a server definition for the Excel wrapper. The server definition is required to preserve the hierarchy of federated objects. BIOCHEM_LAB is the name assigned to the Excel server definition. EXCEL_2000_WRAPPER is the name of the wrapper used to access this data source.

```
CREATE SERVER BIOCHEM_DATA
  WRAPPER EXCEL_2000_WRAPPER
```

4. Register a server definition for the ODBC wrapper. To access Apache Hive within DSN-less mode, the server type must be specified. HOST is the Hive server address. PORT is the connection port number that is used to access this data source. PORT is an optional parameter. If PORT is not defined, the default number is 10000.

```
CREATE SERVER HIVE_SERV TYPE HIVE
      WRAPPER ODBC AUTHORIZATION ‘root’ PASSWORD ‘hadoop’
       OPTIONS (HOST ‘hives.cn.ibm.com’, PORT ‘10000’,
               DBNAME ‘default’, PASSWORD ‘Y’, PUSHDOWN ‘Y’)
```

5. Register a server definition to access to the Db2 data source, which uses the default DRDA wrapper. The server option **SSO_AUTH** indicates that remote data sources uses a single sign-on (SSO) authentication mechanism, so the **AUTHORIZATION** keyword and **PASSWORD** keyword can be removed.

```
CREATE SERVER server1 TYPE DB2/LUW VERSION 11 OPTIONS(HOST, PORT 50000, DBNAME sample,
SSO_AUTH 'Y')
```

CREATE STOGROUP

The CREATE STOGROUP statement defines a new storage group within the database, assigns storage paths to the storage group, and records the storage group definition and attributes in the catalog.

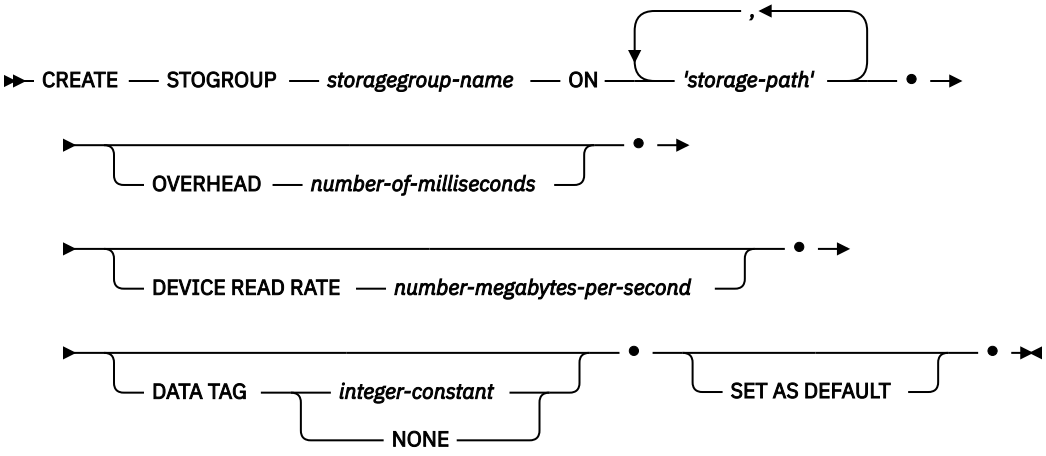
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges that are held by the authorization ID of the statement must include SYSCTRL or SYSADM authority.

Syntax



Description

storagegroup-name

Names the storage group. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *storagegroup-name* must not identify a storage group that exists at the current server (SQLSTATE 42710). The *storagegroup-name* must not begin with the characters 'SYS' (SQLSTATE 42939).

ON

Specifies storage paths to be added for the named storage group. For partitioned database environments, the same storage paths are defined on all database partitions unless database partition expressions are used.

storage-path

A string constant that specifies containers the location where automatic storage table spaces are to be created. The format of the string depends on the operating system, as illustrated in the following table:

Operating system	Format of storage path string
Linux AIX	An absolute path
Windows	The letter name of a drive

The string can include database partition expressions to specify database partition number information in the storage path.

The maximum length of a path is 175 characters (SQLSTATE 54036).

A storage path that is added must be valid according to the naming rules for paths, and must be accessible (SQLSTATE 57019). Similarly, in a partitioned database environment, the storage path must exist and be accessible on every database partition (SQLSTATE 57019).

OVERHEAD *number-of-milliseconds*
Specifies the I/O controller usage and disk seek and latency time. This value is used to determine the cost of I/O during query optimization. The value of *number-of-milliseconds* is any numeric literal (integer, decimal, or floating point). If this value is not the same for all storage paths, set the value to a numeric literal that represents the average for all storage paths that belong to the storage group.

If the OVERHEAD clause is not specified, the OVERHEAD is set to 6.725 milliseconds.

DEVICE READ RATE *number-megabytes-per-second*
Specifies the device specification for the read transfer rate in megabytes per second. This value is used to determine the cost of I/O during query optimization. The value of *number-megabytes-per-second* is any numeric literal (integer, decimal, or floating point). If this value is not the same for all storage paths, set the value to a numeric literal that represents the average for all storage paths that belong to the storage group.

If the DEVICE READ RATE clause is not specified, the DEVICE READ RATE is set to the built-in default of 100 megabytes per second.

DATA TAG *integer-constant* or DATA TAG NONE
Specifies a tag for the data for table spaces that use this storage group unless explicitly overridden by the table space definition. This value can be used as part of a WLM configuration in a work class definition or referenced within a threshold definition. For more information, see the CREATE WORK CLASS SET and CREATE THRESHOLD statements.

integer-constant
Valid values for *integer-constant* are integers 1 - 9.

NONE
If NONE is specified, there is no data tag.

SET AS DEFAULT
Specifies the storage group that is created is designated as the default storage group. If no default storage group exists, the first one created is designated the default even if this clause is not specified. Since there can only be one storage group that is designated as the default storage group, specifying this clause removes the default attribute from the existing default storage group. Specifying a new default storage group has no effect to the storage group used by existing table spaces.

Rules

- The CREATE STOGROUP statement cannot be run while a database partition server is being added (SQLSTATE 55071).
- A storage group can have up to 128 defined storage paths (SQLSTATE 5U009).
- A database instance can have up to 256 defined storage groups (SQLSTATE 54035).

Notes

- ***Calculation of free space:*** When free space is calculated for a storage path on a database partition, the database manager checks for the existence of the following directories or mount points within the storage path. The database manager uses the first one that is found.

```
<storage path>/<instance name>/NODE####/<database name>
<storage path>/<instance name>/NODE####
<storage path>/<instance name>
<storage path>
```

Where:

– `<storage path>` is a storage path that is associated with the database.

  - <instance name> is the instance under which the database resides.
  - NODE#### corresponds to the database partition number (for example, NODE0000 or NODE0001).
  - <database name> is the name of the database.
- ***Isolating multiple database partitions under one storage path:*** The file systems can be mounted at a point beneath the storage path, and the database manager recognizes that the actual amount of free space available for table space containers might not be the same amount that is associated with the storage path directory itself.

  Consider an example in which two logical database partitions exist on one physical computer, and a single storage path exists (/dbdata). Each database partition uses this storage path, but you might want to isolate the data from each partition within its own file system. In this case, a separate file system can be created for each partition and it can be mounted at /dbdata/<instance>/NODE####. When creating containers on the storage path and determining free space, the database manager does not retrieve free space information for /dbdata, but instead retrieves it for the corresponding /dbdata/<instance>/NODE#### directory.
- ***Multiple storage paths:*** A storage path can be added to different storage groups, or to the same storage group multiple times.
- ***Similar media characteristics:*** Ensure that the storage paths added to a storage group share similar media characteristics. If the media characteristics are dissimilar, specify a value that represents an average for OVERHEAD and DEVICE READ RATE.

Examples

1. Create a storage group that is named HIGHEND with two paths under the /db directory (/db/filesystem1 and /db/filesystem2) which are attached to Solid State Disks.

```
CREATE STOGROUP HIGHEND ON '/db/filesystem1', '/db/filesystem2'
   OVERHEAD 0.75 DEVICE READ RATE 500
```

2. Create a storage group that is named MIDRANGE with two drives D and E and designate it as the default storage group.

```
CREATE STOGROUP MIDRANGE ON 'D:\', 'E:\' SET AS DEFAULT
```

3. Create a storage group that is named MIDRANGE with two paths under the /db directory, and designate it as the default storage group.

```
CREATE STOGROUP MIDRANGE ON '/db/filesystem1', '/db/filesystem2' SET AS DEFAULT
```

CREATE SYNONYM

The CREATE SYNONYM statement defines a synonym for a module, nickname, sequence, table, view, or another synonym.

Description

SYNONYM is a synonym for ALIAS.

CREATE TABLE

The CREATE TABLE statement defines a table. The definition must include its name and the names and attributes of its columns. The definition can include other attributes of the table, such as its primary key or check constraints.

To create a created temporary table, use the CREATE GLOBAL TEMPORARY TABLE statement. To declare a declared temporary table, use the DECLARE GLOBAL TEMPORARY TABLE statement.

Invocation

This statement can be embedded in an application program or issued by using dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The authorization ID of the statement must have either DBADM authority, or must have CREATETAB authority in combination with the following additional authorization:

- One of the following privileges or authorities:
  - USE privilege on the table space
  - SYSADM authority
  - SYSCTRL authority
- Plus one of these privileges or authorities:
  - IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the table does not exist.
  - CREATEIN privilege on the schema, if the schema name of the table refers to an existing schema.
  - SCHEMAADM authority on the schema, if the schema name of the table refers to an existing schema.

If a subtable is being defined, at least one of the following conditions must be met:

- The authorization ID must be the same as the owner of the root table of the table hierarchy.
- The privileges that are held by the authorization ID must include SCHEMAADM authority on the schema that contains the root table of the table hierarchy.
- The privileges that are held by the authorization ID must include DBADM authority.

To define a foreign key, the authorization ID of the statement must have one of the following privileges for the parent table:

- REFERENCES privilege on the table
- REFERENCES privilege on each column of the specified parent key
- CONTROL privilege on the table
- SCHEMAADM authority on the schema, if the schema name of the parent table refers to an existing schema.
- DBADM authority

To define a materialized query table, the following conditions must be met:

- The authorization ID of the statement must have at least one of the following privileges on each table or view that is identified in the fullselect (privileges that are held through groups are not considered):
  - SELECT privilege on the table or view
  - CONTROL privilege on the table or view
  - SELECTIN privilege on the schema that contains the table or view
  - DATAACCESS authority on the schema that contains the table or view
  - DATAACCESS authority
- The authorization ID of the statement must have at least one of the following privileges on each table that is identified in the fullselect (this is required for altering the base table to associate it with the materialized query table):
  - ALTER privilege on the table or view
  - CONTROL privilege on the table or view
  - SCHEMAADM authority on the schema that contains the table or view

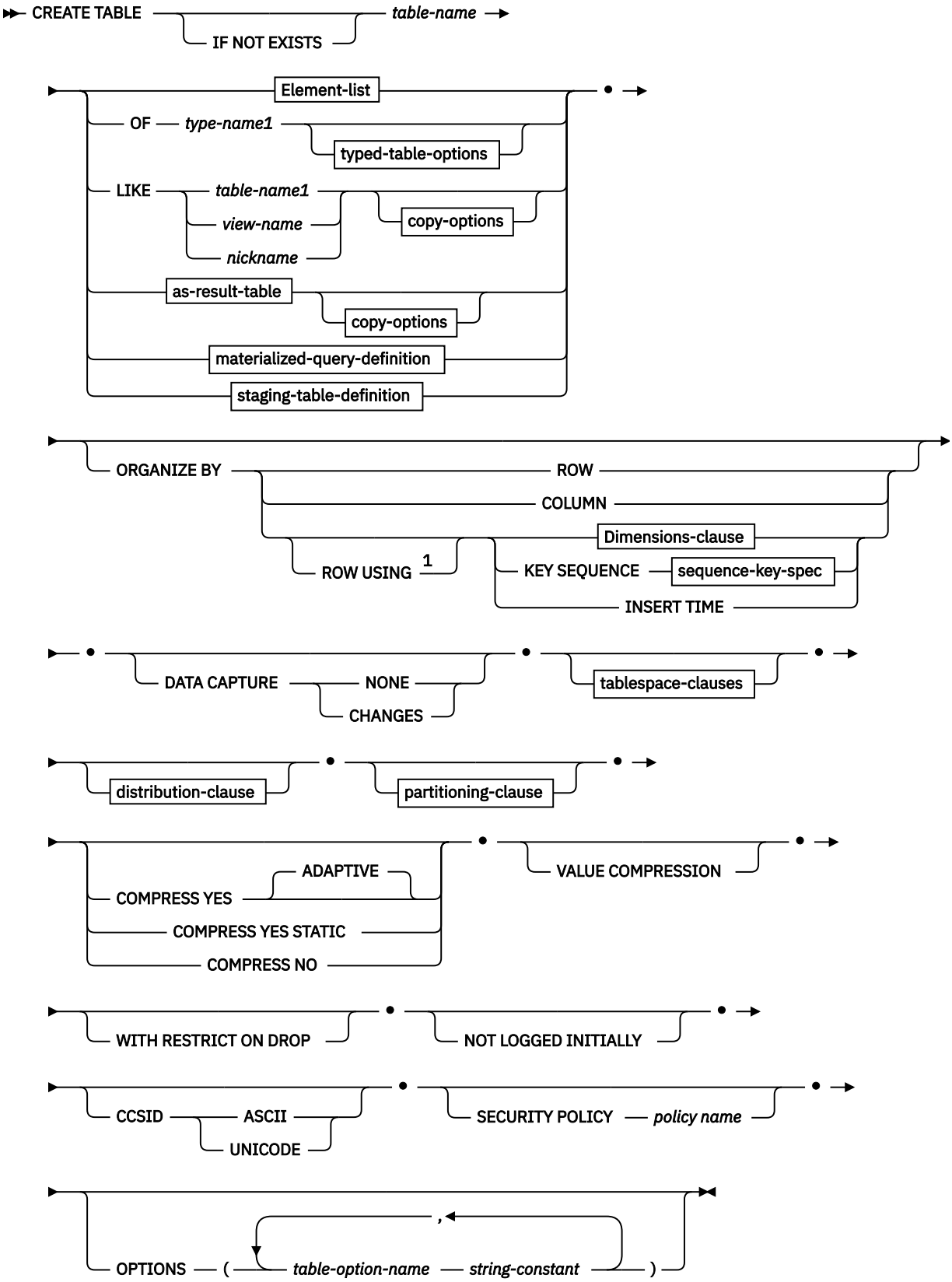
  - DBADM authority

To define a staging table that is associated with a materialized query table, the authorization ID of the statement must hold the following privileges:

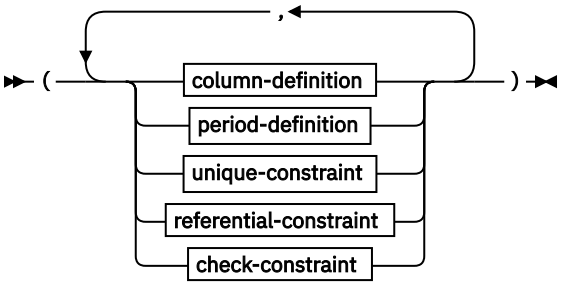
- At least one of the following privileges for the materialized query table:
  - ALTER privilege on the materialized query table
  - CONTROL privilege on the materialized query table
  - SCHEMAADM authority on the schema that contains the materialized query table
  - DBADM authority
- At least one of the following privileges for each table or view that is identified in the fullselect of the materialized query table:
  - SELECT privilege on the table or view
  - CONTROL privilege on the table or view
  - SELECTIN privilege on the schema that contains the table or view
  - DATAACCESS authority on the schema that contains the table or view
  - DATAACCESS authority on the database

  In addition, at least one of the following privileges for each table or view identified in the fullselect of the materialized query table:
  - ALTER privilege on the table or view
  - CONTROL privilege on the table or view
  - SCHEMAADM authority on the schema that contains the table or view
  - DBADM authority

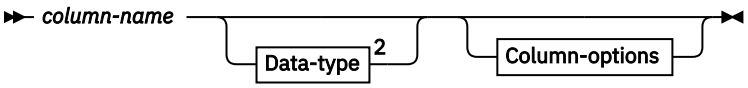
Syntax



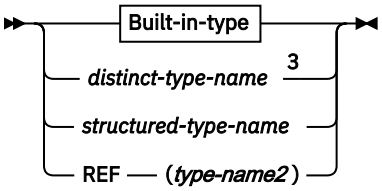
Element-list



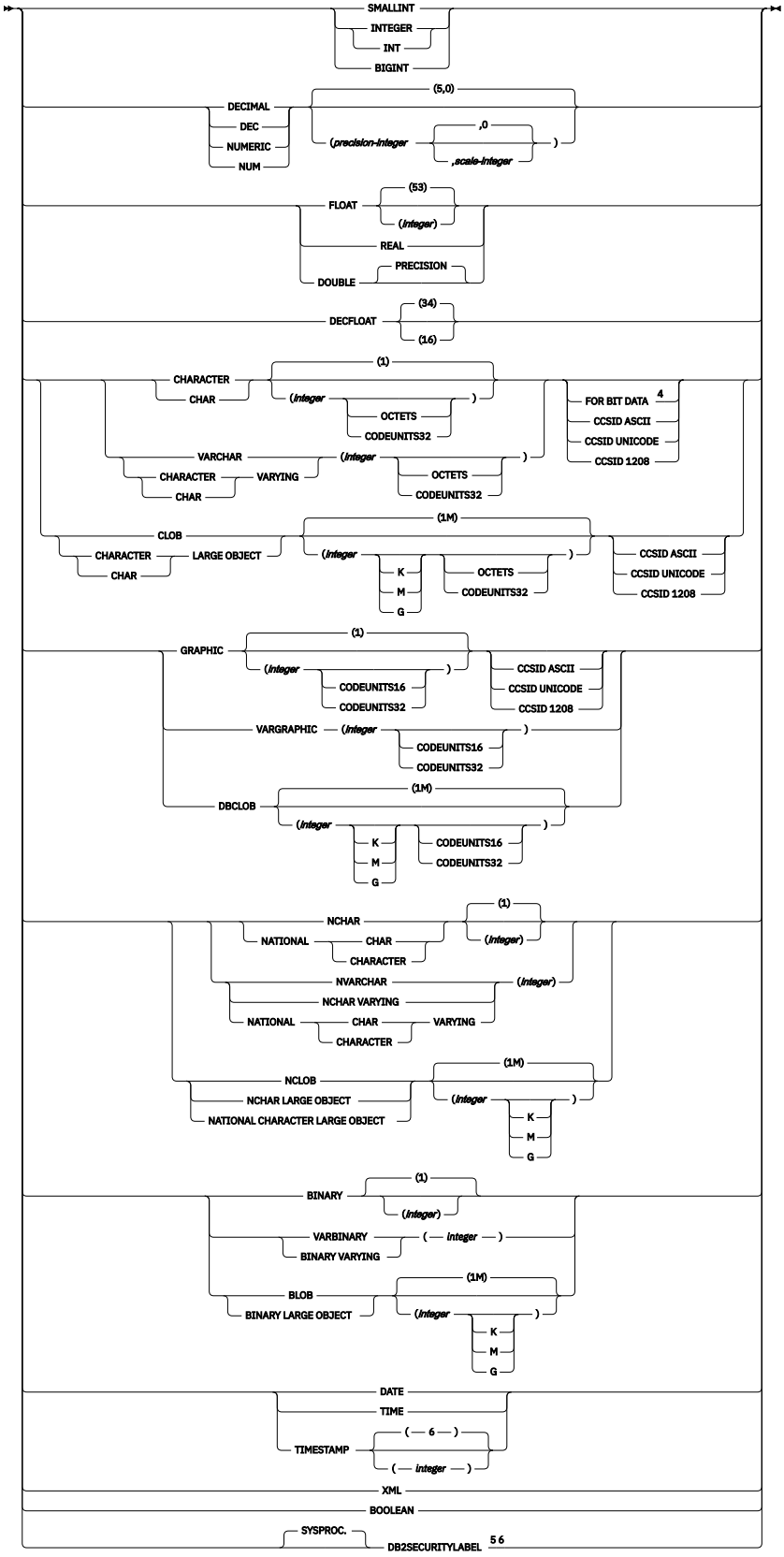
Column-definition



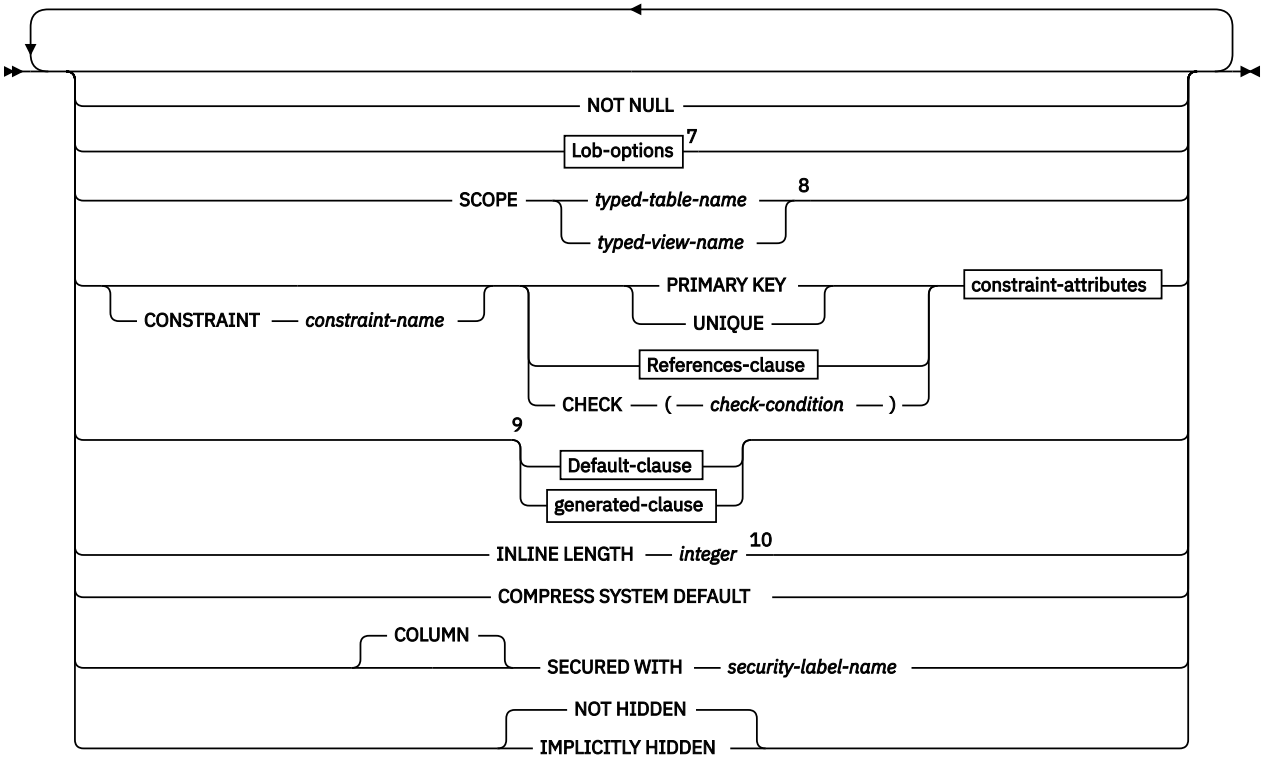
Data-type



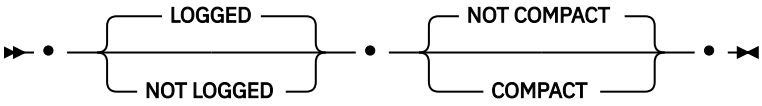
Built-in-type



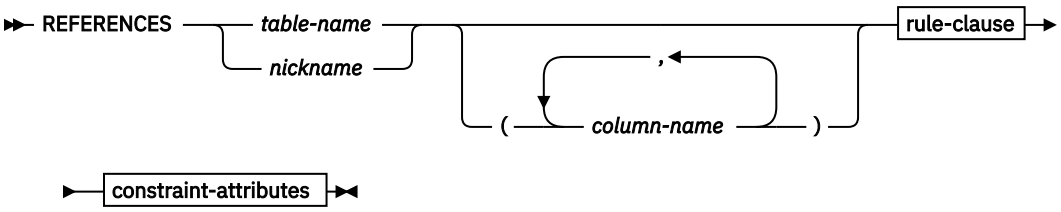
Column-options



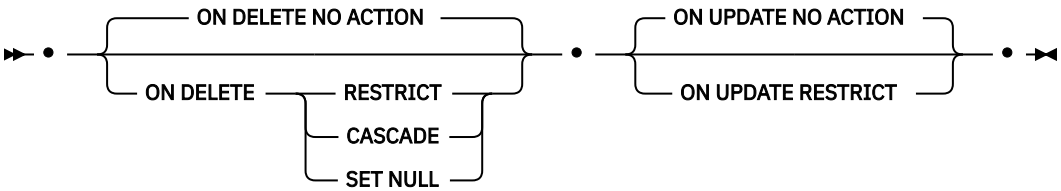
Lob-options



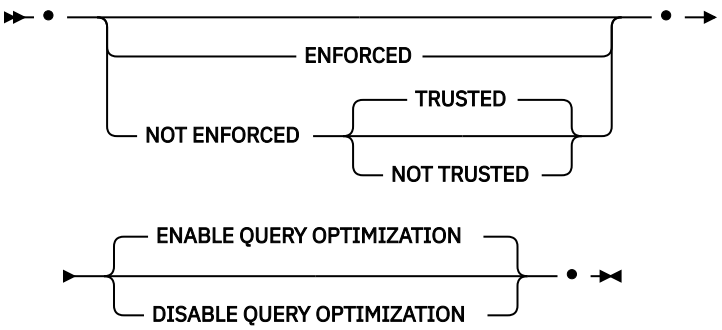
References-clause



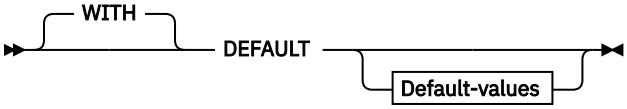
Rule-clause



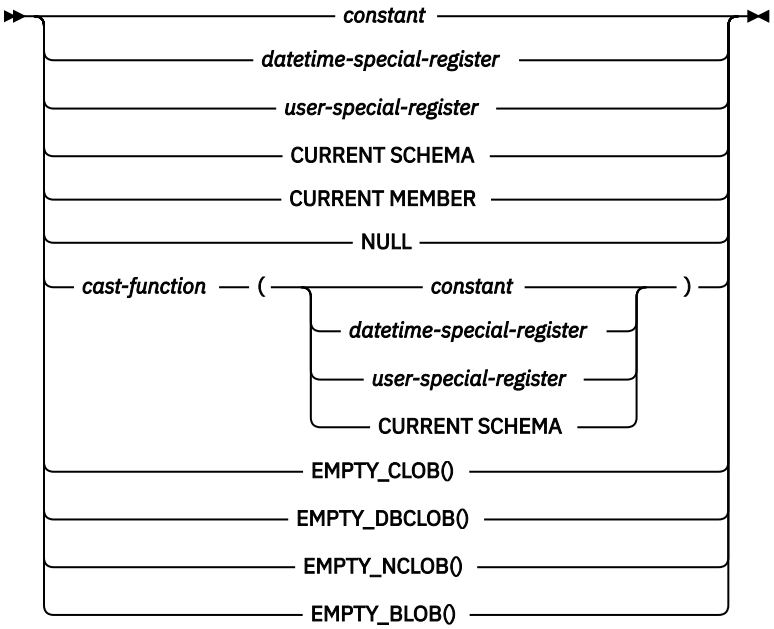
Constraint-attributes



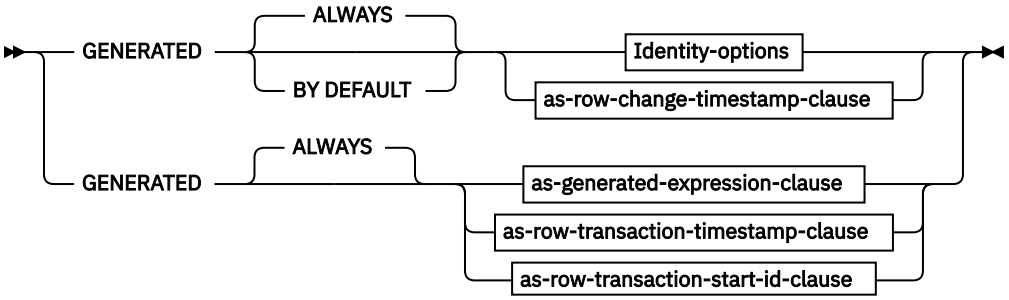
Default-clause



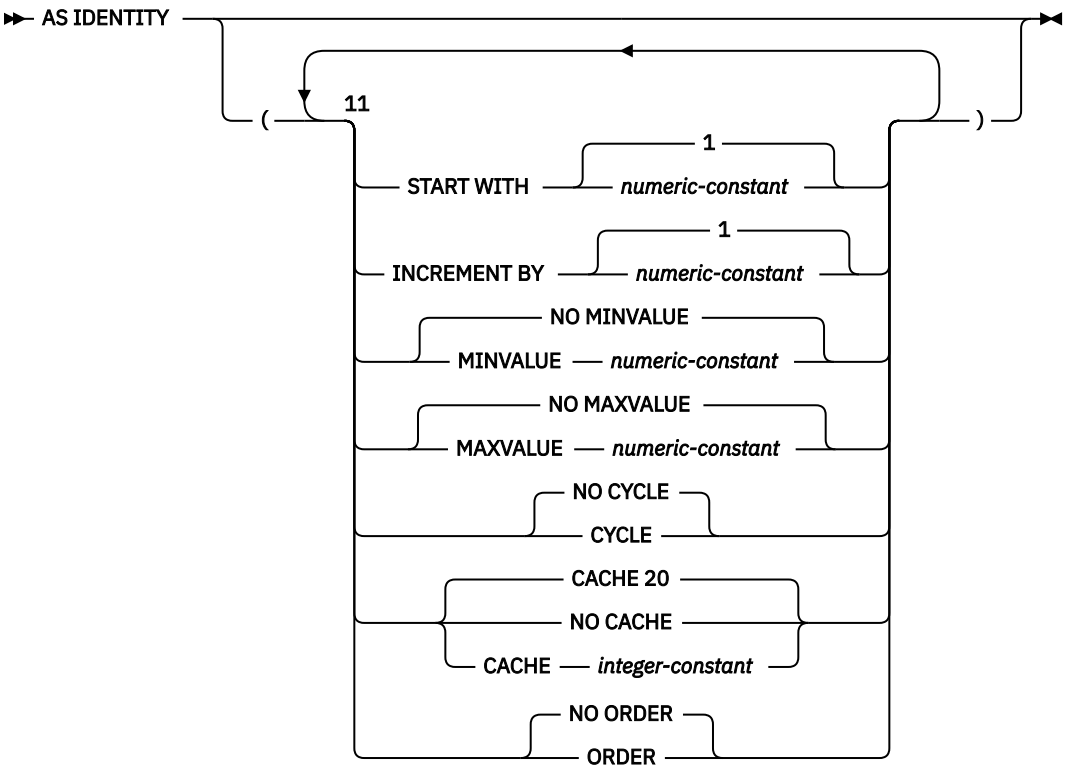
Default-values



Generated-clause



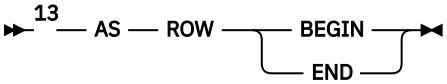
Identity-options



As-row-change-timestamp-clause

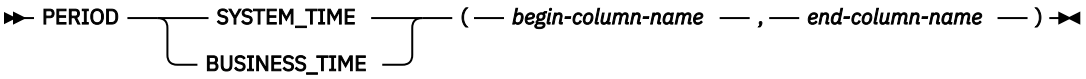
As-generated-expression-clause

As-row-transaction-timestamp-clause

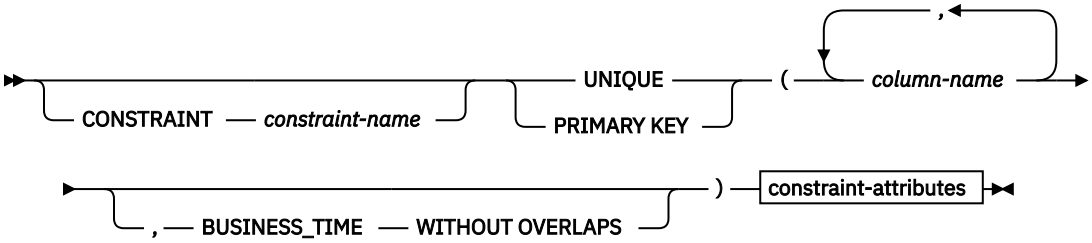


As-row-transaction-start-id-clause

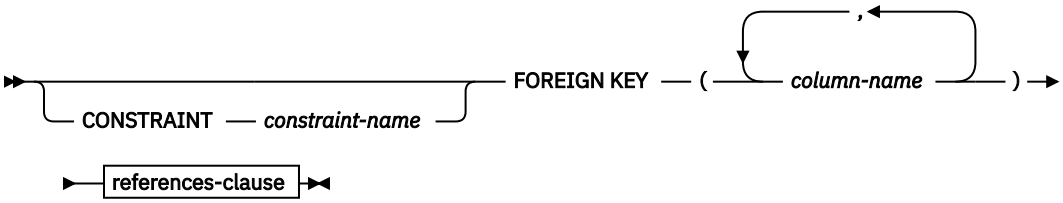
Period-definition



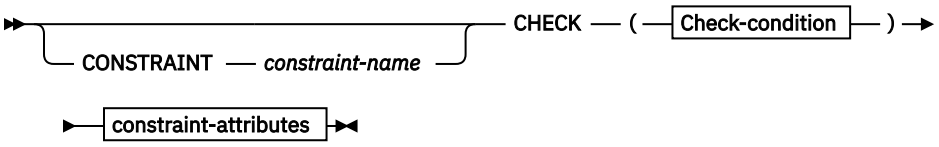
Unique-constraint



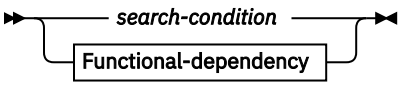
Referential-constraint



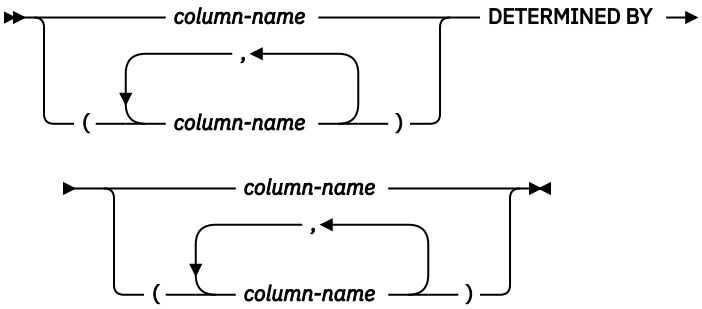
Check-constraint



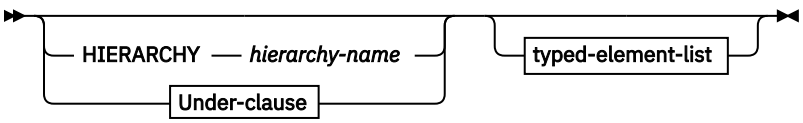
Check-condition



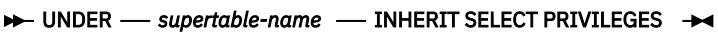
Functional-dependency



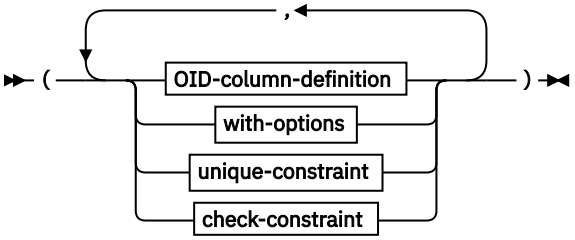
Typed-table-options



Under-clause



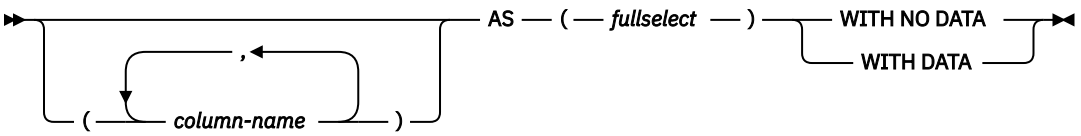
Typed-element-list



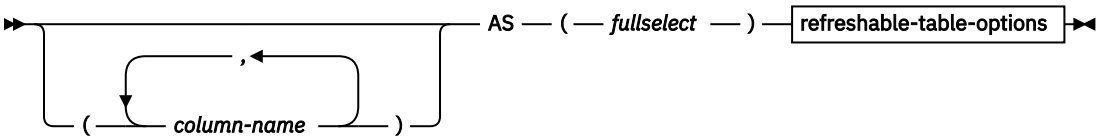
OID-column-definition

With-options

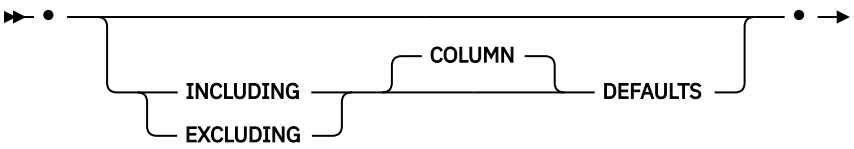
As-result-table

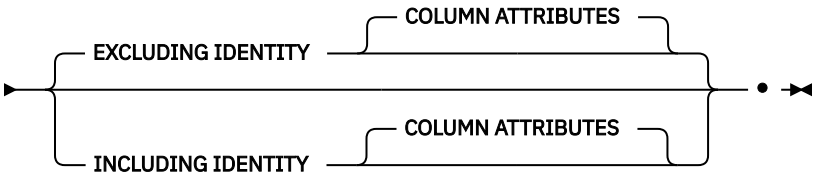


Materialized-query-definition

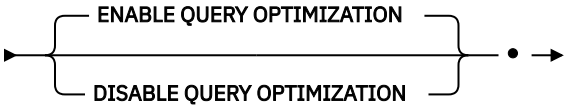


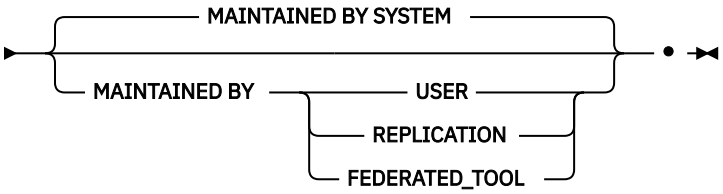
Copy-options



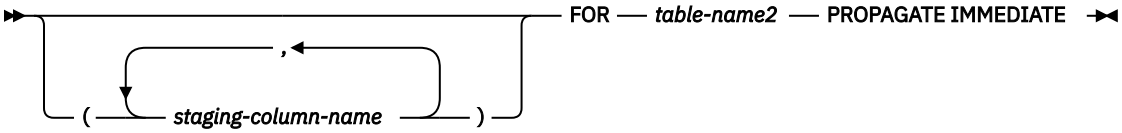


Refreshable-table-options

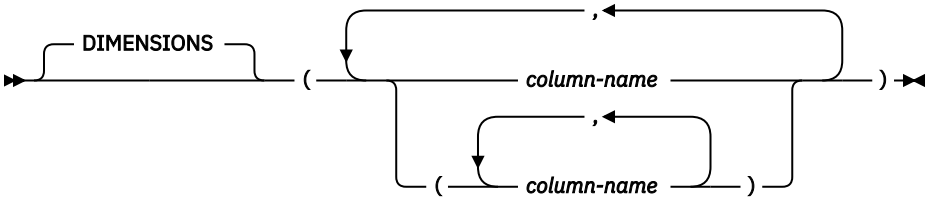




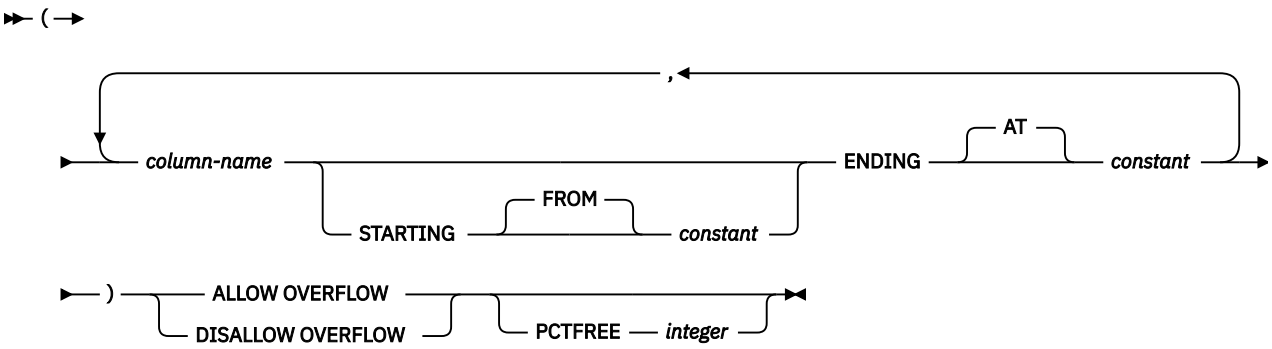
Staging-table-definition



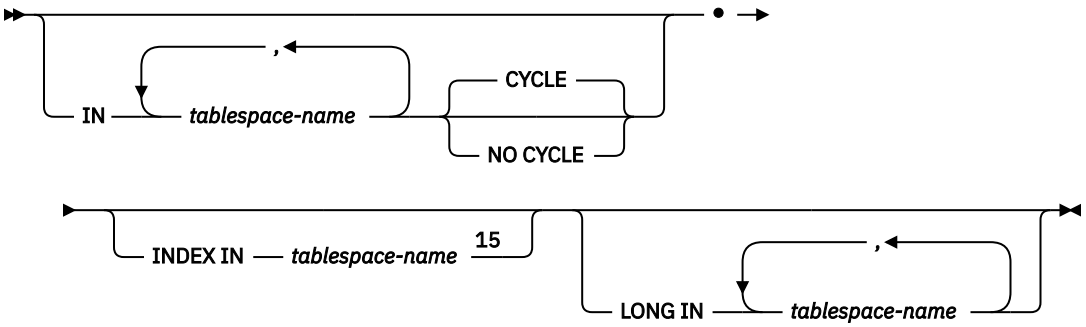
Dimensions-clause



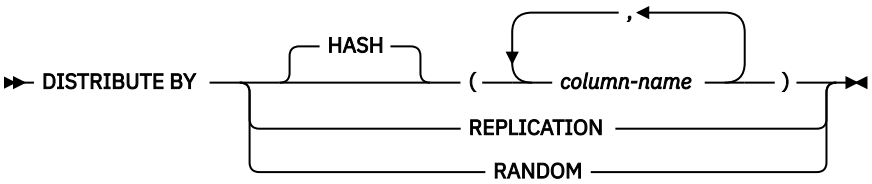
Sequence-key-spec



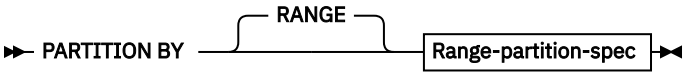
Tablespace-clauses



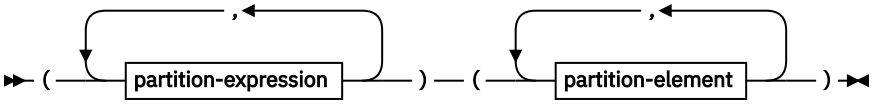
Distribution-clause



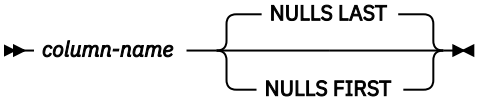
Partitioning-clause



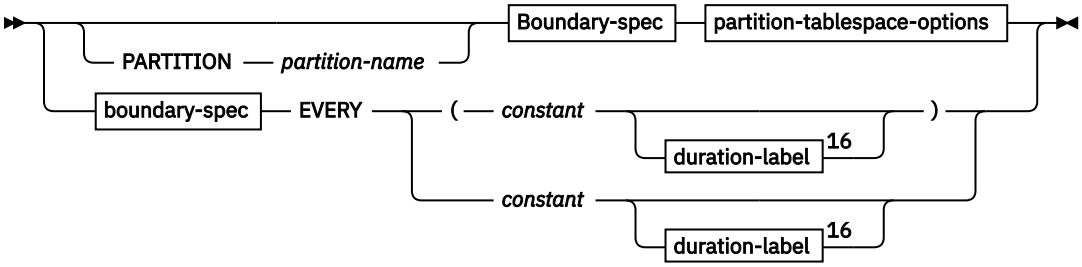
Range-partition-spec



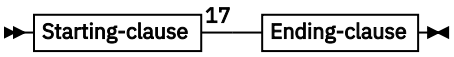
Partition-expression



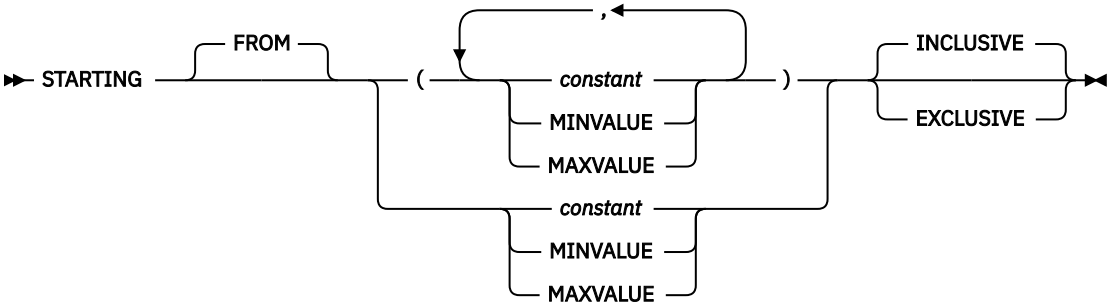
Partition-element



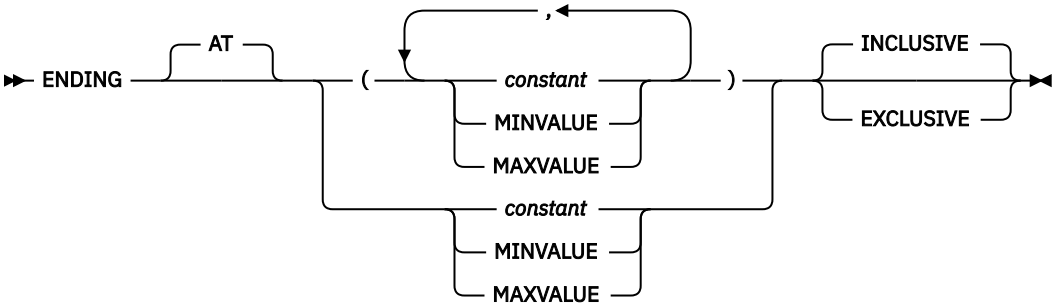
Boundary-spec



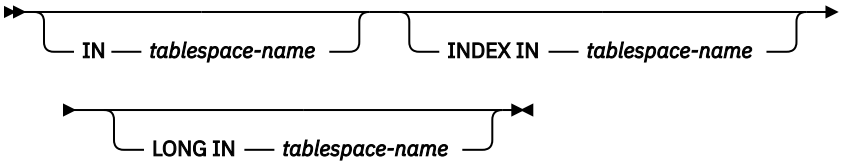
Starting-clause



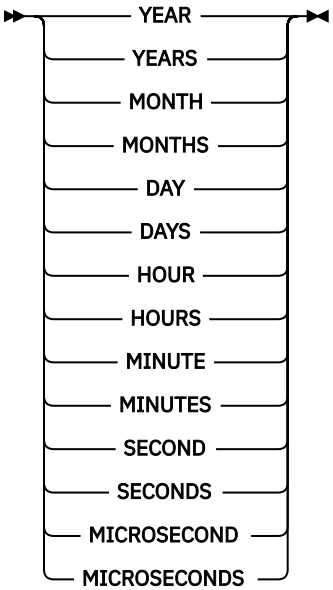
Ending-clause



Partition-tablespace-options



Duration-label



Notes:

[1] If you specify a dimensions clause, key sequence, or insert time, specifying ROW USING is optional unless the default table organization for the database is COLUMN, in which case specifying ROW USING is mandatory.

[2] If the first column-option chosen is a generated-clause with a generation-expression, then the data-type can be omitted. It will be determined from the resulting data-type of the generation-expression.
[3] The specified distinct type cannot have any data type constraints and the source type cannot be an anchored data type.
[4] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).
[5] DB2SECURITYLABEL is the built-in distinct type that must be used to define the row security label column of a protected table.
[6] A column of type DB2SECURITYLABEL, NOT NULL WITH DEFAULT is implicit and cannot be explicitly specified (SQLSTATE 42842). The default value for a column of type DB2SECURITYLABEL is the session authorization ID's security label for write access.
[7] The lob-options clause only applies to large object types (BLOB, CLOB, and DBCLOB) and distinct types based on large object types.
[8] The SCOPE clause only applies to the REF type.
[9] The default-clause and generated-clause cannot both be specified for the same column definition (SQLSTATE 42614).
[10] INLINE LENGTH applies only to columns defined as structured, XML, or LOB types.
[11] The same clause must not be specified more than once.
[12] Data type is optional for a row change timestamp column if the first column-option specified is a generated-clause. The data type default is TIMESTAMP(6).
[13] Data type is optional for row-begin and row-end timestamp columns if the first column-option specified is a generated-clause. The data type default is TIMESTAMP(12).
[14] Data type is optional for a transaction-start-ID timestamp columns if the first column-option specified is a generated-clause. The data type default is TIMESTAMP(12).
[15] Specifying which table space contains a table's indexes can be done when the table is created. If the table is a partitioned table, the index table space for a nonpartitioned index can be specified with the IN clause of the CREATE INDEX statement.
[16] This syntax for a partition-element is valid if only one partition-expression exists with a numeric or datetime data type.
[17] The first partition-element must include a starting-clause and the last partition-element must include an ending-clause.

Description

System-maintained, user-maintained, federated_tool-maintained, and replication-maintained materialized query tables (shadow tables) are referred to by the common term *materialized query table*, unless a need exists to identify each one separately.

IF NOT EXISTS

Specifies that no error message is shown when the table cannot be created because a table with the specified name already exists in the current database and schema. Typically, you use this option for scripted applications that are running SQL commands. When you suppress the `Table not found` error message, the scripted application is not impacted or halted.

The following conditions apply when you use this option:

- You cannot use the IF NOT EXISTS option with the AS SELECT clause. Using the IF NOT EXISTS option with the AS SELECT clause causes a syntax error.
- Unless other errors prevent the creation of the table, a CREATE TABLE message is returned although no table is created. The reason is that the failure is ignored if a table with the specified name already exists.
- The existing table and the specified table in the command are not compared, that is, the tables might have different sizes. The exisiting table remains as is with its current size. The content of the rows is not changed. The application must ensure that the target table and rows are as expected.

table-name
Names the table. The name, including the implicit or explicit qualifier, must not identify a table, view, nickname, or alias described in the catalog. The schema name must not be SYSIBM, SYSCAT, SYSFUN, or SYSSTAT (SQLSTATE 42939).

element-list
Defines the elements of a table, including the definition of columns and constraints on the table.

column-definition
Defines the attributes of a column.

column-name
Names a column of the table. The name cannot be qualified, and the same name cannot be used for more than one column of the table (SQLSTATE 42711).

A row-organized table can have the following:

- A 4K page size with a maximum of 500 columns, where the row size must not be greater than 4005.
- An 8K page size with a maximum of 1012 columns, where the row size must not be greater than 8101.
- A 16K page size with a maximum of 1012 columns, where the row size must not be greater than 16,293.
- A 32K page size with a maximum of 1012 columns, where the row size must not be greater than 32,677.

A column-organized table can have a maximum of 1012 columns, regardless of page size, where the byte counts of the columns must not be greater than 32,677. Extended row size support does not apply to column-organized tables.

For more information, see Row Size Limit.

data-type
Specifies the data type of the column.

built-in-type
One of the following built-in data types:

SMALLINT
A small integer.

[INTEGER | INT]
A large integer.

BIGINT
A big integer.

[DECIMAL | DEC | NUMERIC | NUM](*precision-integer, scale-integer*)
A decimal number.

- The precision integer specifies the total number of digits. It must be in the range 1 - 31. The default is 5.
- The scale integer specifies the number of digits to the right of the decimal point. It cannot be negative and cannot exceed the precision. The default is 0.

FLOAT(*integer*)
A single or double-precision floating-point number. If the specified length is in the range:

- 1 - 24, the number uses single precision.
- 25 - 53, the number uses double-precision.

Instead of FLOAT, you can specify:

REAL
For single precision floating-point.

DOUBLE
For double-precision floating-point.

DOUBLE PRECISION
For double-precision floating-point.

FLOAT
For double-precision floating-point.

DECFLOAT*(precision-integer)*
A decimal floating-point number. The precision integer specifies the total number of digits, which can be either 16 or 34. The default is 34.

[CHARACTER | CHAR](*integer* [OCTETS | CODEUNITS32])
A fixed-length character string of the specified number of code units. This number can range from 1 - 255 OCTETS or from 1 - 63 CODEUNITS32. The default is 1.

[VARCHAR | CHARACTER VARYING | CHAR VARYING](*integer* [OCTETS | CODEUNITS32])
A varying-length character string with a maximum length of the specified number of code units. This number can range from 1 - 32672 OCTETS or from 1 - 8168 CODEUNITS32.

FOR BIT DATA
Specifies that the contents of the column are to be treated as bit (binary) data. During data exchange with other systems, code page conversions are not performed. Comparisons are done in binary, irrespective of the database collating sequence.

CCSID
Specifies the encoding scheme for string data that is stored in the column. If the CCSID clause is not specified, the default is the CCSID of the table.

ASCII
Specifies that string data is encoded in the database code page. If the table is a Unicode table, CCSID ASCII cannot be specified (SQLSTATE 56031).

UNICODE, 1208, 1200
Specifies that string data is encoded in Unicode. Character data is in UTF-8; graphic data is in UTF-16 BE. CCSID 1208 and 1200 are synonyms for CCSID UNICODE. CCSID UNICODE cannot be specified for an SBCS database (SQLSTATE 560AA).

If the table is not a Unicode table, columns can be created with CCSID UNICODE, but the following rules apply:

- The alternative collating sequence must be specified in the database configuration before creating the table (SQLSTATE 56031). CCSID UNICODE columns collate with the alternative collating sequence that is specified in the database configuration.

  The only supported alternative collating sequence is IDENTITY_16BIT.
- The column cannot be a graphic data type.
- Anchored data types cannot anchor to a column that is created with CCSID UNICODE (SQLSTATE 428HS).
- Tables cannot have both the CCSID UNICODE clause and the DATA CAPTURE CHANGES clause specified (SQLSTATE 42613).
- Created temporary tables and declared temporary tables cannot have columns declared with CCSID UNICODE (SQLSTATE 56031).
- CCSID UNICODE columns cannot be specified in a CREATE SCHEMA statement (SQLSTATE 53090).
- A column of the exception table for a load operation must have the same CCSID as the corresponding target table column for the operation (SQLSTATE 428A5).
- A column of the exception table for a SET INTEGRITY statement must have the same CCSID as the corresponding target table column for the statement (SQLSTATE 53090).

- Columns of the target table for event monitor data must not be declared as CCSID UNICODE (SQLSTATE 55049).

[CLOB | CHARACTER LARGE OBJECT | CHAR LARGE OBJECT](*integer* [K | M | G] [OCTETS | CODEUNITS32])

A character large object string with a maximum length of the specified number of code units. The default is 1,048,576 (1M) code units.

If you want to multiply the length integer by 1024 (kilo), 1,048,576 (mega), or 1,073,741,824 (giga), specify a K (kilo), M (mega), or G (giga) multiplier.

- Regardless of which multiplier, if any, you use, the resulting length is limited by the maximum length of a CLOB column, which is 2,147,483,646 (for OCTETS) or 536,870,911 (for CODEUNITS32). If a multiple of K, M, or G slightly exceeds this maximum length (for example, 2G = 2,147,483,648), the maximum length is used instead.
- Any number of spaces (including zero spaces) is allowed between data type and the length specification or between the length integer and the K, M, or G multiplier. For example, the following specifications are all equivalent and valid:

```
CLOB(50K)
CLOB(50 K)
CLOB (50   K)
```

- The K, M, or G multiplier can be specified in either uppercase or lowercase.

In a Unicode database, the default string units for a character string data type are determined by the value of the NLS_STRING_UNITS global variable or *string_units* database configuration parameter. In a non-Unicode database, the default string units for character string data types are OCTETS.

OCTETS
Specifies that the units of the length attribute are bytes.

CODEUNITS32
Specifies that the units of the length attribute are Unicode UTF-32 code units, which approximate counting in characters. This does not affect the underlying code page of the data type. The actual length of a data value is determined by counting the UTF-32 code units as if the data were converted to UTF-32. CODEUNITS32 can be specified only in a Unicode database (SQLSTATE 560AA).

GRAPHIC(*integer* [CODEUNITS16 | CODEUNITS32])

A fixed-length graphic string of the specified length, which can range from 1 - 127 double bytes, 1 - 127 CODEUNITS16, or 1 - 63 CODEUNITS32. The default length is 1.

VARGRAPHIC(*integer* [CODEUNITS16 | CODEUNITS32])

A varying-length graphic string of the specified maximum length, which can range from 1 - 16336 double bytes, 1 - 16336 CODEUNITS16, or 1 - 8168 CODEUNITS32.

DBCLOB(*integer* [K | M | G] [CODEUNITS16 | CODEUNITS32])

A character large object string of the specified maximum length in double bytes, Unicode UTF-16 code units, or Unicode UTF-32 code units. The default is 1,048,576 (1M) code units.

If you want to multiply the length integer by 1024 (kilo), 1,048,576 (mega), or 1,073,741,824 (giga), specify a K (kilo), M (mega), or G (giga) multiplier.

- Regardless of which multiplier, if any, you use, the resulting length is limited by the maximum length of a DBCLOB column, which is which is 1,073,741,823 (for double bytes or CODEUNITS16) or 536,870,911 (for CODEUNITS32). If a multiple of K, M, or G slightly exceeds this maximum length (for example, 1G = 1,073,741,824), the maximum length is used instead.

- Any number of spaces (including zero spaces) is allowed between data type and the length specification or between the length integer and the K, M, or G multiplier. For example, the following specifications are all equivalent and valid:

```
DBCLOB(50K)
DBCLOB(50 K)
DBCLOB (50   K)
```

- The K, M, or G multiplier can be specified in either uppercase or lowercase.

In a Unicode database, the default string units for a character string data type are determined by the value of the NLS_STRING_UNITS global variable or *string_units* database configuration parameter. In a non-Unicode database, the default string units for character string data types is CODEUNITS16.

CODEUNITS16
Specifies that the units of the length attribute are Unicode UTF-16 code units, which are the same as counting in double bytes. CODEUNITS16 can be specified only in a Unicode database (SQLSTATE 560AA).

CODEUNITS32
Specifies that the units of the length attribute are Unicode UTF-32 code units. This does not affect the underlying code page of the data type. The actual length of a data value is determined by counting the UTF-32 code units as if the data were converted to UTF-32. CODEUNITS32 can be specified only in a Unicode database (SQLSTATE 560AA).

[NATIONAL CHARACTER | NATIONAL CHAR | NCHAR](*integer*)
A fixed-length string of the specified length. The default length is 1.

The NATIONAL CHARACTER type maps to either a fixed-length character or a fixed-length graphic string, depending on the value of the *nchar_mapping* database configuration parameter, which also defines the string units.

[NATIONAL CHARACTER VARYING | NATIONAL CHAR VARYING | NCHAR VARYING | NVARCHAR](*integer*)
A varying-length string of the specified maximum length.

The NATIONAL CHARACTER VARYING type maps to either a varying-length character or a varying-length graphic string, depending on the value of the *nchar_mapping* database configuration parameter, which also defines the string units.

[NATIONAL CHARACTER LARGE OBJECT | NCHAR LARGE OBJECT | NCLOB](*integer* [K | M | G])
A large object string of the specified maximum length.

This data type maps to either a character large object (CLOB) or a double-byte character large object (DBCLOB), depending on the current value of the *nchar_mapping* database configuration parameter, which also defines the string units. See the description of the CLOB or DBCLOB parameter (whichever applies) for information about possible values for the length integer and how to use a K (kilo), M (mega), or G (giga) multiplier.

BINARY(*integer*)
A fixed-length binary string of the specified length, which must be in the range 1 - 255 bytes. The default length is 1.

[VARBINARY | BINARY VARYING](*integer*)
A varying-length binary string of the specified maximum length, which must be in the range 1 - 32672 bytes.

[BLOB | BINARY LARGE OBJECT](*integer* [K | M | G])
A binary large object string of the specified maximum length. The default is 1,048,576 (1M) bytes.

If you want to multiply the length integer by 1024 (kilo), 1,048,576 (mega), or 1,073,741,824 (giga), specify a K (kilo), M (mega), or G (giga) multiplier.

- Regardless of which multiplier, if any, you use, the resulting length is limited by the maximum length of a BLOB column, which is 2,147,483,647 bytes. If a multiple of K, M, or G slightly exceeds this maximum length (for example, 2G = 2,147,483,648), the maximum length is used instead.
- Any number of spaces (including zero spaces) is allowed between data type and the length specification or between the length integer and the K, M, or G multiplier. For example, the following specifications are all equivalent and valid:

```
BLOB(50K)
BLOB(50 K)
BLOB (50   K)
```

- The K, M, or G multiplier can be specified in either uppercase or lowercase.

DATE
A date.

TIME
A time.

TIMESTAMP(*integer*) or TIMESTAMP
A time stamp. The integer specifies the precision of fractional seconds from 0 (seconds) to 12 (picoseconds). The default is 6 (microseconds).

XML
An XML document. Only well-formed XML documents can be inserted into an XML column.

An XML column has the following restrictions:

- The column cannot be part of any index except an index over XML data. Therefore, it cannot be included as a column of a primary key or unique constraint (SQLSTATE 42962).
- The column cannot be a foreign key of a referential constraint (SQLSTATE 42962).
- A default value (WITH DEFAULT) cannot be specified for the column (SQLSTATE 42613). If the column is nullable, the default for the column is the null value.
- The column cannot be used as the distribution key (SQLSTATE 42997).
- The column cannot be used as a data partitioning key (SQLSTATE 42962).
- The column cannot be used to organize a multidimensional clustering (MDC) table (SQLSTATE 42962).
- The column cannot be used in a range-clustered table (SQLSTATE 429BG).
- The column cannot be referenced in a check constraint except in a VALIDATED predicate (SQLSTATE 42621).

When a column of type XML is created, an XML path index is created on that column. A table-level XML region index is also created when the first column of type XML is created. The name of these indexes is "SQL" followed by a character time stamp (*yymmddhhmmssxxx*). The schema name is SYSIBM.

BOOLEAN
A Boolean value.

SYSPROC.DB2SECURITYLABEL
A built-in distinct type that must be used to define the row security label column of a protected table. The underlying data type of a column of the built-in distinct type DB2SECURITYLABEL is VARCHAR(128) FOR BIT DATA. A table can have at most one column of type DB2SECURITYLABEL (SQLSTATE 428C1).

distinct-type-name
For a user-defined type that is a distinct type. If a distinct type name is specified without a schema name, the distinct type name is resolved by searching the schemas on the SQL path (defined by the FUNCPATH preprocessing option for static SQL and by the CURRENT PATH register for dynamic SQL).

If a column is defined by using a distinct type, then the data type of the column is the distinct type. The length and the scale of the column are the length and the scale of the source type of the distinct type. The specified distinct type cannot have any data type constraints and the source type cannot be an anchored data type (SQLSTATE 428H2).

If a column defined by using a distinct type is a foreign key of a referential constraint, then the data type of the corresponding column of the primary key must have the same distinct type.

structured-type-name
For a user-defined type that is a structured type. If a structured type name is specified without a schema name, the structured type name is resolved by searching the schemas on the SQL path (defined by the FUNCPATH preprocessing option for static SQL, and by the CURRENT PATH register for dynamic SQL).

If a column is defined by using a structured type, then the static data type of the column is the structured type. The column can include values with a dynamic type that is a subtype of *structured-type-name*.

A column that is defined by using a structured type cannot be used in a primary key, unique constraint, foreign key, index key, or distribution key (SQLSTATE 42962).

If a column is defined by using a structured type, and contains a reference-type attribute at any level of nesting, that reference-type attribute is unscoped. To use such an attribute in a dereference operation, it is necessary to specify a SCOPE explicitly, using a CAST specification.

REF (*type-name2*)
For a reference to a typed table. If *type-name2* is specified without a schema name, the type name is resolved by searching the schemas on the SQL path (defined by the FUNCPATH preprocessing option for static SQL and by the CURRENT PATH register for dynamic SQL). The underlying data type of the column is based on the representation data type specified in the REF USING clause of the CREATE TYPE statement for *type-name2* or the root type of the data type hierarchy that includes *type-name2*.

column-options
Defines additional options that are related to columns of the table.

NOT NULL
Prevents the column from containing null values.

If NOT NULL is not specified, the column can contain null values, and its default value is either the null value or the value that is provided by the WITH DEFAULT clause.

NOT HIDDEN or IMPLICITLY HIDDEN
Specifies whether the column is to be defined as hidden. The hidden attribute determines whether the column is included in an implicit reference to the table, or whether it can be explicitly referenced in SQL statements. The default is NOT HIDDEN.

NOT HIDDEN
Specifies that the column is included in implicit references to the table, and that the column can be explicitly referenced.

IMPLICITLY HIDDEN
Specifies that the column is not visible in SQL statements unless the column is explicitly referenced by name. For example, assuming that a table includes a column that is defined with the IMPLICITLY HIDDEN clause, the result of a SELECT * does not include the implicitly hidden column. However, the result of a SELECT that explicitly refers to the name of an implicitly hidden column includes that column in the result table.

IMPLICITLY HIDDEN must not be specified for all columns of the table (SQLSTATE 428GU).

lob-options
Specifies options for LOB data types.

LOGGED
Specifies that changes that are made to the column are to be written to the log. The data in such columns is then recoverable with database utilities (such as RESTORE DATABASE). LOGGED is the default.

NOT LOGGED
Specifies that changes that are made to the column are not to be logged. This only applies to LOB data that is not inlined.

NOT LOGGED has no effect on a commit or rollback operation; that is, the database's consistency is maintained even if a transaction is rolled back, regardless of whether the LOB value is logged. The implication of not logging is that during a rollforward operation, after a backup or load operation, the LOB data will be replaced by zeros for those LOB values that would have had log records replayed during the rollforward. During crash recovery, all committed changes and changes rolled back reflect the expected results.

COMPACT
Specifies that the values in the LOB column should take up minimal disk space (free any extra disk pages in the last group that is used by the LOB value), rather than leave any leftover space at the end of the LOB storage area that might facilitate subsequent append operations. Storing data in this way might reduce the performance of append (length-increasing) operations on the column.

NOT COMPACT
Specifies some space for insertions to assist in future changes to the LOB values in the column. This is the default.

SCOPE
Identifies the scope of the reference type column.

A scope must be specified for any column that is intended to be used as the left operand of a dereference operator or as the argument of the DEREF function. Specifying the scope for a reference type column can be deferred to a subsequent ALTER TABLE statement to allow the target table to be defined, usually when mutually referencing tables.

typed-table-name
The name of a typed table. The table must already exist or be the same as the name of the table that is being created (SQLSTATE 42704). The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-table-name* (SQLSTATE 428DM). No checking is done of values that are assigned to *column-name* to ensure that the values actually reference existing rows in *typed-table-name*.

typed-view-name
The name of a typed view. The view must already exist or be the same as the name of the view being created (SQLSTATE 42704). The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-view-name* (SQLSTATE 428DM). No checking is done of values that are assigned to *column-name* to ensure that the values actually reference existing rows in *typed-view-name*.

CONSTRAINT *constraint-name*
Names the constraint. A *constraint-name* must not identify a constraint that was already specified within the same CREATE TABLE statement. (SQLSTATE 42710).

If this clause is omitted, an 18 byte long identifier that is unique among the identifiers of existing constraints defined on the table is generated by the system. (The identifier consists of "SQL" followed by a sequence of 15 numeric characters that are generated by a timestamp-based function).

When used with a PRIMARY KEY or UNIQUE constraint, the *constraint-name* can be used as the name of an index that is created to support the constraint.

PRIMARY KEY
This provides a shorthand method of defining a primary key that is composed of a single column. Thus, if PRIMARY KEY is specified in the definition of column C, the effect is the same as if the PRIMARY KEY(C) clause is specified as a separate clause.

A primary key cannot be specified if the table is a subtable (SQLSTATE 429B3) because the primary key is inherited from the supertable.

A ROW CHANGE TIMESTAMP column cannot be used as part of a primary key (SQLSTATE 429BV).

Row-begin, row-end, and transaction-start-ID columns cannot be used as part of a primary key (SQLSTATE 429BV).

See PRIMARY KEY within the *unique-constraint* description.

UNIQUE
This provides a shorthand method of defining a unique key that is composed of a single column. Thus, if UNIQUE is specified in the definition of column C, the effect is the same as if the UNIQUE(C) clause is specified as a separate clause.

A unique constraint cannot be specified if the table is a subtable (SQLSTATE 429B3) since unique constraints are inherited from the supertable.

See UNIQUE within the *unique-constraint* description.

references-clause
This provides a shorthand method of defining a foreign key that is composed of a single column. Thus, if a references-clause is specified in the definition of column C, the effect is the same as if that references-clause were specified as part of a FOREIGN KEY clause in which C is the only identified column.

See *references-clause* under *referential-constraint* description.

CHECK (*check-condition*)
This provides a shorthand method of defining a check constraint that applies to a single column. See description for CHECK (*check-condition*).

default-clause
Specifies a default value for the column.

WITH
An optional keyword.

DEFAULT

Provides a default value if a value is not supplied on insert or is specified as DEFAULT on INSERT or UPDATE. If a default value is not specified following the DEFAULT keyword, the default value depends on the data type of the column as shown in "ALTER TABLE". This clause must not be specified with generated-clause in a column definition (SQLSTATE 42614).

If a column is defined as XML, a default value cannot be specified (SQLSTATE 42613). The only possible default is NULL.

If the column is based on a column of a typed table, a specific default value must be specified when defining a default. A default value cannot be specified for the object identifier column of a typed table (SQLSTATE 42997).

If a column is defined by using a distinct type, then the default value of the column is the default value of the source data type cast to the distinct type.

If a column is defined by using a structured type, the *default-clause* cannot be specified (SQLSTATE 42842).

Omission of DEFAULT from a *column-definition* results in the use of the null value as the default for the column. If such a column is defined NOT NULL, then the column does not have a valid default.

default-values
Specific types of default values that can be specified are as follows.

constant
Specifies the constant as the default value for the column. The specified constant must:

- Represent a value that might be assigned to the column in accordance with the rules of assignment.
- Not be a floating-point constant unless the column is defined with a floating-point data type.
- Be a numeric constant or a decimal floating-point special value if the data type of the column is a decimal floating-point. Floating-point constants are first interpreted as DOUBLE and then converted to decimal floating-point if the target column is DECFLOAT. For DECFLOAT(16) columns, decimal constants having precision greater than 16 digits are rounded by using the rounding modes specified by the CURRENT DECFLOAT ROUNDING MODE special register.
- Not have nonzero digits beyond the scale of the column data type if the constant is a decimal constant (for example, 1.234 cannot be the default for a DECIMAL(5,2) column).
- Be expressed with no more than 254 bytes including the quotation mark characters, any introducer character such as the X for a hexadecimal constant, and characters from the fully qualified function name and parentheses when the constant is the argument of a *cast-function*

datetime-special-register
Specifies the value of the datetime special register (CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP) at the time of INSERT, UPDATE, or LOAD as the default for the column. The data type of the column must be the data type that corresponds to the special register specified (for example, data type must be DATE when CURRENT DATE is specified).

user-special-register
Specifies the value of the user special register (CURRENT USER, SESSION_USER, SYSTEM_USER) at the time of INSERT, UPDATE, or LOAD as the default for the column. The data type of the column must be a character string with a length not less than the length attribute of a user special register. USER can be specified in place of SESSION_USER and CURRENT_USER can be specified in place of CURRENT USER.

CURRENT SCHEMA
Specifies the value of the CURRENT SCHEMA special register at the time of INSERT, UPDATE, or LOAD as the default for the column. If CURRENT SCHEMA is specified, the data type of the column must be a character string with a length greater than or equal to the length attribute of the CURRENT SCHEMA special register.

CURRENT MEMBER
Specifies the value of the CURRENT MEMBER special register at the time of INSERT, UPDATE, or LOAD as the default for the column. If CURRENT MEMBER is specified, the data type of the column must allow assignment from an integer.

NULL
Specifies NULL as the default for the column. If NOT NULL was specified, DEFAULT NULL can be specified within the same column definition but results in an error on any attempt to set the column to the default value.

cast-function
This form of a default value can only be used with columns defined as a distinct type, BLOB, or datetime (DATE, TIME, or TIMESTAMP) data type. For distinct type, except

for distinct types based on BLOB or datetime types, the name of the function must match the name of the distinct type for the column. If qualified with a schema name, it must be the same as the schema name for the distinct type. If not qualified, the schema name from function resolution must be the same as the schema name for the distinct type. For a distinct type based on a datetime type, where the default value is a constant, a function must be used and the name of the function must match the name of the source type of the distinct type with an implicit or explicit schema name of SYSIBM. For other datetime columns, the corresponding datetime function can also be used. For a BLOB or a distinct type based on BLOB, a function must be used and the name of the function must be BLOB with an implicit or explicit schema name of SYSIBM.

constant
Specifies a constant as the argument. The constant must conform to the rules of a constant for the source type of the distinct type or for the data type if not a distinct type. If the *cast-function* is BLOB, the constant must be a string constant.

datetime-special-register
Specifies CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP. The source type of the distinct type of the column must be the data type that corresponds to the specified special register.

user-special-register
Specifies CURRENT USER, SESSION_USER, or SYSTEM_USER. The data type of the source type of the distinct type of the column must be a string data type with a length of at least 8 bytes. If the *cast-function* is BLOB, the length attribute must be at least 8 bytes.

CURRENT SCHEMA
Specifies the value of the CURRENT SCHEMA special register. The data type of the source type of the distinct type of the column must be a character string with a length greater than or equal to the length attribute of the CURRENT SCHEMA special register. If the cast-function is BLOB, the length attribute must be at least 8 bytes.

EMPTY_CLOB(), EMPTY_DBCLOB(), or EMPTY_BLOB()
Specifies a zero-length string as the default for the column. The column must have the data type that corresponds to the result data type of the function.

If the value specified is not valid, an error is returned (SQLSTATE 42894).

generated-clause
Specifies a generated value for the column.

GENERATED

Specifies that the database generates values for the column. GENERATED must be specified if the column is to be considered an identity column or a row change time stamp column, row-begin column, row-end column, transaction-start-ID column, or generated expression column. A default clause must not be specified for a column that is defined as GENERATED (SQLSTATE 42623).

ALWAYS
Specifies that a value is always generated for the column when a row is inserted into the table, or whenever the result value of the *generation-expression* changes. The result of the expression is stored in the table. GENERATED ALWAYS is the recommended value unless data propagation or unload and reload operations are being done. GENERATED ALWAYS is the required value for generated columns.

BY DEFAULT
Specifies that the database generates a value for the column when a row is inserted, or updated specifying the DEFAULT clause, unless an explicit value is specified. BY DEFAULT is the recommended value when using data propagation or performing an unload and reload operation.

Although not explicitly required, to ensure uniqueness of the values, define a unique single-column index on generated IDENTITY columns.

AS IDENTITY
Specifies that the column is to be the identity column for this table. A table can only have a single identity column (SQLSTATE 428C1). The IDENTITY keyword can only be specified if the data type associated with the column is an exact numeric type with a scale of zero, or a user-defined distinct type for which the source type is an exact numeric type with a scale of zero (SQLSTATE 42815). SMALLINT, INTEGER, BIGINT, or DECIMAL with a scale of zero, or a distinct type based on one of these types, are considered exact numeric types. By contrast, single- and double-precision floating points are considered approximate numeric data types. Reference types, even if represented by an exact numeric type, cannot be defined as identity columns.

An identity column is implicitly NOT NULL. An identity column cannot have a DEFAULT clause (SQLSTATE 42623).

START WITH *numeric-constant*
Specifies the first value for the identity column. This value can be any positive or negative value that might be assigned to this column (SQLSTATE 42815), without nonzero digits existing to the right of the decimal point (SQLSTATE 428FA). The default is MINVALUE for ascending sequences, and MAXVALUE for descending sequences. This value is not necessarily the value that would be cycled to after reaching the maximum or minimum value for the identity column. The START WITH clause can be used to start the generation of values outside the range that is used for cycles. The range that is used for cycles is defined by MINVALUE and MAXVALUE.

INCREMENT BY *numeric-constant*
Specifies the interval between consecutive values of the identity column. This value can be any positive or negative value that might be assigned to this column (SQLSTATE 42815), and does not exceed the value of a large integer constant (SQLSTATE 42820), without nonzero digits existing to the right of the decimal point (SQLSTATE 428FA).

If this value is negative, this is a descending sequence. If this value is 0, or positive, this is an ascending sequence. The default is 1.

NO MINVALUE or MINVALUE
Specifies the minimum value at which a descending identity column either cycles or stops generating values, or an ascending identity column cycles to after reaching the maximum value.

NO MINVALUE
For an ascending sequence, the value is the START WITH value, or 1 if START WITH was not specified. For a descending sequence, the value is the minimum value of the data type of the column. This is the default.

MINVALUE *numeric-constant*
Specifies the numeric constant that is the minimum value. This value can be any positive or negative value that might be assigned to this column (SQLSTATE 42815), without nonzero digits existing to the right of the decimal point (SQLSTATE 428FA), but the value must be less than or equal to the maximum value (SQLSTATE 42815).

NO MAXVALUE or MAXVALUE
Specifies the maximum value at which an ascending identity column either cycles or stops generating values, or a descending identity column cycles to after reaching the minimum value.

NO MAXVALUE
For an ascending sequence, the value is the maximum value of the data type of the column. For a descending sequence, the value is the START WITH value, or -1 if START WITH was not specified. This is the default.

MAXVALUE *numeric-constant*
Specifies the numeric constant that is the maximum value. This value can be any positive or negative value that might be assigned to this column (SQLSTATE 42815), without nonzero digits existing to the right of the decimal point (SQLSTATE 428FA), but the value must be greater than or equal to the minimum value (SQLSTATE 42815).

NO CYCLE or CYCLE
Specifies whether this identity column should continue to generate values after generating either its maximum or minimum value.

NO CYCLE
Specifies that values are not generated for the identity column after the maximum or minimum value is reached. This is the default.

CYCLE
Specifies that values continue to be generated for this column after the maximum or minimum value is reached. If this option is used, after an ascending identity column reaches the maximum value, it generates its minimum value; or after a descending sequence reaches the minimum value, it generates its maximum value. The maximum and minimum values for the identity column determine the range that is used for cycling.

When CYCLE is in effect, duplicate values might be generated for an identity column. Although not explicitly required, a unique, single-column index should be defined on the generated column to ensure uniqueness of the values, if unique values are required. If a unique index exists on such an identity column and a non-unique value is generated, an error occurs (SQLSTATE 23505).

NO CACHE or CACHE
Specifies whether to keep some pre-allocated values in memory for faster access. If a new value is needed for the identity column, and none is available in the cache, then the end of the new cache block must be logged. However, when a new value is needed for the identity column, and an unused value exists in the cache, then the allocation of that identity value is faster, because no logging is necessary. This is a performance and tuning option.

NO CACHE
Specifies that values for the identity column are not to be pre-allocated.

When this option is specified, the values of the identity column are not stored in the cache. In this case, every request for a new identity value results in synchronous I/O to the log.

CACHE *integer-constant*
Specifies how many values of the identity sequence are to be pre-allocated and kept in memory. When values are generated for the identity column, pre-allocating and storing values in the cache reduces synchronous I/O to the log.

If a new value is needed for the identity column and no unused values are available in the cache, the allocation of the value involves waiting for I/O to the log. However, when a new value is needed for the identity column and an unused value exists in the cache, the allocation of that identity value can happen more quickly by avoiding the I/O to the log.

The minimum value is 2 (SQLSTATE 42815). The default value is CACHE 20.

Use the CACHE and NO ORDER options to allow multiple caches of identity values simultaneously. In a multi-partition or Db2 pureScale environment, multiple members can cache them.

In a Db2 pureScale environment, if both CACHE and ORDER are specified, the specification of ORDER overrides the specification of CACHE and instead NO CACHE will be in effect.

NO ORDER or ORDER
Specifies whether the identity values must be generated in order of request.

NO ORDER
Specifies that the values do not need to be generated in order of request. This is the default.

ORDER
Specifies that the values must be generated in order of request.

FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP
Specifies that the column is a time stamp column for the table. A value is generated for the column in each row that is inserted, and for any row in which any column is updated. The value that is generated for a ROW CHANGE TIMESTAMP column is a time stamp that corresponds to the insert or update time for that row. If multiple rows are inserted or updated with a single statement, the value of the ROW CHANGE TIMESTAMP column might be different for each row.

A table can only have one ROW CHANGE TIMESTAMP column (SQLSTATE 428C1). If *data-type* is specified, it must be TIMESTAMP or TIMESTAMP(6) (SQLSTATE 42842). A ROW CHANGE TIMESTAMP column cannot have a DEFAULT clause (SQLSTATE 42623). NOT NULL must be specified for a ROW CHANGE TIMESTAMP column (SQLSTATE 42831).

AS (*generation-expression*)
Specifies that the definition of the column is based on an expression. (If the expression for a GENERATED ALWAYS column includes a user-defined external function, changing the executable for the function (such that the results change for given arguments) can result in inconsistent data. This can be avoided by using the SET INTEGRITY statement to force the generation of new values). The *generation-expression* cannot contain any of the following (SQLSTATE 42621):

- Subqueries
- XMLQUERY or XMLEXISTS expressions
- Column functions
- Dereference operations or DEREF functions
- User-defined or built-in functions that are non-deterministic
- User-defined functions that use the EXTERNAL ACTION option
- User-defined functions that are not defined with NO SQL
- Host variables or parameter markers
- Special registers and built-in functions that depend on the value of a special register
- Global variables
- References to columns defined later in the column list
- References to other generated columns
- References to columns of type XML

The data type for the column is based on the result data type of the *generation-expression*. A CAST specification can be used to force a particular data type and to provide a scope (for a reference type only). If *data-type* is specified, values are assigned to the column according to the appropriate assignment rules. A generated column is considered to be nullable unless the NOT NULL column option is specified. The data type of a generated column and the result data type of the *generation-expression* must have equality defined (see "Assignments and comparisons"). This excludes columns and generation expressions of type LOB data types, XML, structured types, and distinct types based on any of these types (SQLSTATE 42962).

AS ROW BEGIN

Specifies that the generated value is assigned by the database manager whenever a row is inserted into the table or any column in the row is updated. The value is generated by using a reading from the time-of-day clock during execution of the first data change statement in the transaction that requires a value to be assigned to the row-begin column or transaction-start-ID column in the table, or a row in a system-period temporal table is deleted.

For a system-period temporal table, the database manager ensures uniqueness of the generated values for a row-begin column across transactions. The time stamp value might be adjusted to ensure that rows that are inserted into an associated history table have the end time stamp value greater than the begin time stamp value. This can happen when a conflicting transaction is updating the same row in the system-period temporal table. The database configuration parameter **systime_period_adj** must be set to Yes for this adjustment to the time stamp value to occur. If multiple rows are inserted or updated within a single SQL transaction and an adjustment is not needed, the values for the row-begin column are the same for all the rows and are unique from the values that are generated for the column for another transaction. A row-begin column is required as the begin column of a SYSTEM_TIME period, which is the intended use for this type of generated column.

A table can have only one row-begin column (SQLSTATE 428C1). If *data-type* is not specified the column is defined as a TIMESTAMP(12). If *data-type* is specified, it must be TIMESTAMP(12) (SQLSTATE 42842). The column cannot have a DEFAULT clause (SQLSTATE 42623), and must be defined as NOT NULL (SQLSTATE 42831). A row-begin column is not updatable.

AS ROW END

Specifies that a value for the data type of the column is assigned by the database manager whenever a row is inserted or any column in the row is updated. The assigned value is TIMESTAMP "9999-12-30-00.00.00.000000000000".

A row-end column is required as the second column of a SYSTEM_TIME period, which is the intended use for this type of generated column.

A table can have only one row-end column (SQLSTATE 428C1). If *data-type* is not specified, the column is defined as TIMESTAMP(12). If *data-type* is specified, it must be TIMESTAMP(12) (SQLSTATE 42842). The column cannot have a DEFAULT clause (SQLSTATE 42623), and must be defined as NOT NULL (SQLSTATE 42831). A row-end column is not updatable.

AS TRANSACTION START ID

Specifies that the value is assigned by the database manager whenever a row is inserted into the table or any column in the row is updated. The database manager assigns a unique time stamp value per transaction or the null value. The null value is assigned to the transaction-start-ID column if the column is nullable and if there is a row-begin column in the table for which the value did not need to be adjusted. Otherwise, the value is generated by using a reading of the time-of-day clock during execution of the first data change statement in the transaction that requires a value to be assigned to a row-begin column or transaction-start-ID column in the table, or a row in a system-period temporal table is deleted. If multiple rows are inserted or updated within a single SQL transaction, the values for the transaction-start-ID column are the same for all the rows and are unique from the values that are generated for the column for another transaction.

A transaction-start-ID column is required for a system-period temporal table, which is the intended use for this type of generated column.

A table can have only one transaction-start-ID column (SQLSTATE 428C1). If *data-type* is not specified, the column is defined as TIMESTAMP(12). If *data-type* is specified, it

must be TIMESTAMP(12). A transaction-start-ID column cannot have a DEFAULT clause (SQLSTATE 42623). A transaction-start-ID column is not updatable.

INLINE LENGTH *integer*

This option is valid only for a column that is defined using a structured type, XML, or LOB data type (SQLSTATE 42842).

For a column of data type XML or LOB, *integer* indicates the maximum byte size of the internal representation of an XML document or LOB data to store in the base table row. XML documents that have a larger internal representation are stored separately from the base table row in an auxiliary storage object. This takes place automatically. There is no default inline length for XML type columns. If the XML document or LOB data is stored inlined in the base table row, there is an additional overhead. For LOB data, the overhead is 4 bytes.

For a column of data type LOB, the default inline length is set to be the maximum size of the LOB descriptor if the clause is not specified. Any explicit INLINE LENGTH must be at least the maximum LOB descriptor size. The following table summarizes the LOB descriptor sizes.

Table 137. Sizes of the LOB descriptor for various LOB lengths.

Maximum LOB length in bytes	Minimum explicit INLINE LENGTH
1,024	68
8,192	92
65,536	116
524,000	140
4,190,000	164
134,000,000	196
536,000,000	220
1,070,000,000	252
1,470,000,000	276
2,147,483,647	312

For a structured type column, *integer* indicates the maximum size in bytes of an instance of a structured type to store inline with the rest of the values in the row. Instances of structured types that cannot be stored inline are stored separately from the base table row, similar to the way that LOB values are stored. This takes place automatically. The default INLINE LENGTH for a structured-type column is the inline length of its type (specified explicitly or by default in the CREATE TYPE statement). If INLINE LENGTH of the structured type is less than 292, the value 292 is used for the INLINE LENGTH of the column.

Note: The inline lengths of subtypes are not counted in the default inline length, meaning that instances of subtypes might not fit inline unless an explicit INLINE LENGTH is specified at CREATE TABLE time to account for existing and future subtypes.

The explicit INLINE LENGTH value cannot exceed 32 673. For a structured type or XML data type, it must be at least 292 (SQLSTATE 54010).

COMPRESS SYSTEM DEFAULT

Specifies that system default values are to be stored using minimal space. If the VALUE COMPRESSION clause is not specified, a warning is returned (SQLSTATE 01648), and system default values are not stored using minimal space.

Allowing system default values to be stored in this manner causes a slight performance penalty during insert and update operations on the column because of extra checking that is done.

The base data type must not be a DATE, TIME, TIMESTAMP, XML, or structured data type (SQLSTATE 42842). If the base data type is a varying-length string, this clause is ignored. String values of length 0 are automatically compressed if a table has been set with VALUE COMPRESSION.

COLUMN SECURED WITH *security-label-name*
Identifies a security label that exists for the security policy that is associated with the table. The name must not be qualified (SQLSTATE 42601). The table must have a security policy associated with it (SQLSTATE 55064). The table must not be a system-period temporal table.

Generally, you are not allowed to protect data in such a way that your current LBAC credentials do not allow you to write to that data. To protect a column with a particular security label, you must have LBAC credentials that allow you to write to data protected by that security label. You do not have to have SECADM authority.

period-definition

PERIOD
Defines a period for the table.

SYSTEM_TIME (*begin-column-name*, *end-column-name*)

Defines a system period with the name SYSTEM_TIME. There must not be a column in the table with the name SYSTEM_TIME (SQLSTATE 42711). A table can have only one SYSTEM_TIME period (SQLSTATE 42711). *begin-column-name* must be defined as ROW BEGIN and *end-column-name* must be defined as ROW END (SQLSTATE 428HN).

BUSINESS_TIME (*begin-column-name*, *end-column-name*)

Defines an application period with the name BUSINESS_TIME. There must not be a column in the table with the name BUSINESS_TIME (SQLSTATE 42711). A table can have only one BUSINESS_TIME period (SQLSTATE 42711). *begin-column-name* and *end-column-name* must both be defined as DATE or TIMESTAMP(*p*) where *p* is in the range 0 - 12 (SQLSTATE 42842), and the columns must be defined as NOT NULL (SQLSTATE 42831). *begin-column-name* and *end-column-name* must not identify a column that is defined with a GENERATED clause (SQLSTATE 428HZ).

An implicit check constraint is generated to ensure that the value of *end-column-name* is greater than the value of *begin-column-name*. The name of the implicitly created check constraint is DB2_GENERATED_CHECK_CONSTRAINT_FOR_BUSINESS_TIME and must not be the name of any other check constraint that is specified in the statement (SQLSTATE 42710).

unique-constraint
Defines a unique or primary key constraint. If the table has a distribution key, any unique or primary key must be a superset of the distribution key. A unique or primary key constraint cannot be specified for a table that is a subtable (SQLSTATE 429B3). Primary or unique keys cannot be subsets of dimensions (SQLSTATE 429BE). If the table is a root table, the constraint applies to the table and all its subtables.

CONSTRAINT *constraint-name*
Names the primary key or unique constraint.

UNIQUE (*column-name*, ...)
Defines a unique key that is composed of the identified columns. The identified columns must be defined as NOT NULL. Each *column-name* must identify a column of the table and the same column must not be identified more than once.

If the table has a BUSINESS_TIME period defined, BUSINESS_TIME WITHOUT OVERLAPS can be specified as the last item in the key expression list. If BUSINESS_TIME WITHOUT OVERLAPS is specified, the list must include at least one *column-name*. WITHOUT OVERLAPS means that for the other specified keys, the values are unique with respect to time for the BUSINESS_TIME period. When BUSINESS_TIME WITHOUT OVERLAPS is specified, the columns of the BUSINESS_TIME period must not be specified as part of the constraint (SQLSTATE 428HW).

The specification of BUSINESS_TIME WITHOUT OVERLAPS adds the following attributes to the constraint:

- The end column of the BUSINESS_TIME period in ascending order
- The begin column of the BUSINESS_TIME period in ascending order

The number of identified columns must not exceed 64, and the sum of their stored lengths must not exceed the index key length limit for the page size. For column stored lengths, see Byte Counts. For key length limits, see "SQL limits". No LOB, XML, distinct type based on one of these types, or structured type can be used as part of a unique key, even if the length attribute of the column is small enough to fit within the index key length limit for the page size (SQLSTATE 54008).

The set of columns in the unique key cannot be the same as the set of columns in the primary key or another unique key (SQLSTATE 01543). (If LANGLEVEL is SQL92E or MIA, an error is returned, SQLSTATE 42891).

A unique constraint cannot be specified if the table is a subtable (SQLSTATE 429B3)because unique constraints are inherited from the supertable.

The description of the table as recorded in the catalog includes the unique key and, if enforced, its unique index. If enforced, a unique bidirectional index, which allows forward and reverse scans, is automatically created for the columns in the sequence that are specified with ascending order for each column. The name of the index is the same as the *constraint-name* if this does not conflict with an existing index in the schema where the table is created. If the index name conflicts, the name is SQL, followed by a character time stamp (*yymmddhhmmssxxx*), with SYSIBM as the schema name.

PRIMARY KEY (*column-name,...*)

Defines a primary key that is composed of the identified columns. The clause must not be specified more than once, and the identified columns must be defined as NOT NULL. Each *column-name* must identify a column of the table, and the same column must not be identified more than once.

If the table has a BUSINESS_TIME period defined, BUSINESS_TIME WITHOUT OVERLAPS can be specified as the last item in the key expression list. If BUSINESS_TIME WITHOUT OVERLAPS is specified, the list must include at least one *column-name*. WITHOUT OVERLAPS means that for the rest of the specified keys, the values are unique with respect to time for the BUSINESS_TIME period. When BUSINESS_TIME WITHOUT OVERLAPS is specified, the columns of the BUSINESS_TIME period must not be specified as part of the constraint (SQLSTATE 428HW). The specification of BUSINESS_TIME WITHOUT OVERLAPS adds the following attributes to the constraint:

- The end column of the BUSINESS_TIME period in ascending order
- The begin column of the BUSINESS_TIME period in ascending order

The number of identified columns must not exceed 64, and the sum of their stored lengths must not exceed the index key length limit for the page size. For column stored lengths, see Byte Counts. For key length limits, see "SQL limits". No LOB, XML, distinct type based on one of these types, or structured type can be used as part of a primary key, even if the length attribute of the column is small enough to fit within the index key length limit for the page size (SQLSTATE 54008).

The set of columns in the primary key cannot be the same as the set of columns in a unique key (SQLSTATE 01543). (If LANGLEVEL is SQL92E or MIA, an error is returned, SQLSTATE 42891).

Only one primary key can be defined on a table.

A primary key cannot be specified if the table is a subtable (SQLSTATE 429B3) because the primary key is inherited from the supertable.

The description of the table as recorded in the catalog includes the primary key and, if enforced, its primary index. If enforced, a unique bidirectional index, which allows forward and reverse scans, will automatically be created for the columns in the sequence specified with ascending order for each column. The name of the index is the same as the *constraint-name* if this does

not conflict with an existing index in the schema where the table is created. If the index name conflicts, the name is SQL, followed by a character time stamp (*yymmddhhmmssxxx*), with SYSIBM as the schema name.

When explicitly defining distribution keys using the **DISTRIBUTE BY HASH** clause, the columns of a *unique-constraint* must be a superset of the distribution key columns; column order is unimportant. When distribution keys are implicitly defined, they are selected based on the definition of the unique constraint. Implicit selection of distribution keys occurs in the following cases:

- Omit **DISTRIBUTE BY HASH** clause and the table is defined in a database partition group with multiple partitions.
- **DISTRIBUTE BY RANDOM** clause is used.

referential-constraint
Defines a referential constraint.

CONSTRAINT *constraint-name*
Names the referential constraint.

FOREIGN KEY (*column-name*,...)
Defines a referential constraint with the specified *constraint-name*.

Let T1 denote the object table of the statement. The foreign key of the referential constraint is composed of the identified columns. Each name in the list of column names must identify a column of T1 and the same column must not be identified more than once.

The number of identified columns must not exceed 64, and the sum of their stored lengths must not exceed the index key length limit for the page size. For column stored lengths, see Byte Counts. For key length limits, see "SQL limits". No LOB, XML, distinct type based on one of these types, or structured type column can be used as part of a foreign key (SQLSTATE 42962). There must be the same number of foreign key columns as there are in the parent key and the data types of the corresponding columns must be compatible (SQLSTATE 42830). Two-column descriptions are compatible if they have compatible data types (both columns are numeric, character strings, graphic, date/time, or have the same distinct type).

references-clause
Specifies the parent table or the parent nickname, and the parent key for the referential constraint.

REFERENCES *table-name* or *nickname*
The table or nickname that is specified in a REFERENCES clause must identify a base table or nickname that is described in the catalog, but must not identify a catalog table.

A referential constraint is a duplicate if its foreign key, parent key, and parent table or parent nickname are the same as the foreign key, parent key, and parent table or parent nickname of a previously specified referential constraint. Duplicate referential constraints are ignored, and a warning is returned (SQLSTATE 01543).

In the following discussion, let T2 denote the identified parent table, and let T1 denote the table that is being created (or altered). (T1 and T2 can be the same table).

The specified foreign key must have the same number of columns as the parent key of T2 and the description of the *n*th column of the foreign key must be comparable to the description of the *n*th column of that parent key. Datetime columns are not considered to be comparable to string columns for the purposes of this rule.

(*column-name*,...)
The parent key of a referential constraint is composed of the identified columns. Each *column-name* must be an unqualified name that identifies a column of T2. The same column must not be identified more than once.

The list of column names must match the set of columns (in any order) of the primary key or a unique constraint that exists on T2 (SQLSTATE 42890). If a column name list is not specified, then T2 must have a primary key (SQLSTATE 42888). Omission of the column

name list is an implicit specification of the columns of that primary key in the sequence originally specified.

The referential constraint that is specified by a FOREIGN KEY clause defines a relationship in which T2 is the parent and T1 is the dependent.

rule-clause

Specifies what action to take on dependent tables.

ON DELETE

Specifies what action is to take place on the dependent tables when a row of the parent table is deleted. There are four possible actions:

- NO ACTION (default)
- RESTRICT
- CASCADE
- SET NULL

The delete rule applies when a row of T2 is the object of a DELETE or propagated delete operation and that row has dependents in T1. Let *p* denote such a row of T2.

- If RESTRICT or NO ACTION is specified, an error occurs and no rows are deleted.
- If CASCADE is specified, the delete operation is propagated to the dependents of *p* in T1.
- If SET NULL is specified, each nullable column of the foreign key of each dependent of *p* in T1 is set to null.

SET NULL must not be specified unless some column of the foreign key allows null values. Omission of the clause is an implicit specification of ON DELETE NO ACTION.

If T1 is delete-connected to T2 through multiple paths, defining two SET NULL rules with overlapping foreign key definitions is not allowed. For example: T1 (i1, i2, i3). Rule1 with foreign key (i1, i2) and Rule2 with foreign key (i2, i3) is not allowed.

The firing order of the rules is:

1. RESTRICT
2. SET NULL OR CASCADE
3. NO ACTION

If any row in T1 is affected by two different rules, error occurs and no rows are deleted.

A referential constraint cannot be defined if it would cause a table to be delete-connected to itself by a cycle involving two or more tables, and where one of the delete rules is RESTRICT or SET NULL (SQLSTATE 42915).

A referential constraint that would cause a table to be delete-connected to either itself or another table by multiple paths can be defined, except in the following cases (SQLSTATE 42915):

- A table must not be both a dependent table in a CASCADE relationship (self-referencing, or referencing another table), and have a self-referencing relationship in which the delete rule is RESTRICT or SET NULL.
- A key overlaps another key when at least one column in one key is the same as a column in the other key. When a table is delete-connected to another table through multiple relationships with overlapping foreign keys, those relationships must have the same delete rule, and none of the delete rules can be SET NULL.
- When a table is delete-connected to another table through multiple relationships, and at least one of those relationships is specified with a delete rule of SET NULL, the foreign key definitions of these relationships must not contain any distribution key or multidimensional clustering (MDC) key column.

- When two tables are delete-connected to the same table through CASCADE relationships, the two tables must not be delete-connected to each other if the delete rule of the last relationship in each delete-connected path is RESTRICT or SET NULL.

If any row in T1 is affected by different delete rules, the result would be the effect of all the actions that are specified by these rules. AFTER triggers and CHECK constraints on T1 will also see the effect of all the actions. An example of this is a row that is targeted to be set null through one delete-connected path to an ancestor table, and targeted to be deleted by a second delete-connected path to the same ancestor table. The result would be the deletion of the row. AFTER DELETE triggers on this descendant table would be activated, but AFTER UPDATE triggers would not.

In applying the previously mentioned rules to referential constraints, in which either the parent table or the dependent table is a member of a typed table hierarchy, all the referential constraints that apply to any table in the respective hierarchies are considered.

ON UPDATE
Specifies what action is to take place on the dependent tables when a row of the parent table is updated. The clause is optional. ON UPDATE NO ACTION is the default and ON UPDATE RESTRICT is the only alternative.

The difference between NO ACTION and RESTRICT is described in the "Notes" section.

check-constraint
Defines a check constraint. A *check-constraint* is a *search-condition* that must evaluate to not false or a functional dependency that is defined between columns.

CONSTRAINT *constraint-name*
Names the check constraint.

CHECK (*check-condition*)
Defines a check constraint. The *search-condition* must be true or unknown for every row of the table.

search-condition
The *search-condition* has the following restrictions:

- A column reference must be to a column of the table that is being created.
- The *search-condition* cannot contain a TYPE predicate.
- The *search-condition* cannot contain any of the following (SQLSTATE 42621):
  - Subqueries
  - XMLQUERY or XMLEXISTS expressions
  - Dereference operations or DEREF functions where the scoped reference argument is other than the object identifier (OID) column
  - CAST specifications with a SCOPE clause
  - Column functions
  - Functions that are not deterministic
  - Functions that are defined to have an external action
  - User-defined functions that are defined with either MODIFIES SQL or READS SQL DATA
  - Host variables
  - Parameter markers
  - *sequence-references*
  - OLAP specifications
  - Special registers and built-in functions that depend on the value of a special register
  - Global variables
  - References to generated columns other than the identity column

- References to columns of type XML (except in a VALIDATED predicate)
- An error tolerant *nested-table-expression*

functional-dependency
Defines a functional dependency between columns.

***column-name* DETERMINED BY *column-name* or (*column-name*,...) DETERMINED BY (*column-name*,...)**
The parent set of columns contains the identified columns that immediately precede the DETERMINED BY clause. The child set of columns contains the identified columns that immediately follow the DETERMINED BY clause. All of the restrictions on the *search-condition* apply to parent set and child set columns, and only simple column references are allowed in the set of columns (SQLSTATE 42621). The same column must not be identified more than once in the functional dependency (SQLSTATE 42709). The data type of the column must not be a LOB data type, a distinct type based on a LOB data type, an XML data type, or a structured type (SQLSTATE 42962). A ROW CHANGE TIMESTAMP column cannot be used as part of a primary key (SQLSTATE 429BV). No column in the child set of columns can be a nullable column (SQLSTATE 42621).

If a check constraint is specified as part of a *column-definition*, a column reference can only be made to the same column. Check constraints that are specified as part of a table definition can have column references identifying columns that are previously defined in the CREATE TABLE statement. Check constraints are not checked for inconsistencies, duplicate conditions, or equivalent conditions. Therefore, contradictory or redundant check constraints can be defined, resulting in possible errors at execution time.

The *search-condition* "IS NOT NULL" can be specified; however, it is recommended that nullability is enforced directly, by using the NOT NULL attribute of a column. For example, `CHECK (salary + bonus > 30000)` is accepted if salary is set to NULL, because CHECK constraints must be either satisfied or unknown, and in this case, salary is unknown. However, `CHECK (salary IS NOT NULL)` would be considered false and a violation of the constraint if salary is set to NULL.

Check constraints with *search-condition* are enforced when rows in the table are inserted or updated. A check constraint that is defined on a table automatically applies to all subtables of that table.

A functional dependency is not enforced by the database manager during normal operations such as insert, update, delete, or set integrity. The functional dependency might be used during query rewrite to optimize queries. Incorrect results might be returned if the integrity of a functional dependency is not maintained.

constraint-attributes
Defines attributes that are associated with primary key, unique, referential integrity, or check constraints.

ENFORCED or NOT ENFORCED
Specifies whether the constraint is enforced by the database manager during normal operations such as insert, update, or delete. The default is determined by the setting of the **ddl_constraint_def** configuration parameter. You can override the default behavior by specifying either ENFORCED or NOT ENFORCED explicitly.

ENFORCED
The constraint is enforced by the database manager. ENFORCED cannot be specified in the following situations:

- For a functional dependency (SQLSTATE 42621)
- When a referential constraint refers to a nickname (SQLSTATE 428G7)

NOT ENFORCED
The constraint is not enforced by the database manager. A primary key constraint or unique constraint cannot be NOT ENFORCED if a dependent ENFORCED referential constraint exists.

TRUSTED
The data can be trusted to conform to the constraint. TRUSTED must be used only if the data in the table is independently known to conform to the constraint. Query results might be unpredictable if the data does not conform to the constraint. This is the default option.

Informational constraints must not be violated at any time. Informational constraints are used in query optimization, as well as the incremental processing of REFRESH IMMEDIATE MQT and staging tables. These processes might produce unpredictable results or incorrect MQT and staging table content if the constraints are violated. For example, the order in which parent-child tables are maintained is important. When you want to add rows to a parent-child table, you must insert rows into the parent table first. To remove rows from a parent-child table, you must delete rows from the child table first. This ensures that no orphan rows exist in the child table at any time. If informational constraints are violated, the incremental maintenance of dependent MQT data and staging table data might be optimized based on the violated informational constraints, producing incorrect data.

NOT TRUSTED
The data cannot be trusted to conform to the constraint. NOT TRUSTED is intended for cases where the data conforms to the constraint for most rows, but it is not independently known that all the rows or future additions will conform to the constraint. If a constraint is NOT TRUSTED and enabled for query optimization, then it will not be used to perform optimizations that depend on the data conforming completely to the constraint. NOT TRUSTED can be specified only for referential integrity constraints (SQLSTATE 42613).

ENABLE QUERY OPTIMIZATION or DISABLE QUERY OPTIMIZATION
Specifies whether the constraint or functional dependency can be used for query optimization under appropriate circumstances. The default is ENABLE QUERY OPTIMIZATION.

ENABLE QUERY OPTIMIZATION
The constraint is assumed to be true and can be used for query optimization.

DISABLE QUERY OPTIMIZATION
The constraint cannot be used for query optimization. DISABLE QUERY OPTIMIZATION cannot be specified for primary key and unique constraints (SQLSTATE 42613).

OF *type-name1*
Specifies that the columns of the table are based on the attributes of the structured type that is identified by *type-name1*. If *type-name1* is specified without a schema name, the type name is resolved by searching the schemas on the SQL path (defined by the FUNCPATH preprocessing option for static SQL and by the CURRENT PATH register for dynamic SQL). The type name must be the name of an existing user-defined type (SQLSTATE 42704) and it must be an instantiable structured type (SQLSTATE 428DP) with at least one attribute (SQLSTATE 42997).

If UNDER is not specified, an object identifier column must be specified (refer to the *OID-column-definition*). This object identifier column is the first column of the table. The object ID column is followed by columns based on the attributes of *type-name1*.

HIERARCHY *hierarchy-name*
Names the hierarchy table that is associated with the table hierarchy. It is created at the same time as the root table of the hierarchy. The data for all subtables in the typed table hierarchy is stored in the hierarchy table. A hierarchy table cannot be directly referenced in SQL statements. A *hierarchy-name* is a *table-name*. The *hierarchy-name*, including the implicit or explicit schema name, must not identify a table, nickname, view, or alias described in the catalog. If the schema name is specified, it must be the same as the schema name of the table that is being created (SQLSTATE 428DQ). If this clause is omitted when defining the root table, a name is generated by the system. This name consists of the name of the table that is being created, followed by a unique suffix, such that the identifier is unique among the identifiers of existing tables, views, and nicknames.

UNDER *supertable-name*
Indicates that the table is a subtable of *supertable-name*. The supertable must be an existing table (SQLSTATE 42704) and the table must be defined by using a structured type that is the immediate supertype of *type-name1* (SQLSTATE 428DB). The schema name of *table-name* and *supertable-name*

must be the same (SQLSTATE 428DQ). The table that is identified by *supertable-name* must not have any existing subtable already defined that uses *type-name1* (SQLSTATE 42742).

The columns of the table include the object identifier column of the supertable with its type modified to be REF(*type-name1*), followed by columns based on the attributes of *type-name1* (remember that the type includes the attributes of its supertype). The attribute names cannot be the same as the OID column name (SQLSTATE 42711).

Other table options, including table space, data capture, not logged initially, and distribution key options cannot be specified. These options are inherited from the supertable (SQLSTATE 42613).

INHERIT SELECT PRIVILEGES
Any user or group holding a SELECT privilege on the supertable is granted an equivalent privilege on the newly created subtable. The subtable definer is considered to be the grantor of this privilege.

typed-element-list
Defines the additional elements of a typed table. This includes the additional options for the columns, the addition of an object identifier column (root table only), and constraints on the table.

OID-column-definition
Defines the object identifier column for the typed table.

REF IS *OID-column-name* USER GENERATED
Specifies that an object identifier (OID) column is defined in the table as the first column. An OID is required for the root table of a table hierarchy (SQLSTATE 428DX). The table must be a typed table (the OF clause must be present) that is not a subtable (SQLSTATE 42613). The name for the column is defined as *OID-column-name* and cannot be the same as the name of any attribute of the structured type *type-name1* (SQLSTATE 42711). The column is defined with type REF(*type-name1*), NOT NULL, and a system required unique index (with a default index name) is generated. This column is referred to as the *object identifier column* or *OID column*. The keywords USER GENERATED indicate that the initial value for the OID column must be provided by the user when inserting a row. Once a row is inserted, the OID column cannot be updated (SQLSTATE 42808).

with-options
Defines additional options that apply to columns of a typed table.

column-name
Specifies the name of the column for which additional options are specified. The *column-name* must correspond to the name of a column of the table that is not also a column of a supertable (SQLSTATE 428DJ). A column name can only appear in one WITH OPTIONS clause in the statement (SQLSTATE 42613).

If an option is already specified as part of the type definition (in CREATE TYPE), the options specified here override the options in CREATE TYPE.

WITH OPTIONS *column-options*
Defines options for the specified column. See *column-options* described earlier. If the table is a subtable, primary key or unique constraints cannot be specified (SQLSTATE 429B3).

LIKE *table-name1* or *view-name* or *nickname*
Specifies that the columns of the table have the same name and description as the columns of the specified table (*table-name1*), view (*view-name*), or nickname (*nickname*). The specified table, view, or nickname must either exist in the catalog or must be a declared temporary table. A typed table or typed view cannot be specified (SQLSTATE 428EC).

The use of LIKE is an implicit definition of *n* columns, where *n* is the number of columns in the identified table (including implicitly hidden columns), view, or nickname. A column of the new table that corresponds to an implicitly hidden column in the existing table will also be defined as implicitly hidden. The implicit definition depends on what is specified after LIKE:

- If a table is specified, then the implicit definition includes the column name, data type, hidden attribute, and nullability characteristic of each of the columns of that table. If EXCLUDING COLUMN DEFAULTS is not specified, then the column default is also included.

- If a view is specified, then the implicit definition includes the column name, data type, and nullability characteristic of each of the result columns of the fullselect defined in that view. The data types of the view columns must be data types that are valid for columns of a table.
- If a nickname is specified, then the implicit definition includes the column name, data type, and nullability characteristic of each column of that nickname.
- If a protected table is specified, the new table inherits the same security policy and protected columns as the identified table.
- If a table is specified and if that table contains a row-begin column, row-end column, or transaction-start-ID column, the corresponding column of the new table inherits only the data type of the source column. The new column is not considered a generated column.
- If a table that includes a period is specified, the new table does not inherit the period definition.
- If a system-period temporal table is specified, the new table is not a system-period temporal table.
- If a random distribution table that uses the **random by generation** method is specified, and if the new table that is being created does not share the same table distribution, the **RANDOM_DISTRIBUTION_KEY** column that is used to generate the random distribution values is not included.

Column default and identity column attributes can be included or excluded, based on the copy-attributes clauses. The implicit definition does not include any other attributes of the identified table, view, or nickname. Consequently, the new table does not have any primary key, unique constraints, foreign key constraints, referential integrity constraints, triggers, indexes, ORGANIZE BY specification, or PARTITIONING KEY specification. The table is created in the table space implicitly or explicitly specified by the IN clause, and the table has any other optional clause only if the optional clause is specified.

When a table is identified in the LIKE clause and that table contains a ROW CHANGE TIMESTAMP column, the corresponding column of the new table inherits only the data type of the ROW CHANGE TIMESTAMP column. The new column is not considered to be a generated column.

If a table is specified, and if row or column level access control is activated for that table, it is not inherited by the new table.

copy-options

These options specify whether to copy additional attributes of the source result table definition (table, view, or fullselect).

INCLUDING COLUMN DEFAULTS

Column defaults for each updatable column of the source result table definition are copied. Columns that are not updatable will not have a default defined in the corresponding column of the created table.

If LIKE *table-name* is specified and *table-name* identifies a base table, created temporary table, or declared temporary table, then INCLUDING COLUMN DEFAULTS is the default. If LIKE *table-name* is specified and *table-name* identifies a nickname, then INCLUDING COLUMN DEFAULTS has no effect and column defaults are not copied.

EXCLUDING COLUMN DEFAULTS

Columns defaults are not copied from the source result table definition.

This clause is the default, except when LIKE *table-name* is specified and *table-name* identifies a base table, created temporary table, or declared temporary table.

INCLUDING IDENTITY COLUMN ATTRIBUTES

Identity column attributes are copied from the source result table definition, if possible. It is possible to copy the identity column attributes, if the element of the corresponding column in the table, view, or fullselect is the name of a table column, or the name of a view column that directly or indirectly maps to the name of a base table column with the identity property. In all other cases, the columns of the new table will not get the identity property. For example:

- The select list of the fullselect includes multiple instances of the name of an identity column (that is, selecting the same column more than once).

- The select list of the fullselect includes multiple identity columns (that is, it involves a join).
- The identity column is included in an expression in the select list
- The fullselect includes a set operation (union, except, or intersect).

EXCLUDING IDENTITY COLUMN ATTRIBUTES
Identity column attributes are not copied from the source result table definition.

as-result-table

column-name
Names the columns in the table. If a list of column names is specified, it must consist of as many names as there are columns in the result table of the *fullselect*. Each *column-name* must be unique and unqualified. If a list of column names is not specified, the columns of the table inherit the names of the columns of the result table of the *fullselect*.

A list of column names must be specified if the result table of the fullselect has duplicate column names of an unnamed column (SQLSTATE 42908). An unnamed column is a column that is derived from a constant, function, expression, or set operation that is not named using the AS clause of the select list.

AS (*fullselect*)
Specifies that, for each column in the derived result table of the *fullselect*, a corresponding column is to be defined for the table. Each defined column adopts the following attributes from its corresponding column of the result table (if applicable to the data type):

- Column name
- Column description
- Data type, length, precision, and scale
- Nullability

The following attributes are not included (although the default value and identity attributes can be included by using the *copy-options*):

- Default value
- Identity attributes
- Hidden attribute
- ROW CHANGE TIMESTAMP
- Any other optional attributes of the tables or views that are referenced in the fullselect

The following restrictions apply:

- Every select list element must have a unique name (SQLSTATE 42711). The AS clause can be used in the select clause to provide unique names.
- The *fullselect* cannot refer to host variables or include parameter markers.
- The data types of the result columns of the *fullselect* must be data types that are valid for columns of a table.
- If row or column level access control (RCAC) is activated for any table that is specified in the *fullselect*, RCAC is not cascaded to the new table.
- The *fullselect* cannot include a *data-change-table-reference* clause (SQLSTATE 428FL).
- Any valid *fullselect* that does not reference a typed table or a typed view can be specified.

WITH NO DATA | WITH DATA
Determines whether to fill the columns of the table with data:

WITH NO DATA
Do not run the *fullselect*. It is used only to define the table, which is not populated with the results of the query.

WITH DATA
Run the *fullselect* and populate the table with the results of the query.

materialized-query-definition

column-name
Names the columns in the table. If a list of column names is specified, it must consist of as many names as there are columns in the result table of the fullselect. Each *column-name* must be unique and unqualified. If a list of column names is not specified, the columns of the table inherit the names of the columns of the result table of the fullselect.

A list of column names must be specified if the result table of the *fullselect* has duplicate column names of an unnamed column (SQLSTATE 42908). An unnamed column is a column that is derived from a constant, function, expression, or set operation that is not named using the AS clause of the select list.

If MAINTAINED BY REPLICATION is specified, the column names in the column list must match the names of the columns from the table that is specified in the *fullselect*.

AS
Introduces the query that is used for the definition of the table and that determines the data to be included in the table.

fullselect
Defines the query on which the table is based. The resulting column definitions are the same as those for a view that is defined with the same query. A column of the new table that corresponds to an implicitly hidden column of a base table that is referenced in the fullselect is not considered hidden in the new table.

Every select list element must have a name (use the AS clause for expressions). The *materialized-query-definition* defines attributes of the materialized query table. The option that is chosen also defines the contents of the fullselect as follows:

The fullselect cannot include a *data-change-table-reference* clause (SQLSTATE 428FL), the fetch-clause, or the ORDER BY clause (SQLSTATE 428FJ).

When REFRESH DEFERRED or REFRESH IMMEDIATE is specified, the fullselect cannot include (SQLSTATE 428EC):

- References to a materialized query table, created temporary table, declared temporary table, or typed table in any FROM clause
- References to a view where the fullselect of the view violates any of the listed restrictions on the fullselect of the materialized query table
- Expressions that are a reference type (or distinct type based on this type)
- Functions that have any of the following attributes:
  - EXTERNAL ACTION
  - LANGUAGE SQL
  - CONTAINS SQL
  - READS SQL DATA
  - MODIFIES SQL DATA
- NOT SECURED functions if the functions reference a materialized query table, which then references a table that has row or column access control activated.
- Functions that depend on physical characteristics (for example, DBPARTITIONNUM, HASHEDVALUE, RID_BIT, RID)
- A ROW CHANGE expression or reference to a ROW CHANGE TIMESTAMP column of the row
- Table or view references to system objects (Explain tables also should not be specified)
- Expressions that are a structured type, LOB type (or a distinct type based on a LOB type), or XML type
- References to a protected table or protected nickname

When DISTRIBUTE BY REPLICATION is specified, the following restrictions apply:

- The GROUP BY clause is not allowed.
- The materialized query table must only reference a single table; that is, it cannot include a join.

When MAINTAINED BY REPLICATION is specified, the following restrictions apply:

- The query must be a subselect consisting of only a SELECT clause and a FROM clause.
- The FROM clause must reference a single table that is organized by row and that is not specified in an existing shadow table definition.
- The referenced table cannot be a range-partitioned table, a multidimensional clustering table, a range-clustered table, a temporal table, or a table that contains a LONG VARCHAR or LONG VARGRAPHIC column.
- The referenced table cannot be protected by row and column access control (RCAC) or label-based access control (LBAC).
- The select list can include only direct references to the columns of the table whose data types are supported in a column-organized table. No expressions can be used.
- The columns that are specified in the select list cannot be renamed by using the column name list or the AS clause in the select list.
- The referenced table must have at least one enforced primary key constraint or unique constraint, and the columns that are specified in the select list must include all the key columns from at least one of these constraints.

When REFRESH IMMEDIATE is specified:

- The query must be a subselect, with the exception that UNION ALL is supported in the input table expression of a GROUP BY.
- The query cannot be recursive.
- The query cannot include:
  - References to a nickname
  - Functions that are not deterministic
  - Scalar fullselects
  - Predicates with fullselects
  - Special registers and built-in functions that depend on the value of a special register
  - Global variables
  - SELECT DISTINCT
  - An error tolerant *nested-table-expression*
- If the FROM clause references more than one table or view, it can only define an inner join without using the explicit INNER JOIN syntax.
- When a GROUP BY clause is specified, the following considerations apply:
  - The supported column functions are SUM, COUNT, COUNT_BIG, and GROUPING (without DISTINCT). The select list must contain a COUNT(*) or COUNT_BIG(*) column. If the materialized query table select list contains SUM(X), where X is a nullable argument, the materialized query table must also have COUNT(X) in its select list. These column functions cannot be part of any expressions.
  - A HAVING clause is not allowed.
  - If in a multiple partition database partition group, the distribution key must be a subset of the GROUP BY items.
- The materialized query table must not contain duplicate rows, and the following restrictions specific to this uniqueness requirement apply, depending upon whether a GROUP BY clause is specified.
  - When a GROUP BY clause is specified, the following uniqueness-related restrictions apply:
    - All GROUP BY items must be included in the select list.

- When the GROUP BY contains GROUPING SETS, CUBE, or ROLLUP, the GROUP BY items and associated GROUPING column functions in the select list must form a unique key of the result set. Thus, the following restrictions must be satisfied:
  - No grouping sets can be repeated. For example, `ROLLUP(X,Y),X` is not allowed, because it is equivalent to `GROUPING SETS((X,Y),(X),(X))`.
  - If X is a nullable GROUP BY item that appears within GROUPING SETS, CUBE, or ROLLUP, then GROUPING(X) must appear in the select list.

– When a GROUP BY clause is not specified, the following uniqueness-related restrictions apply:
  - The materialized query table's uniqueness requirement is achieved by deriving a unique key for the materialized view from one of the unique key constraints defined in each of the underlying tables. Therefore, the underlying tables must have at least one unique key constraint that is defined on them, and the columns of these keys must appear in the select list of the materialized query table definition.

When REFRESH DEFERRED is specified:

- If the materialized query table is created with the intention of providing it with an associated staging table in a later statement, the fullselect of the materialized query table must follow the same restrictions and rules as a fullselect used to create a materialized query table with the REFRESH IMMEDIATE option.
- If the query is recursive, the materialized query table is not used to optimize the processing of queries.
- The materialized query table is not used to optimize the processing of static queries.

A materialized query table whose fullselect contains a GROUP BY clause is summarizing data from the tables that are referenced in the fullselect. Such a materialized query table is also known as a *summary table*. A summary table is a specialized type of materialized query table.

If the *fullselect* references a table or a view that depends on a table for which row or column level access control has been activated, those row or column level access controls are ignored when populating the materialized query table. The materialized query table is automatically created with row level access control activated. Direct access by users to this table does not see any content unless appropriate permissions are created or a user with SECADM authority chooses to deactivate row level access control on this materialized query table. Row and column level access control on the materialized query table does not affect internal routing by the SQL compiler to the materialized query table.

refreshable-table-options

Define the refreshable options of the materialized query table attributes.

DATA INITIALLY DEFERRED

Data is not inserted into the table as part of the CREATE TABLE statement. A REFRESH TABLE statement specifying the *table-name* is used to insert data into the table.

REFRESH

Indicates how the data in the table is maintained.

DEFERRED

The data in the table can be refreshed at any time by using the REFRESH TABLE statement. The data in the table only reflects the result of the query as a snapshot at the time the REFRESH TABLE statement is processed. System-maintained materialized query tables that are defined with this attribute do not allow INSERT, UPDATE, or DELETE statements (SQLSTATE 42807). User-maintained materialized query tables that are defined with this attribute do allow INSERT, UPDATE, or DELETE statements.

IMMEDIATE

The changes that are made to the underlying tables as part of a DELETE, INSERT, or UPDATE are cascaded to the materialized query table. In this case, the content of the table, at any point-in-time, is the same as if the specified *subselect* is processed.

Materialized query tables (MQTs) defined with this attribute do not allow INSERT, UPDATE, or DELETE statements (SQLSTATE 42807). Column-organized MQTs using the REFRESH IMMEDIATE option are not supported when the MAINTAINED BY SYSTEM clause is specified (SQL20058N).

ENABLE QUERY OPTIMIZATION
The materialized query table can be used for query optimization under appropriate circumstances.

DISABLE QUERY OPTIMIZATION
The materialized query table will not be used for query optimization. The table can still be queried directly.

MAINTAINED BY
Specifies whether the data in the materialized query table is maintained by the system, user, or replication tool. The default is SYSTEM.

SYSTEM
Specifies that the data in the materialized query table is maintained by the system. A system-maintained materialized query table that is defined as ORGANIZE BY COLUMN must use the REFRESH DEFERRED and DISTRIBUTE BY REPLICATION options.

USER
Specifies that the data in the materialized query table is maintained by the user. The user is allowed to perform update, delete, or insert operations against user-maintained materialized query tables. The REFRESH TABLE statement, which is used for system-maintained materialized query tables, cannot be invoked against user-maintained materialized query tables. Only a REFRESH DEFERRED materialized query table can be defined as MAINTAINED BY USER.

REPLICATION
Specifies that the data in the materialized query table is maintained by an external replication technology. MAINTAINED BY REPLICATION cannot be specified in a partitioned database environment or in a Db2 pureScale environment (SQLSTATE 56038). The REFRESH TABLE statement, which is used for system-maintained materialized query tables, cannot be issued against replication-maintained materialized query tables, which are referred to as *shadow tables*. Only a REFRESH DEFERRED materialized query table can be defined as MAINTAINED BY REPLICATION, and the definition must include ORGANIZE BY COLUMN.

FEDERATED_TOOL
Specifies that the data in the materialized query table is maintained by a federated replication tool. The REFRESH TABLE statement, which is used for system-maintained materialized query tables, cannot be invoked against federated_tool-maintained materialized query tables. Only a REFRESH DEFERRED materialized query table can be defined as MAINTAINED BY FEDERATED_TOOL.

When specifying this option, the select clause in the CREATE TABLE statement cannot contain a reference to a base table (SQLSTATE 428EC).

staging-table-definition
Defines the query that is supported by the staging table indirectly through an associated materialized query table. The underlying tables of the materialized query table are also the underlying tables for its associated staging table. The staging table collects changes that need to be applied to the materialized query table to synchronize it with the contents of the underlying tables.

If the *fullselect* references a table or a view that depends on a table for which row or column level access control has been activated, those row or column level access controls are ignored when populating the staging table. However, the staging table is automatically created with row level access control activated. Direct access by users to this staging table does not see any content unless appropriate permissions are created or a user with SECADM authority chooses to deactivate row level access control on this staging table. Row and column level access control on the staging table does not affect the internal process of applying the changes that are captured by the staging table to the associated materialized query table.

staging-column-name
Names the columns in the staging table. If a list of column names is specified, it must consist of *two* more names than exist columns in the materialized query table for which the staging table is defined. If the materialized query table is a replicated materialized query table, or the query defining the materialized query table does not contain a GROUP BY clause, the list of column names must consist of *three* more names than there are columns in the materialized query table for which the staging table is defined. Each column name must be unique and unqualified. If a list of column names is not specified, the columns of the table inherit the names of the columns of the associated materialized query table. The additional columns are named GLOBALTRANSID and GLOBALTRANSTIME, and if a third column is necessary, it is named OPERATIONTYPE.

Table 138. Extra Columns Appended in Staging Tables

Column Name	Data Type	Column Description
GLOBALTRANSID	CHAR(8) FOR BIT DATA	The global transaction ID for each propagated row
GLOBALTRANSTIME	CHAR(13) FOR BIT DATA	The time stamp of the transaction
OPERATIONTYPE	INTEGER	Operation for the propagated row, either insert, update, or delete.

A list of column names must be specified if any of the columns of the associated materialized query table duplicate any of the generated column names (SQLSTATE 42711).

FOR *table-name2*
Specifies the materialized query table that is used for the definition of the staging table. The name, including the implicit or explicit schema, must identify a materialized query table that exists at the current server defined with REFRESH DEFERRED. The fullselect of the associated materialized query table must follow the same restrictions and rules as a fullselect used to create a materialized query table with the REFRESH IMMEDIATE option.

The contents of the staging table can be used to refresh the materialized query table, by invoking the REFRESH TABLE statement, if the contents of the staging table are consistent with the associated materialized query table and the underlying source tables.

PROPAGATE IMMEDIATE
The changes that are made to the underlying tables as part of a delete, insert, or update operation are cascaded to the staging table in the same delete, insert, or update operation. If the staging table is not marked inconsistent, its content, at any point-in-time, is the delta changes to the underlying table since the last refresh materialized query table.

ORGANIZE BY
Specifies how the data is organized in the data pages of the table.

The following restrictions apply to a column-organized MQT:

- MQTs other than shadow tables must reference tables with the same organization as the MQT.
- The ORGANIZE BY COLUMN clause must be specified when creating a column-organized MQT, even if the **dft_table_org** database configuration parameter is set to COLUMN.
- For a column-organized MQT, the following types of tables can be used:
  - Shadow tables
  - User-maintained MQTs
  - System-maintained MQTs that are defined with the `REFRESH DEFERRED` and `DISTRIBUTE BY REPLICATION` clauses.

The default organization is determined by the value of the dft_table_org database configuration parameter. If **dft_table_org** is not specified, the default is ROW.

ROW
The data is stored by row in the data pages of the table. A given data page stores the data for one or more rows of the table.

COLUMN
The data is stored by column in the data pages of the table. Each data page stores data for one column of the table.

ROW USING
The data is stored by row in the data pages of the table and is further organized by using a dimensions clause, key sequence, or insert time. If you specify a dimensions clause, key sequence, or insert time, specifying ROW USING is optional unless the default table organization for the database is COLUMN, in which case specifying ROW USING is mandatory.

DIMENSIONS (*column-name*,...)
Specifies a dimension for each column or group of columns used to cluster the table data. A table whose definition specifies this clause is known as a multidimensional clustering (MDC) table. Use parentheses within the dimension list to specify that a group of columns is to be treated as a single dimension. The DIMENSIONS keyword is optional.

A clustering block index is automatically maintained for each specified dimension, and a block index, consisting of all columns used in the clause, is maintained if none of the clustering block indexes include them all. The set of columns that are used in the ORGANIZE BY clause must follow the rules for the CREATE INDEX statement that specifies CLUSTER.

Each column name that is specified in the ORGANIZE BY clause must be defined for the table (SQLSTATE 42703). A dimension cannot occur more than once in the dimension list (SQLSTATE 42709). The dimensions cannot contain a ROW CHANGE TIMESTAMP column, row-begin column, row-end column, transaction-start-ID column (SQLSTATE 429BV), or an XML column (SQLSTATE 42962). If the table uses extended row size, each dimension column with a data type of VARCHAR or VARGRAPHIC cannot have a length attribute that is greater than 24 bytes (SQLSTATE 54010).

Pages of the table are arranged in blocks of equal size, which is the extent size of the table space, and all rows of each block contain the same combination of dimension values.

A table can be both a multidimensional clustering (MDC) table and a partitioned table. Columns in such a table can be used in both the *range-partition-spec* and in the MDC key. Table partitioning is multi-column, not multidimensional.

For a partitioned MDC table created by Db2 Version 9.7 Fix Pack 1 or later releases, the block indexes are partitioned. The partitioned block index placement follows the general partitioned index storage placement rule. All index partitions for a given data partition, including MDC block indexes, share a single index object. By default, the index partitions for each specific data partition reside in the same table space as the data partition. This can be overridden with the partition level INDEX IN clause.

For MDC tables that were created using Db2 V9.7 or earlier, the block indexes are nonpartitioned and remain nonpartitioned if they are rebuilt. MDC tables with partitioned block indexes can co-exist in the same database as MDC tables with nonpartitioned block indexes. To change nonpartitioned block indexes to partitioned block indexes, use an online table move to migrate the MDC table.

KEY SEQUENCE *sequence-key-spec*
Specifies that the table is organized in ascending key sequence with a fixed size based on the specified range of key sequence values. A table that is organized in this way is referred to as a *range-clustered table*. Each possible key value in the defined range has a predetermined location in the physical table. The storage that is required for a range-clustered table must be available when the table is created, and must be sufficient to contain the number of rows in the specified range multiplied by the row size (for details on determining the space requirement, see Row Size Limit and Byte Counts).

column-name
Specifies a column of the table that is included in the unique key that determines the sequence of the range-clustered table. The data type of the column must be SMALLINT, INTEGER, or BIGINT (SQLSTATE 42611), and the columns must be defined as NOT NULL (SQLSTATE 42831). The same column must not be identified more than once in the sequence key. The number of identified columns must not exceed 64 (SQLSTATE 54008).

A unique index entry will automatically be created in the catalog for the columns in the key sequence specified with ascending order for each column. The name of the index will be SQL, followed by a character time stamp (*yymmddhhmmssxxx*), with SYSIBM as the schema name. An actual index object is not created in storage because the table organization is ordered by this key. If a primary key or a unique constraint is defined on the same columns as the range-clustered table sequence key, this same index entry is used for the constraint.

For the key sequence specification, a check constraint exists to reflect the column constraints. If the DISALLOW OVERFLOW clause is specified, the name of the check constraint is RCT, and the check constraint is enforced. If the ALLOW OVERFLOW clause is specified, the name of the check constraint is RCT_OFLOW, and the check constraint is not enforced.

STARTING FROM *constant*
Specifies the constant value at the low end of the range for *column-name*. Values less than the specified constant are only allowed if the ALLOW OVERFLOW option is specified. If *column-name* is a SMALLINT or INTEGER column, the constant must be an INTEGER constant. If *column-name* is a BIGINT column, the constant must be an INTEGER or BIGINT constant (SQLSTATE 42821). If a starting constant is not specified, the default value is 1.

ENDING AT *constant*
Specifies the constant value at the high end of the range for *column-name*. Values greater than the specified constant are only allowed if the ALLOW OVERFLOW option is specified. The value of the ending constant must be greater than the starting constant. If *column-name* is a SMALLINT or INTEGER column, the constant must be an INTEGER constant. If *column-name* is a BIGINT column, the constant must be an INTEGER or BIGINT constant (SQLSTATE 42821).

ALLOW OVERFLOW
Specifies that the range-clustered table allows rows with key values that are outside of the defined range of values. When a range-clustered table is created to allow overflows, the rows with key values outside of the range are placed at the end of the defined range without any predetermined order. Operations involving these overflow rows are less efficient than operations on rows having key values within the defined range.

DISALLOW OVERFLOW
Specifies that the range-clustered table does not allow rows with key values that are not within the defined range of values (SQLSTATE 23513). Range-clustered tables that disallow overflows will always maintain all rows in ascending key sequence.

The DISALLOW OVERFLOW clause cannot be specified if the table is a range-clustered materialized query table (SQLSTATE 429BG).

PCTFREE *integer*
Specifies the percentage of each page that is to be left as free space. The first row on each page is added without restriction. When additional rows are added to a page, at least *integer* percent of the page is left as free space. The value of *integer* can range from 0 to 99. A PCTFREE value of -1 in the system catalog (SYSCAT.TABLES) is interpreted as the default value. The default PCTFREE value for a table page is 0.

INSERT TIME
Specifies that rows are clustered in the table relative to the time they are inserted. Rows are inserted at the logical end of the table object instead of searching for available space.

A table that is organized by insert time is known as an insert time clustering (ITC) table. This type of table can use REORG TABLE RECLAIM EXTENTS to reclaim free extents for immediate use by other objects in the table space.

Data is clustered by using an implicitly created virtual dimension. A clustering block index is automatically maintained for this virtual dimension. The virtual dimension cannot be manipulated and it uses no space for each row that exists in the table. Pages of the table are arranged in blocks of equal size, which is the extent size of the table space.

The ORGANIZE BY INSERT TIME clause cannot be specified if the table is a typed table (SQLSTATE 428DH).

DATA CAPTURE
Indicates whether extra information for inter-database data replication is to be written to the log. This clause cannot be specified when creating a subtable (SQLSTATE 428DR).

If the clause is not specified and that table is not a typed table, then the default is determined by the DATA CAPTURE setting of the schema at the time the table is created.

NONE
Indicates that no extra information will be logged.

CHANGES
Indicates that extra information regarding SQL changes to this table will be written to the log. This option is required if this table will be replicated and the Capture program is used to capture changes for this table from the log.

If the table is a typed table that is not a subtable, then this option is not supported (SQLSTATE 428DH).

IN *tablespace-name*,...
Identifies the table spaces in which the table will be created. The table spaces must exist, they must be in the same database partition group, and they must be all regular DMS or all large DMS or all SMS table spaces (SQLSTATE 42838) on which the authorization ID of the statement holds the USE privilege.

A maximum of one IN clause is allowed at the table level. All data table spaces that are used by a table must have the same page size and extent size.

If only one table space is specified, all table parts are stored in this table space. This clause cannot be specified when creating a subtable (SQLSTATE 42613) because the table space is inherited from the root table of the table hierarchy.

If this clause is not specified, the database manager chooses a table space (from the set of existing table spaces in the database) with the smallest sufficient page size and where the row size is within the row size limit of the page size on which the authorization ID of the statement has USE privilege.

If more than one table space qualifies, choose the table space in the following order of preference, depending how the authorization ID of the statement was granted USE privilege on the table space:

1. The authorization ID
2. A role to which the authorization ID is granted
3. A group to which the authorization ID belongs
4. A role to which a group the authorization ID belongs is granted
5. PUBLIC
6. A role to which PUBLIC is granted

If more than one table space still qualifies, the final choice is made by the database manager.

Table space determination can change if:

- Table spaces are dropped or created
- USE privileges are granted or revoked

Partitioned tables can have their data partitions spread across multiple table spaces. When multiple table spaces are specified, all of the table spaces must exist, and they must all be either SMS or regular DMS or large DMS table spaces (SQLSTATE 42838). The authorization ID of the statement must hold the USE privilege on all of the specified table spaces.

The sufficient page size of a table is determined by either the byte count of the row or the number of columns. For more information, see Row Size Limits.

When a table is placed in a large table space:

- The table can be larger than a table in a regular table space. For more information on table and table space limits, see "SQL limits".
- The table can support more than 255 rows per data page, which can improve space usage on data pages.
- Indexes that are defined on the table will require an extra 2 bytes per row entry, compared to indexes defined on a table that resides in a regular table space.

CYCLE or NO CYCLE
Specifies whether the number of data partitions with no explicit table space can exceed the number of specified table spaces.

CYCLE
Specifies that if the number of data partitions with no explicit table space exceeds the number of specified table spaces, the table spaces are assigned to data partitions in a round-robin fashion.

NO CYCLE
Specifies that the number of data partitions with no explicit table space must not exceed the number of specified tables spaces (SQLSTATE 428G1). This option prevents the round-robin assignment of table spaces to data partitions.

tablespace-options
Specifies the table space in which indexes or long column values are to be stored. For details on types of table spaces, see "CREATE TABLESPACE".

INDEX IN *tablespace-name*
Identifies the table space in which any indexes on a nonpartitioned table or nonpartitioned indexes on a partitioned table are to be created. The specified table space must exist; it must be a DMS table space if the table has data in DMS table spaces, or an SMS table space if the partitioned table has data in SMS table spaces; it must be a table space on which the authorization ID of the statement holds the USE privilege; and it must be in the same database partition group as *tablespace-name* (SQLSTATE 42838).

Specifying which table space will contain indexes can be done when a table is created, or in the case of partitioned tables, it can be done by specifying the IN clause of the CREATE INDEX statement for a nonpartitioned index. Checking for the USE privilege on the table space is done at table creation time, not when an index is created later.

For a nonpartitioned index on a partitioned table, storage of the index is as follows:

- The table space by the IN clause of the CREATE INDEX statement
- The table-level table space that is specified for the INDEX IN clause of the CREATE TABLE statement
- If neither of the preceding are specified, the index is stored in the table space of the first attached or visible data partition

For more information about partitioned indexes on partitioned tables, see the description of the partition-element INDEX IN clause.

LONG IN *tablespace-name*
Identifies the table spaces in which the values of any long columns are to be stored. Long columns include those with LOB data types, XML type, distinct types with any of these as source types, or any columns that are defined with user-defined structured types whose

values cannot be stored inline. This option is allowed only if the IN clause identifies a DMS table space.

Note: An automatic storage table space is also a DMS table space.

The specified table space must exist. It can be a regular table space if it is the same table space in which the data is stored; otherwise, it must be a large DMS table space on which the authorization ID of the statement holds the USE privilege. It must also be in the same database partition group as *tablespace-name* (SQLSTATE 42838).

Specifying which table space will contain long, LOB, or XML columns can only be done when a table is created. Checking for the USE privilege is done at table creation time, not when a long or LOB column is added later.

For rules governing the use of the LONG IN clause with partitioned tables, see "Large object behavior in partitioned tables".

distribution-clause

Specifies the database partitioning or the way the data is distributed across multiple database partitions.

DISTRIBUTE BY HASH (*column-name*,...)

Specifies the use of the default hashing function on the specified columns as the distribution method across database partitions. The specified columns are called a *distribution key*.

- Each column name must be an unqualified name that identifies a column of the table (SQLSTATE 42703).
- The same column must not be identified more than once (SQLSTATE 42709).
- A column cannot be used as part of a distribution key if its data type is BLOB, CLOB, DBCLOB, XML, a distinct type based on any of these types, or a structured type (SQLSTATE 42962).
- The distribution key cannot contain a ROW CHANGE TIMESTAMP column (SQLSTATE 429BV).
- A distribution key cannot be specified for a table that is a subtable, because the distribution key is inherited from the root table in the table hierarchy (SQLSTATE 42613).
- A distribution key cannot contain row begin, row end, or transaction start ID columns.
- If a DISTRIBUTE BY HASH clause is not specified, and if the table resides in a multiple partition database partition group with multiple database partitions, a default distribution key is automatically defined.
- The columns of the distribution key must be a subset of the columns that make up any enforced unique constraints.

If none of the columns satisfy the requirements for a default distribution key, the table is created without one. Such tables are allowed only in table spaces that are defined on single-partition database partition groups.

For tables in table spaces that are defined on single-partition database partition groups, any collection of columns with data types that are valid for a distribution key can be used to define the distribution key. If you do not specify this clause, no distribution key is created.

For restrictions related to the distribution key, see Rules.

DISTRIBUTE BY RANDOM

Specifies that the database manager will select a distribution key to spread data evenly across all database partitions of the database partitioning group. There are two methods that the database manager uses to achieve this:

- **Random by unique**: If the table includes a unique or primary key, it uses the unique characteristics of the key columns to create a random spread of the data. The columns of the unique or primary key are used as the distribution keys.
- **Random by generation**: If the table does not have a unique or primary key, the database manager will include a column in the table to generate and store a generated value to use in the hashing function. The column will be created with the **IMPLICITLY HIDDEN** clause

so that it does not appear in queries unless explicitly included. The value of the column is automatically generated as new rows are added to the table. By default, the column name is **RANDOM_DISTRIBUTION_KEY**. If it collides with the existing column, a non-conflicting name is generated by the database manager.

DISTRIBUTE BY REPLICATION
Specifies that the data that is stored in the table is physically replicated on each database partition of the database partition group for the table spaces in which the table is defined. This means that a copy of all of the data in the table exists on each database partition. This option can only be specified for a materialized query table (SQLSTATE 42997).

partitioning-clause
Specifies how the data is partitioned within a database partition.

PARTITION BY RANGE *range-partition-spec*
Specifies the table partitioning scheme for the table.

partition-expression
Specifies the key data over which the range is defined to determine the target data partition of the data.

column-name
Identifies a column of the table-partitioning key. The *column-name* must be an unqualified name that identifies a column of the table (SQLSTATE 42703). The same column must not be identified more than once (SQLSTATE 42709). No column with a data type that is a BLOB, CLOB, DBCLOB, XML, distinct type based on any of these types, or structured type can be used as part of a table-partitioning key (SQLSTATE 42962).

The numeric literals that are used in the range specification are governed by the rules for numeric literals. All of the numeric literals (except the decimal floating-point special values) used in ranges corresponding to numeric columns are interpreted as integer, floating-point or decimal constants, in accordance with the rules specified for numeric constants. As a result, for decimal floating-point columns, the minimum and maximum numeric constant value that can be used in the range specification of a data partition is the smallest DOUBLE value and the largest DOUBLE value, respectively. Decimal floating-point special values can be used in the range specification. All decimal floating-point special values are interpreted as greater than MINVALUE and less than MAXVALUE.

The table partitioning columns cannot contain a ROW CHANGE TIMESTAMP column (SQLSTATE 429BV). The number of identified columns must not exceed 16 (SQLSTATE 54008).

NULLS LAST or NULLS FIRST
Indicates the partition placement of rows that have null values in the table partitioning key columns. These clauses do not affect the order of rows that are returned in an ORDER BY clause.

NULLS LAST
Indicates that null values are compared as the highest possible value, and are placed in a range ending at MAXVALUE.

NULLS FIRST
Indicates that null values are compared as the lowest possible value, and are placed in a range starting at MINVALUE.

partition-element
Specifies ranges for a data partitioning key and the table space where rows of the table in the range will be stored.

PARTITION *partition-name*
Names the data partition. The name must not be the same as any other data partition for the table (SQLSTATE 42710). If this clause is not specified, the name will be "PART" followed by the character form of an integer value to make the name unique for the table.

boundary-spec
Specifies the boundaries of a data partition. The lowest data partition must include a starting-clause, and the highest data partition must include an ending-clause (SQLSTATE 56016). Data partitions between the lowest and the highest can include either a starting-clause, ending-clause, or both clauses. If only the ending-clause is specified, the previous data partition must also have included an ending-clause (SQLSTATE 56016).

starting-clause
Specifies the low end of the range for a data partition. There must be at least one starting value specified and no more values than the number of columns in the data partitioning key (SQLSTATE 53038). If fewer values are specified than the number of columns, the remaining values are implicitly MINVALUE.

STARTING FROM
Introduces the *starting-clause*.

constant
Specifies a constant value with a data type that is assignable to the data type of the *column-name* to which it corresponds (SQLSTATE 53045). The value must not be in the range of any other boundary-spec for the table (SQLSTATE 56016).

MINVALUE
Specifies a value that is lower than the lowest possible value for the data type of the *column-name* to which it corresponds.

MAXVALUE
Specifies a value that is greater than the greatest possible value for the data type of the *column-name* to which it corresponds.

INCLUSIVE
Indicates that the specified range values are to be included in the data partition.

EXCLUSIVE
Indicates that the specified *constant* values are to be excluded from the data partition. This specification is ignored when MINVALUE or MAXVALUE is specified.

ending-clause
Specifies the high end of the range for a data partition. There must be at least one starting value specified and no more values than the number of columns in the data partitioning key (SQLSTATE 53038). If fewer values are specified than the number of columns, the remaining values are implicitly MAXVALUE.

ENDING AT
Introduces the *ending-clause*.

constant
Specifies a constant value with a data type that is assignable to the data type of the *column-name* to which it corresponds (SQLSTATE 53045). The value must not be in the range of any other boundary-spec for the table (SQLSTATE 56016).

MINVALUE
Specifies a value that is lower than the lowest possible value for the data type of the *column-name* to which it corresponds.

MAXVALUE
Specifies a value that is greater than the greatest possible value for the data type of the *column-name* to which it corresponds.

INCLUSIVE
Indicates that the specified range values are to be included in the data partition.

EXCLUSIVE
Indicates that the specified *constant* values are to be excluded from the data partition. This specification is ignored when MINVALUE or MAXVALUE is specified.

IN *tablespace-name*
Specifies the table space where the data partition is to be stored. The named table space must have the same page size, be in the same database partition group, and manage space in the same way as the other table spaces of the partitioned table (SQLSTATE 42838); it must be a table space on which the authorization ID of the statement holds the USE privilege. If this clause is not specified, a table space is assigned by default in a round-robin fashion from the list of table spaces that are specified for the table. If a table space was not specified for large objects by using the LONG IN clause, large objects are placed in the same table space as are the rest of the rows for the data partition. For partitioned tables, the LONG IN clause can be used to provide a list of table spaces. This list is used in round robin-fashion to place large objects for each data partition. For rules governing the use of the LONG IN clause with partitioned tables, see "Large object behavior in partitioned tables".

If the INDEX IN clause is not specified on the CREATE TABLE or the CREATE INDEX statement, the index is placed in the same table space as the first visible or attached partition of the table.

INDEX IN *tablespace-name*
Specifies the table space where the partitioned index on the partitioned table is to be stored.

The partition-element level INDEX IN clause only affects the storage of partitioned indexes. Storage of the index is as follows:

- If the INDEX IN clause is specified at the partition level when the table is created, the partitioned index is stored in the specified table space.
- If the INDEX IN clause is not specified at the partition level when the table is created, the partitioned index is stored in the table space of the corresponding data partition.

The INDEX IN clause can only be specified if the data table spaces are DMS table spaces and the table space specified by the INDEX IN clause is a DMS table space. If the data table space is an SMS table space, an error is returned (SQLSTATE 42839).

LONG IN *tablespace-name*
Identifies the table spaces in which the values of any long columns are to be stored. Long columns include those with LOB data types, XML type, distinct types with any of these as source types, or any columns defined with user-defined structured types whose values cannot be stored inline. This option is allowed only if the IN clause identifies a DMS table space.

Note: An automatic storage table space is also a DMS table space.

The specified table space must exist. It can be a regular table space if it is the same table space in which the data is stored; otherwise, it must be a large DMS table space on which the authorization ID of the statement holds the USE privilege. It must also be in the same database partition group as *tablespace-name* (SQLSTATE 42838).

Specifying which table space will contain long, LOB, or XML columns can only be done when a table is created. Checking for the USE privilege is done at table creation time, not when a long or LOB column is added later.

For rules governing the use of the LONG IN clause with partitioned tables, see "Large object behavior in partitioned tables".

EVERY (*constant*)
Specifies the width of each data partition range when using the automatically generated form of the syntax. Data partitions will be created starting at the STARTING FROM value and containing this number of values in the range. This form of the syntax is only supported for tables that are partitioned by a single numeric or datetime column (SQLSTATE 53038).

If the partitioning key column is a numeric type, the starting value of the first partition is the value that is specified in the starting-clause. The ending value for the first and all other partitions is calculated by adding the starting value of the partition to the increment value specified as *constant* in the EVERY clause. The starting value for all other partitions is calculated by taking the starting value for the previous partition and adding the increment value that is specified as *constant* in the EVERY clause.

If the partitioning key column is a DATE or a TIMESTAMP, the starting value of the first partition is the value that is specified in the starting-clause. The ending value for the first and all other partitions is calculated by adding the starting value of the partition to the increment value specified as a labeled duration in the EVERY clause. The starting value for all other partitions is calculated by taking the starting value for the previous partition and adding the increment value that is specified as a labeled duration in the EVERY clause.

For a numeric column, the EVERY value must be a positive numeric constant, and for a datetime column, the EVERY value must be a labeled duration (SQLSTATE 53045).

COMPRESS

Specifies whether row compression is to be used for the table.The **ddl_compression_def** configuration parameter determines the default value of the COMPRESS keyword.

NO

Row compression is disabled.

YES

Row compression is enabled. Insert and update operations on the table use row compression. Any XML storage objects that exist are also compressed. For both adaptive and classic row compression, a table-level compression dictionary is automatically created after the table is sufficiently populated with data. This also applies to the data in the XML storage object; if there is sufficient data in the XML storage object, a compression dictionary is automatically created and XML documents are subject to compression.

Note: The compression that is applied to the XML storage object is the same, regardless of whether you use adaptive or classic row compression.

For adaptive row compression, page-level compression dictionaries are created or updated as soon as data is inserted or changed in the table.

ADAPTIVE

Enables adaptive compression, and records are subject to being compressed with a table-level and a page-level compression dictionary. The functionality of COMPRESS YES ADAPTIVE is a superset of the functionality of COMPRESS YES STATIC. This is the default when COMPRESS YES is explicitly specified.

STATIC

Enables classic row compression using a table-level compression dictionary. This is the same row compression functionality that existed in previous Db2 versions. This is the default when row compression is used by default but COMPRESS YES is not explicitly specified.

VALUE COMPRESSION

This determines the row format that is to be used. Each data type has a different byte count depending on the row format that is used. For more information, see Byte Counts. If the table is a typed table, this option is only supported on the root table of the typed table hierarchy (SQLSTATE 428DR).

The null value is stored using 3 bytes. This is the same or less space than when VALUE COMPRESSION is not active for columns of all data types, except for CHAR(1). Whether a column is defined as nullable has no effect on the row size calculation. The zero-length data values for columns whose data type is VARCHAR, VARGRAPHIC, LONG VARCHAR, LONG VARGRAPHIC, CLOB, DBCLOB, VARBINARY, BLOB, or XML are to be stored using 2 bytes only, which is less than the storage required when VALUE COMPRESSION is not active. When a column is defined using the COMPRESS SYSTEM DEFAULT option, this also allows the system default value for the column to be stored using 3 bytes of total storage. The row format that is used to support this determines the byte counts for each data type,

and tends to cause data fragmentation when updating to or from the null value, a zero-length value, or the system default value.

WITH RESTRICT ON DROP
Indicates that the table cannot be dropped, and that the table space that contains the table cannot be dropped.

NOT LOGGED INITIALLY
Any changes that are made to the table by an Insert, Delete, Update, Create Index, Drop Index, or Alter Table operation in the same unit of work in which the table is created are not logged. For other considerations when using this option, see the "Notes" section of this statement.

All catalog changes and storage-related information are logged, as are all operations that are done on the table in subsequent units of work.

Note: If non-logged activity occurs against a table that has the NOT LOGGED INITIALLY attribute activated, and if a statement fails (causing a rollback), or a ROLLBACK TO SAVEPOINT is executed, the entire unit of work is rolled back (SQL1476N). Furthermore, the table for which the NOT LOGGED INITIALLY attribute was activated is marked inaccessible after the rollback has occurred, and can only be dropped. Therefore, the opportunity for errors within the unit of work in which the NOT LOGGED INITIALLY attribute is activated should be minimized.

CCSID
Specifies the encoding scheme for string data that is stored in the table. If the CCSID clause is not specified, the default is CCSID UNICODE for Unicode databases, and CCSID ASCII for all other databases.

ASCII
Specifies that string data is encoded in the database code page. If the database is a Unicode database, CCSID ASCII cannot be specified (SQLSTATE 56031).

UNICODE
Specifies that string data is encoded in Unicode. Character data is in UTF-8. Graphic data is not allowed.

If the database is not a Unicode database, tables can be created with CCSID UNICODE, but the following rules apply:

- The alternative collating sequence must be specified in the database configuration before creating the table (SQLSTATE 56031). CCSID UNICODE tables collate with the alternative collating sequence that is specified in the database configuration.

  The only supported alternative collating sequence is IDENTITY_16BIT.
- Graphic types, the XML type, and user-defined types cannot be used in CCSID UNICODE tables (SQLSTATE 560C1).
- Anchored data types cannot anchor to columns of a table that is created with CCSID UNICODE (SQLSTATE 428HS).
- Tables cannot have both the CCSID UNICODE clause and the DATA CAPTURE CHANGES clause specified (SQLSTATE 42613).
- The Explain tables cannot be created with CCSID UNICODE (SQLSTATE 55002).
- Created temporary tables and declared temporary tables cannot be created with CCSID UNICODE (SQLSTATE 56031).
- CCSID UNICODE tables cannot be created in a CREATE SCHEMA statement (SQLSTATE 53090).
- The exception table for a load operation must have the same CCSID as the target table for the operation (SQLSTATE 428A5).
- The exception table for a SET INTEGRITY statement must have the same CCSID as the target table for the statement (SQLSTATE 53090).
- The target table for event monitor data must not be declared as CCSID UNICODE (SQLSTATE 55049).

- SQL statements are always interpreted in the database code page. In particular, this means that every character in literals, hex literals, and delimited identifiers must have a representation in the database code page; otherwise, the character will be replaced with the substitution character.

Host variables in the application are always in the application code page, regardless of the CCSID of any tables in the SQL statements that are invoked. The database manager will perform code page conversions as necessary to convert data between the application code page and the section code page. The registry variable DB2CODEPAGE can be set at the client to change the application code page.

SECURITY POLICY
Names the security policy to be associated with the table.

policy-name
Identifies a security policy that already exists at the current server (SQLSTATE 42704).

OPTIONS (*table-option-name string-constant*, ...)
Table options are used to identify the remote base table. The *table-option-name* is the name of the option. The *string-constant* specifies the setting for the table option. The *string-constant* must be enclosed in single quotation marks.

The remote server (the server name that was specified in the CREATE SERVER statement) must be specified in the OPTIONS clause. The OPTIONS clause can also be used to override the schema or the unqualified name of the remote base table that is being created.

It is recommended that a schema name is specified. If a remote schema name is not specified, the qualifier for the table name is used. If the table name has no qualifier, the authorization ID of the statement is used.

If an unqualified name for the remote base table is not specified, *table-name* is used.

Rules

- The sum of the byte counts of the columns, including the inline lengths of all structured or XML type columns, must not be greater than the row size limit that is based on the page size of the table space (SQLSTATE 54010). For more information, see Byte Counts. For typed tables, the byte count is applied to the columns of the root table of the table hierarchy, and every additional column introduced by every subtable in the table hierarchy (extra subtable columns must be considered nullable for byte count purposes, even if defined as not nullable). There is also an additional 4 bytes of overhead to identify the subtable to which each row belongs.
- The number of columns in a table cannot exceed 1,012 (SQLSTATE 54011). For typed tables, the total number of attributes of the types of all of the subtables in the table hierarchy cannot exceed 1010. For random distribution tables using the random by generation method, the number of columns cannot exceed 1,011 because of the inclusion of the **RANDOM_DISTRIBUTION_KEY** column.
- An object identifier column of a typed table cannot be updated (SQLSTATE 42808).
- Any enforced unique or primary key constraint that is defined on the table must be a superset of the distribution key (SQLSTATE 42997).
- The following rules only apply to multiple database partition databases.
  - Tables that are composed only of columns with types LOB, XML, a distinct type based on one of these types, or a structured type can only be created in table spaces that are defined on single-partition database partition groups.
  - The distribution key definition of a table in a table space that is defined on a multiple partition database partition group cannot be altered.
  - The distribution key column of a typed table must be the OID column.
  - Partitioned staging tables are not supported.
- For databases running in a Db2 pureScale environment, the ORGANIZE BY clause cannot be specified (SQLSTATE 42997).

- The following restrictions apply to range-clustered tables:
  - A range-clustered table cannot be specified in a Db2 pureScale environment (SQLSTATE 42997).
  - A clustering index cannot be created.
  - Altering the table to add a column is not supported.
  - Altering the table to change the data type of a column is not supported.
  - Altering the table to change PCTFREE is not supported.
  - Altering the table to set APPEND ON is not supported.
  - DETAILED statistics are not available.
  - The load utility cannot be used to populate the table.
  - Columns cannot be of type XML.
  - Cannot be created as a random distribution table.
- The following restrictions apply to random distribution tables:
  - Cannot be defined as a typed table
  - Cannot be defined as a range-clustered table
  - Cannot be defined as a materialized-query-table
  - Cannot be defined as a staging table
  - For random distribution tables that use the "random by" generation method (this happens when a random distribution table is created without a unique or primary key), the following additional restrictions apply:
    - Cannot be used as exception tables when constraints are checked in bulk, such as during load operations or during execution of the SET INTEGRITY statement
    - Cannot be used as an explain table
- A table is not protected unless it has a security policy associated with it and it includes either a column of type DB2SECURITYLABEL or a column defined with the SECURED WITH clause. The former indicates that the table is a protected table with **row level granularity** and the latter indicates that the table a protected table with **column level granularity**.
- Declaring a column of type DB2SECURITYLABEL fails if the table does not have a security policy associated with it (SQLSTATE 55064).
- A security policy cannot be added to a typed table (SQLSTATE 428DH), materialized query table, or staging table (SQLSTATE 428FG).
- An error tolerant *nested-table-expression* cannot be specified in the fullselect of a *materialized-query-definition* (SQLSTATE 428GG).
- When creating a materialized query table and any of the base tables it depends upon are protected with label-based access control, the following rules apply:
  - Row level security
    - Only one table in the materialized query table's fullselect can have a column type of DB2SECURITYLABEL (SQLSTATE 428FG).
    - The row security label column must be selected and referenced as a stand-alone column in the outermost SELECT list in the materialized query table definition (SQLSTATE 428FG). The corresponding column in the materialized query table will be marked as the row security label column.
  - Column level security
    - If a table involved in the materialized query table definition has a column that is protected with a security label, and that column appears in the materialized query table definition, that column's security label is inherited by the corresponding column in the materialized query table. See the examples in this topic for more details.

  - When creating a materialized query table that depends on one or more tables that are protected by label-based access control, all base tables must have the same security policy object (SQLSTATE, 428FG). The materialized query table is automatically protected with that security policy object.
  - The security label that is associated with a materialized query table column is computed as the aggregate of one or more security labels. This aggregate consists of the security labels that are associated with the base tables' columns that participate in the definition of that materialized query table column. The aggregate also consists of the security labels that are associated with any base table columns that appear in other parts of the materialized query table definition, such as the `WHERE`, `ORDER BY`, and `HAVING` clauses. The **ALTER SECURITY POLICY** has a description of how two security labels are aggregated. See the examples in this topic for more details.
  - When a staging table is created for a materialized query table that is protected with label-based access control, that staging table carries automatic protection like the materialized query table. See the examples in this topic for more details.
  - Label-based access control is enforced for direct access to a materialized query table just as it is enforced for a regular table. There are no differences from this perspective. When the SQL compiler services a query through a materialized query table, the label-based access control defined on the materialized query table itself does not need to be enforced. The SQL compiler uses the materialized query table which factors in the label-based access control rules from the appropriate base tables.
- The *isolation-clause* cannot be specified in the *full-select* of the *materialized-query-definition* (SQLSTATE 42601).
- Subselect statements that contain a *lock-request-clause* are not eligible for MQT routing.
- National character data types can be specified only in an MBCS database (SQLSTATE 560AA).
- The following restrictions apply to insert time clustering (ITC) tables:
  - ITC tables are not supported in an SMS table space (SQLSTATE 42838).
  - Indexes that are defined on ITC tables are not supported in an SMS table space (SQLSTATE 42838).

Notes

- Creating a table with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- If a foreign key is specified:
  - All packages with a delete usage on the parent table are invalidated.
  - All packages with an update usage on at least one column in the parent key are invalidated.
- Creating a subtable causes invalidation of all packages that depend on any table in table hierarchy.
- VARCHAR and VARGRAPHIC columns that are greater than 4,000 and 2,000 respectively should not be used as input parameters in functions in SYSFUN schema. Errors will occur when the function is invoked with an argument value that exceeds these lengths (SQLSTATE 22001).
- The use of NO ACTION or RESTRICT as delete or update rules for referential constraints determines when the constraint is enforced:

  RESTRICT
  The delete or update rule is enforced *before* all other constraints, including those referential constraints with modifying rules such as CASCADE or SET NULL.

  NO ACTION
  The delete or update rule is enforced *after* other referential constraints.

  One example where different behavior is evident involves the deletion of rows from a view that is defined as a UNION ALL of related tables.

```
Table T1 is a parent of table T3; delete rule as noted below.
Table T2 is a parent of table T3; delete rule CASCADE.

CREATE VIEW V1 AS SELECT * FROM T1 UNION ALL SELECT * FROM T2
```

```
DELETE FROM V1
```

If table T1 is a parent of table T3:

- With a delete rule of RESTRICT, a restrict violation (SQLSTATE 23001) is raised if t3 contains any child rows for parent keys of T1.
- With a delete rule of NO ACTION, the child rows might be deleted by the delete rule of CASCADE when deleting rows from T2 before the NO ACTION delete rule is enforced for the deletions from T1. If deletions from T2 did not result in the deletion of all child rows for parent keys of T1 in T3, then a constraint violation is raised (SQLSTATE 23504).

Note that the SQLSTATE returned is different depending on whether the delete or update rule is RESTRICT or NO ACTION.

- For tables in table spaces that are defined on multiple partition database partition groups, consider table collocation when choosing the distribution keys:
  - The tables must be in the same database partition group for collocation. The table spaces can be different, but must be defined in the same database partition group.
  - The distribution keys of the tables must have the same number of columns, and the corresponding key columns must be database partition-compatible for collocation.
  - The choice of distribution key also has an impact on performance of joins. If a table is frequently joined with another table, consider the joining columns as a distribution key for both tables.
- The NOT LOGGED INITIALLY option is useful for situations where a large result set needs to be created with data from an alternative source (another table or a file) and recovery of the table is not necessary. Using this option will save the overhead of logging the data. The following considerations apply when this option is specified:
  - When the unit of work is committed, all changes that were made to the table during the unit of work are flushed to disk.
  - When you run the rollforward utility and it encounters a log record that indicates that a table in the database was either populated by the Load utility or created with the NOT LOGGED INITIALLY option, the table will be marked as unavailable. The table will be dropped by the rollforward utility if it later encounters a DROP TABLE log. Otherwise, after the database is recovered, an error will be issued if any attempt is made to access the table (SQLSTATE 55019). The only operation that is permitted is to drop the table.
  - Once such a table is backed up as part of a database or table space backup, recovery of the table becomes possible.
- ***Use of materialized query tables to optimize query processing:*** The various types of materialized query tables use different controls to optimize the processing of queries.
  - A REFRESH DEFERRED materialized query table that is defined with ENABLE QUERY OPTIMIZATION can be used to optimize the processing of queries if each of the following conditions is true:
    - CURRENT REFRESH AGE is set to ANY.
    - CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION is set such that it includes the materialized query table type.
    - CURRENT QUERY OPTIMIZATION is set to 2 or a value greater than or equal to 5.

    Note: CURRENT REFRESH AGE does not affect query routing to MAINTAINED BY FEDERATED_TOOL materialized query tables.
  - A shadow table that is defined with ENABLE QUERY OPTIMIZATION can be used to optimize the processing of queries based on a replication latency threshold if each of the following conditions is true:
    - CURRENT REFRESH AGE is set to a duration other than zero or ANY.
    - CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION is set to contain only REPLICATION or ALL.

- CURRENT QUERY OPTIMIZATION is set to 2 or a value greater than or equal to 5.

For a description of the nonzero duration values that can be specified, see "SET CURRENT REFRESH AGE statement".

– A REFRESH IMMEDIATE materialized query table that is defined with ENABLE QUERY OPTIMIZATION is always considered for optimization if CURRENT QUERY OPTIMIZATION is set to 2 or a value greater than or equal to 5.

– For this optimization to be able to use a REFRESH DEFERRED materialized query table that is not maintained by replication or a REFRESH IMMEDIATE materialized query table, the fullselect must conform to certain rules in addition to those already described:

- The fullselect must not include any special registers or built-in functions that depend on the value of a special register.
- The fullselect must not include any global variables.
- The fullselect must not include functions that are not deterministic.

If the query that is specified when creating a materialized query table does not conform to these rules, a warning is returned (SQLSTATE 01633).

- If a materialized query table is defined with REFRESH IMMEDIATE, or a staging table is defined with PROPAGATE IMMEDIATE, it is possible for an error to occur when attempting to apply the change resulting from an insert, update, or delete operation on an underlying table. The error will cause the failure of the insert, update, or delete operation on the underlying table.
- Materialized query tables or staging tables cannot be used as exception tables when constraints are checked in bulk, such as during load operations or during execution of the SET INTEGRITY statement.
- Certain operations cannot be performed on a table that is referenced by a materialized query table that is defined with REFRESH IMMEDIATE, or defined with REFRESH DEFERRED with an associated staging table:
  – IMPORT REPLACE cannot be used.
  – ALTER TABLE NOT LOGGED INITIALLY WITH EMPTY TABLE cannot be done.
- In a federated system, nicknames for relational data sources or local tables can be used as the underlying tables to create a materialized query table. Nicknames for non-relational data sources are not supported. When a nickname is one of the underlying tables, the REFRESH DEFERRED option must be used. System-maintained materialized query tables that reference nicknames are not supported in a partitioned database environment.
- ***Considerations for transaction-start-ID columns:*** A transaction-start-ID column contains a null value if the column allows null values, and there is a row-begin column and the value of the column is unique from values for row-begin columns that are generated for other transactions. Because the column might contain null values, it is recommended that you use one of the following methods when retrieving a value from the column:

```
COALESCE ( transaction_start_id_col,  row_begin_col)

CASE WHEN transaction_start_id_col IS NOT NULL
          THEN transaction_start_id_col
     ELSE row_begin_col END
```

- ***Defining a system-period temporal table:*** A system-period temporal table definition includes the following:
  – A system period that is named SYSTEM_TIME, which is defined by using a row-begin column and a row-end column. See the descriptions of AS ROW BEGIN, AS ROW END, and period-definition.
  – A transaction-start-ID column. See the description of AS TRANSACTION START ID.
  – A system-period data versioning definition that is specified on a subsequent ALTER TABLE statement that specifies the ADD VERSIONING action, which includes the name of the associated history table. See the description of the ADD VERSIONING clause under ALTER TABLE.

To ensure that the history table cannot be implicitly dropped when a system-period temporal table is dropped, use the WITH RESTRICT ON DROP clause in the definition of the history table. A history table can manually be dropped only when the RESTRICT ON DROP attribute is removed by an ALTER TABLE statement.

- ***Defining an application-period temporal table:*** An application-period temporal table definition includes an application period named BUSINESS_TIME. The application period is defined using a begin time stamp column and an end column. See the description of period-definition.

  Data change operations on an application-period temporal table might result in an automatic insert of one or two extra rows when a row is updated or deleted. When an update or delete of a row in an application-period temporal table is specified for a portion of the period represented by that row, the row is updated or deleted and one or two rows are automatically inserted to represent the portion of the row that is not changed. New values are generated for each generated column in an application-period temporal table for each row that is automatically inserted as a result of an update or delete operation on the table. If a generated column is defined as part of a unique or primary key, parent key in a referential constraint, or unique index, it is possible that an automatic insert will violate a constraint or index in which case an error is returned.

- ***Considerations for implicitly hidden columns:*** Creating a table with implicitly hidden columns can impact the behavior of data movement utilities that are working with the table. When a table contains implicitly hidden columns, utilities like IMPORT, INGEST, and LOAD require that you specify whether data for the hidden columns is included in the operation. For example, this might mean that a load operation runs successfully against a table without any hidden columns, but fails when run against a table that contains implicitly hidden columns (SQLCODE SQL2437N). Similarly, EXPORT requires that you specify whether data for the hidden columns is included in the operation.

  Data movement utilities must use the DB2_DMU_DEFAULT registry variable, or the **implicitlyhiddeninclude** or **implicitlyhiddenmissing** file type modifiers when working with tables that contain implicitly hidden columns.

- ***Transparent DDL:*** In a federated system, a remote base table can be created, altered, or dropped using Db2 SQL. This capability is known as *transparent DDL*. Before a remote base table can be created on a data source, the federated server must be configured to access that data source. This configuration includes creating the wrapper for the data source, supplying the server definition for the server where the remote base table will be located, and creating the user mappings between the federated server and the data source.

  Transparent DDL does impose some limitations on what can be included in the CREATE TABLE statement:

  – Only columns and a primary key can be created on the remote base table.
  – Specific clauses that are supported by transparent DDL include:
    - *column-definition* and *unique-constraint* in the *element-list* clause
    - NOT NULL and PRIMARY KEY in the *column-options* clause
    - OPTIONS
  – The remote data source must support:
    - The remote column data types to which the database column data types are mapped
    - The primary key option in the CREATE TABLE statement

    Depending on how the data source responds to requests it does not support, an error might be returned or the request might be ignored.

  When a remote base table is created using transparent DDL, a nickname is automatically created for that remote base table.

- A referential constraint can be defined in such a way that either the parent table or the dependent table is a part of a table hierarchy. In such a situation, the effect of the referential constraint depends on the type of statement:

1. For an INSERT, UPDATE, or DELETE statement, the constraint ensures that, for each row of the dependent table (or any of its subtables) that has a non-null foreign key, a row exists in the parent table (or one of its subtables) with a matching parent key. This rule is enforced against any action that affects a row of either table, regardless of how that action is initiated.
2. For a DROP TABLE statement:
   - If the dropped table is the parent table or dependent table, the constraint is dropped.
   - If a supertable of the dropped table is the parent table, the rows of the dropped table are considered to be deleted from the supertable. The referential constraint is checked and its delete rule is invoked for each of the deleted rows.
   - If a supertable of the dropped table is the dependent table, the constraint is not checked. Deletion of a row from a dependent table cannot result in violation of a referential constraint.

- ***Privileges:*** When any table is created, the definer of the table is granted CONTROL privilege. When a subtable is created, the SELECT privilege that each user or group has on the immediate supertable is automatically granted on the subtable with the table definer as the grantor.
- ***Row size limit:*** The maximum number of bytes allowed in the row of a row-organized table is dependent on the page size of the table space in which the table is created (*tablspace-name1*). The following table shows the row size limit and number of columns limit associated with each table space page size.

Table 139. Limits for Number of Columns and Row Size in Each Table Space Page Size (row-organized tables)

Page Size	Row Size Limit	Column Count Limit
4K	4005	500
8K	8101	1012
16K	16,293	1012
32K	32,677	1012

The actual number of columns for a row-organized table can be further limited by the following formula:

```
Total Columns * 8 + Number of LOB Columns * 12 <=
  Row Size Limit for Page Size
```

A column-organized table can have a maximum of 1012 columns, regardless of page size, where the byte counts of the columns must not be greater than 32,677.

- ***Byte counts:*** The following table contains the byte counts of columns by data type. This is used to calculate the row size. The byte counts depend on whether VALUE COMPRESSION is active. When VALUE COMPRESSION is not active, the byte counts also depend on whether the column is nullable. The byte counts shown apply when row compression is not enabled. If row compression is active, the total number of bytes used by a row will generally be smaller than for an uncompressed version of the row; it will never be larger.

  If a table is based on a structured type, an additional 4 bytes of overhead is reserved to identify rows of subtables, regardless of whether subtables are defined. Additional subtable columns must be considered nullable for byte count purposes, even if defined as not nullable.

Table 140. Byte Counts of Columns by Data Type

Data type	VALUE COMPRESSION is active[1]	VALUE COMPRESSION is not active	
		Column is nullable	Column is not nullable
SMALLINT	4	3	2
INTEGER	6	5	4
BIGINT	10	9	8

Table 140. Byte Counts of Columns by Data Type (continued)

Data type	VALUE COMPRESSION is active[1]	VALUE COMPRESSION is not active	
		Column is nullable	Column is not nullable
REAL	6	5	4
DOUBLE	10	9	8
DECIMAL	The integral part of (*p*/2)+3, where *p* is the precision	The integral part of (*p*/2)+2, where *p* is the precision	The integral part of (*p*/2)+1, where *p* is the precision
DECFLOAT(16)	10	9	8
DECFLOAT(34)	18	17	16
CHAR(*n*)	*n*+2	*n*+1	*n*
VARCHAR(*n*)	*n*+2	*n*+5 (within a table)	*n*+4 (within a table)
LONG VARCHAR[2]	22	25	24
BINARY	*n*+2	*n*+1	*n*
VARBINARY	*n*+2	*n*+5 (within a table)	*n*+4 (within a table)
GRAPHIC(*n*)	*n**2+2	*n**2+1	*n**2
VARGRAPHIC(*n*)	*n**2+2	*n**2+5 (within a table)	*n**2+4 (within a table)
LONG VARGRAPHIC[2]	22	25	24
DATE	6	5	4
TIME	5	4	3
TIMESTAMP(*p*)	The integral part of (*p*+1)/2+9, where *p* is the precision of fractional seconds	The integral part of (*p*+1)/2+8, where *p*is the precision of fractional seconds	The integral part of (*p*+1)/2+7, where *p*is the precision of fractional seconds
BOOLEAN	3	2	1
XML (without INLINE LENGTH specified)	82	85	84
XML (with INLINE LENGTH specified)	INLINE LENGTH +2	INLINE LENGTH +4	INLINE LENGTH +3
Maximum LOB[3] length 1024 (without INLINE LENGTH specified)	70	73	72
Maximum LOB length 8192 (without INLINE LENGTH specified)	94	97	96
Maximum LOB length 65,536 (without INLINE LENGTH specified)	118	121	120
Maximum LOB length 524,000 (without INLINE LENGTH specified)	142	145	144

Table 140. Byte Counts of Columns by Data Type (continued)

Data type	VALUE COMPRESSION is active[1]	VALUE COMPRESSION is not active	
		Column is nullable	Column is not nullable
Maximum LOB length 4,190,000 (without INLINE LENGTH specified)	166	169	168
Maximum LOB length 134,000,000 (without INLINE LENGTH specified)	198	201	200
Maximum LOB length 536,000,000 (without INLINE LENGTH specified)	222	225	224
Maximum LOB length 1,070,000,000 (without INLINE LENGTH specified)	254	257	256
Maximum LOB length 1,470,000,000 (without INLINE LENGTH specified)	278	281	280
Maximum LOB length 2,147,483,647 (without INLINE LENGTH specified)	314	317	316
LOB with INLINE LENGTH specified	INLINE LENGTH + 2	INLINE LENGTH + 5	INLINE LENGTH + 4

[1] There is an additional 2 bytes of storage used by each row when VALUE COMPRESSION is active for that row.

[2]The LONG VARCHAR and LONG VARGRAPHIC data types are deprecated and might be removed in a future release.

[3] Each LOB value has a *LOB descriptor* in the base record that points to the location of the actual value. The size of the descriptor varies according to the maximum length defined for the column. When INLINE LENGTH is not specified for a LOB column, the size of the descriptor is used as the default inline length value.

When determining the byte counts for LOB columns, there are extra bytes to consider when a LOB column is part of a system temporary table that might get generated for insensitive cursors, scrollable cursors, and other queries that require temporary space or sorting of data. The number of extra bytes required might go as high as 70 bytes, depending on the specific query. If the base table is close to the maximum row length for the pagesize, an error might be returned when processing a query if the system temporary table cannot fit in the largest available system temporary table space. If an existing system temporary table space is available that has a 32K page size, then extended row size support is used where possible.

For a *distinct type*, the byte count is equivalent to the length of the source type of the distinct type. For a *reference type*, the byte count is equivalent to the length of the built-in data type on which the reference type is based. For a *structured type*, the byte count is equivalent to the INLINE LENGTH + 4. The INLINE LENGTH is the value specified (or implicitly calculated) for the column in the *column-options* clause.

The row sizes for the following sample tables assume that VALUE COMPRESSION is not specified:

```
DEPARTMENT 63 (0 + 3 + 33 + 7 +  3 + 17)
ORG        57 (0 + 3 + 19 + 2 + 15 + 18)
```

If VALUE COMPRESSION were to be specified, the row sizes would change to:

```
DEPARTMENT 69 (2 + 5 + 31 + 8 +  5 + 18)
ORG        53 (2 + 4 + 16 + 4 + 12 + 15)
```

Minimum page size requirements for a table with extended row size : When a data row is inserted or updated in a table with extended row size support and the physical data row length exceeds the maximum record length for the table space, a subset of the varying length string columns (VARCHAR or VARGRAPHIC) is stored as large object (LOB) data outside of the data row. The table column in the base row is replaced by a descriptor that is 24 bytes in size. In order to accommodate the extreme case where all VARCHAR or VARGRAPHIC data is stored outside of the data row, the database manager computes the minimum row size using the following method:

– Handles every VARCHAR(n) column where n > 24 as if it were VARCHAR(24)

– Handles every VARGRAPHIC(m) column where m > 12 as if it were VARGRAPHIC(12

The value is computed using the *Byte Counts of Columns by Data Type* table. The computed result is then used to find the lower bound of the page size where the table with extended row size can be created.

- ***Storage byte counts:*** The following tables describe the storage byte counts of columns by data type for data values.

  The first table defines the sets of attributes. Those attributes are referenced in the second table, which contains the details for the byte counts for each data type.

  The byte counts depend on whether VALUE COMPRESSION is active. When VALUE COMPRESSION is not active, the byte counts also depend on whether the column is nullable. The values in the table represent the amount of storage (in bytes) that is used to store the value. The byte counts shown apply when row compression is not enabled. If row compression is active, the total number of bytes used by a row will generally be smaller than for an uncompressed version of the row; it will never be larger.

Table 141. Definitions of the criteria referenced in the related table

Case	Data value	VALUE COMPRESSION	Column nullability
A	NULL	Not active	Nullable
B	NULL	Active [2]	Nullable
C	Zero-length	Active [2]	Not applicable
D	System default[1]	Active [2]	Not applicable
E	All other data values	Not active	Nullable
F	All other data values	Not active	Not nullable
G	All other data values	Active [2]	Not applicable

[1] When COMPRESS SYSTEM DEFAULT is specified for the column.

[2] There is an additional 2 bytes of storage used by each row when VALUE COMPRESSION is active for that row.

Table 142. Storage Byte Counts Based on Row Format, Data Type, and Data Value

Data type	Case A	Case B	Case C	Case D	Case E	Case F	Case G
SMALLINT	3	3	-	3	3	2	4
INTEGER	5	3	-	3	5	4	6
BIGINT	9	3	-	3	9	8	10
REAL	5	3	-	3	5	4	6
DOUBLE	9	3	-	3	9	8	10
DECIMAL	The integral part of (*p*/2)+2, where *p* is the precision	3	-	3	The integral part of (*p*/2)+2, where *p* is the precision	The integral part of (*p*/2)+1, where *p* is the precision	The integral part of (*p*/2)+3, where *p* is the precision

Table 142. Storage Byte Counts Based on Row Format, Data Type, and Data Value (continued)

Data type	Case A	Case B	Case C	Case D	Case E	Case F	Case G
DECFLOAT(16)	9	3	-	3	9	8	10
DECFLOAT(34)	17	3	-	3	17	16	18
CHAR(*n*)	*n*+1	3	-	3	*n*+1	*n*	*n*+2
VARCHAR(*n*)	5	3	2	2	*N*+5, where *N* is the number of bytes in the data	*N*+4, where *N* is the number of bytes in the data	*N*+2, where *N* is the number of bytes in the data
LONG VARCHAR[2]	5	3	2	2	25	24	22
BINARY	*n*+1	3	-	3	*n*+1	*n*	*n*+2
VARBINARY	5	3	2	2	*N*+5, where *N* is the number of bytes in the data	*N*+4, where *N* is the number of bytes in the data	*N*+2, where *N* is the number of bytes in the data
GRAPHIC(*n*)	*n**2+1	3	-	3	*n**2+1	*n**2	*n**2+2
VARGRAPHIC(*n*)	5	3	2	2	*N**2+5, where *N* is the number of bytes in the data	*N**2+4, where *N* is the number of bytes in the data	*N**2+2, where *N* is the number of bytes in the data
LONG VARGRAPHIC[2]	5	3	2	2	25	24	22
DATE	5	3	-	-	5	4	6
TIME	4	3	-	-	4	3	5
TIMESTAMP(*p*)	The integral part of (*p*+1)/2+8, where *p* is the precision of fractional seconds	3	-	-	The integral part of (*p*+1)/2+8, where *p* is the precision of fractional seconds	The integral part of (*p*+1)/2+7, where *p* is the precision of fractional seconds	The integral part of (*p*+1)/2+9, where *p* is the precision of fractional seconds
BOOLEAN	2	2	-	2	2	1	3
Maximum LOB[1] length 1024	5	3	2	2	(60 to 68)+5	(60 to 68)+4	(60 to 68)+2
Maximum LOB length 8192	5	3	2	2	(60 to 92)+5	(60 to 92)+4	(60 to 92)+2
Maximum LOB length 65,536	5	3	2	2	(60 to 116)+5	(60 to 116)+4	(60 to 116)+2
Maximum LOB length 524,000	5	3	2	2	(60 to 140)+5	(60 to 140)+4	(60 to 140)+2
Maximum LOB length 4,190,000	5	3	2	2	(60 to 164)+5	(60 to 164)+4	(60 to 164)+2
Maximum LOB length 134,000,000	5	3	2	2	(60 to 196)+5	(60 to 196)+4	(60 to 196)+2
Maximum LOB length 536,000,000	5	3	2	2	(60 to 220)+5	(60 to 220)+4	(60 to 220)+2
Maximum LOB length 1,070,000,000	5	3	2	2	(60 to 252)+5	(60 to 252)+4	(60 to 252)+2
Maximum LOB length 1,470,000,000	5	3	2	2	(60 to 276)+5	(60 to 276)+4	(60 to 276)+2
Maximum LOB length 2,147,483,647	5	3	2	2	(60 to 312)+5	(60 to 312)+4	(60 to 312)+2

Table 142. Storage Byte Counts Based on Row Format, Data Type, and Data Value (continued)

Data type	Case A	Case B	Case C	Case D	Case E	Case F	Case G
XML	5	3	-	-	85	84	82

1 When COMPRESS SYSTEM DEFAULT is specified for the column.

2 The LONG VARCHAR and LONG VARGRAPHIC data types are deprecated and might be removed in a future release.

- ***Dimension columns:*** Because each distinct value of a dimension column is assigned to a different block of the table, clustering on an expression might be desirable, such as "INTEGER(ORDER_DATE)/100". In this case, a generated column can be defined for the table, and this generated column can then be used in the ORGANIZE BY DIMENSIONS clause. If the expression is monotonic with respect to a column of the table, the database might use the dimension index to satisfy range predicates on that column. For example, if the expression is simply *column-name* + *some-positive-constant*, it is monotonic increasing. User-defined functions, certain built-in functions, and using more than one column in an expression, prevent monotonicity or its detection.

  Dimensions involving generated columns whose expressions are non-monotonic, or whose monotonicity cannot be determined, can still be created, but range queries along slice or cell boundaries of these dimensions are not supported. Equality and IN predicates *can* be processed by slices or cells.

  A generated column is monotonic if the following is true with respect to the generating function, fn:

  – Monotonic increasing.

    For every possible pair of values x1 and x2, if x2>x1, then fn(x2)>fn(x1). For example:

    ```
    SALARY - 10000
    ```

  – Monotonic decreasing.

    For every possible pair of values x1 and x2, if x2>x1, then fn(x2)<fn(x1). For example:

    ```
    -SALARY
    ```

  – Monotonic non-decreasing.

    For every possible pair of values x1 and x2, if x2>x1, then fn(x2)>=fn(x1). For example:

    ```
    SALARY/1000
    ```

  – Monotonic non-increasing.

    For every possible pair of values x1 and x2, if x2>x1, then fn(x2)<=fn(x1). For example:

    ```
    -SALARY/1000
    ```

  The expression "PRICE*DISCOUNT" is not monotonic, because it involves more than one column of the table.

- ***Range-clustered tables:*** Organizing a table by key sequence is effective for certain types of tables. The table should have an integer key that is tightly clustered (dense) over the range of possible values. The columns of this integer key must not be nullable, and the key should logically be the primary key of the table. The organization of a range-clustered table precludes the need for a separate unique index object, providing direct access to the row for a specified key value, or a range of rows for a specified range of key values. The allocation of all the space for the complete set of rows in the defined key sequence range is done during table creation, and must be considered when defining a range-clustered table. The storage space is not available for any other use, even though the rows are initially marked deleted. If the full key sequence range will be populated with data only over a long period of time, this table organization might not be appropriate.
- A table can have at most one security policy.
- Referential integrity constraints that are defined on protected tables are enforced. Constraints violations in this case can be difficult to debug, because the database manager will not allow you

to see what row has caused a violation if you do not have the appropriate security label or exemptions credentials.

- When defining the order of columns in a table, frequently updated columns should be placed at the end of the definition to minimize the amount of data logged for updates. This includes ROW CHANGE TIMESTAMP columns. ROW CHANGE TIMESTAMP columns are guaranteed to be updated on each row update.
- ***Security and replication:*** Replication can cause data rows from a protected table to be replicated outside of the database. Care must be taken when setting up replication for a protected table, because data that is outside of the database cannot be protected.
- ***Considerations for a multi-partition or Db2 pureScale environment***:
  - If the CACHE and NO ORDER options are in effect, multiple caches can be active simultaneously. This can happen at each member in a multi-partition or Db2 pureScale environment. The requests for next value assignments from different members might not result in the assignment of values in strict numeric order. Assume, for example, in a multi-partition or Db2 pureScale environment, that members DB1A and DB1B are using the same sequence, and DB1A gets the cache values in the range 1 - 20 and DB1B gets the cache values in the range 21 - 40. In this scenario, if DB1A requested the next value first, then DB1B requested, and then DB1A requested again, the actual order of values assigned would be 1,21,2. Therefore, to guarantee that sequence numbers are generated in strict numeric order among multiple members using the same sequence concurrently, specify the ORDER option.
  - In a Db2 pureScale environment, using the ORDER or NO CACHE option ensures that the values assigned to a sequence which is shared by applications across multiple members are in strict numeric order. In a Db2 pureScale environment, if ORDER is specified, then NO CACHE is implied even if CACHE *n* is specified
- ***Considerations for row and column access control (RCAC)***: The ACTIVATE ROW ACCESS CONTROL, ACTIVATE COLUMN ACCESS CONTROL, DEACTIVATE ROW ACCESS CONTROL, and DEACTIVATE COLUMN ACCESS CONTROL clauses are not supported. Use the ALTER TABLE statement to activate or deactivate row or column level access control on a table.
- ***Considerations for column-organized tables:*** Create column-organized tables in automatic storage table spaces only.

  The following options are not supported for column-organized tables (underlined options are defaults). They can, however, be specified for row-organized tables that will be used in the same database and workloads as column-organized tables.

  - ORGANIZE BY {DIMENSIONS | KEY SEQUENCE | INSERT TIME}
  - DATA CAPTURE CHANGES
  - VALUE COMPRESSION
  - COMPRESS YES [ADAPTIVE | STATIC]
  - COMPRESS NO
  - PARTITION BY RANGE
  - FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP
  - CREATE TABLE OF <type-name1> (to create a typed table)
  - PROPAGATE IMMEDIATE
  - CHECK
  - DETERMINED BY

  Structured type columns are not supported.

  The columns of a column-organized table must have one of the following data types:

  - SMALLINT
  - INTEGER
  - BIGINT

  - DECIMAL
  - REAL
  - DOUBLE
  - DECFLOAT
  - CHAR (including FOR BIT DATA)
  - VARCHAR (including FOR BIT DATA)
  - BINARY
  - VARBINARY
  - GRAPHIC
  - VARGRAPHIC
  - DATE
  - TIME
  - TIMESTAMP (*n*)
  - BOOLEAN
  - CLOB
  - BLOB
  - DBCLOB
  - NCLOB
  - Distinct types of a supported data type

Syntax alternatives

The following alternatives are non-standard. They are supported for compatibility with earlier product versions or with other database products.

- The following syntax is accepted as the default behavior:
  - IN database-name.tablespace-name
  - IN DATABASE database-name
  - FOR MIXED DATA
  - FOR SBCS DATA
- PART can be specified in place of PARTITION.
- PARTITION *partition-number* can be specified instead of PARTITION *partition-name*. A *partition-number* must not identify a partition that was previously specified in the CREATE TABLE statement. If a *partition-number* is not specified, a unique partition number is generated by the database manager.
- VALUES can be specified in place of ENDING AT.
- The CONSTRAINT keyword can be omitted from a *column-definition* defining a references-clause.
- *constraint-name* can be specified following FOREIGN KEY (without the CONSTRAINT keyword).
- SUMMARY can optionally be specified after CREATE.
- DEFINITION ONLY can be specified in place of WITH NO DATA.
- PARTITIONING KEY can be specified in place of DISTRIBUTE BY.
- DISTRIBUTE ON can be specified in place of DISTRIBUTE BY when it is followed by the HASH option, but not when it is followed by the REPLICATION option.
- REPLICATED can be specified in place of DISTRIBUTE BY REPLICATION
- A comma can be used to separate multiple options in the *identity-options* clause.
- NOMINVALUE, NOMAXVALUE, NOCYCLE, NOCACHE, and NOORDER can be specified in place of NO MINVALUE, NO MAXVALUE, NO CYCLE, NO CACHE, and NO ORDER, respectively.

- ADD can be specified before *table-option-name string-constant*.
- When specifying the value of the datetime special register, NOW() can be specified in place of CURRENT_TIMESTAMP.

Examples

1. Create table TDEPT in the DEPARTX table space. DEPTNO, DEPTNAME, MGRNO, and ADMRDEPT are column names. CHAR means the column will contain character data. NOT NULL means that the column cannot contain a null value. VARCHAR means the column will contain varying-length character data. The primary key consists of the column DEPTNO.

```
CREATE TABLE TDEPT
  (DEPTNO   CHAR(3)     NOT NULL,
   DEPTNAME VARCHAR(36) NOT NULL,
   MGRNO    CHAR(6),
   ADMRDEPT CHAR(3)     NOT NULL,
   PRIMARY KEY(DEPTNO))
IN DEPARTX
```

2. Create table PROJ in the SCHED table space. PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTAFF, PRSTDATE, PRENDATE, and MAJPROJ are column names. CHAR means the column will contain character data. DECIMAL means the column will contain packed decimal data. 5,2 means the following: 5 indicates the number of decimal digits, and 2 indicates the number of digits to the right of the decimal point. NOT NULL means that the column cannot contain a null value. VARCHAR means the column will contain varying-length character data. DATE means the column will contain date information in a three-part format (year, month, and day).

```
CREATE TABLE PROJ
  (PROJNO   CHAR(6)      NOT NULL,
   PROJNAME VARCHAR(24)  NOT NULL,
   DEPTNO   CHAR(3)      NOT NULL,
   RESPEMP  CHAR(6)      NOT NULL,
   PRSTAFF  DECIMAL(5,2)         ,
   PRSTDATE DATE                 ,
   PRENDATE DATE                 ,
   MAJPROJ  CHAR(6)      NOT NULL)
IN SCHED
```

3. Create a table called EMPLOYEE_SALARY where any unknown salary is considered 0. No table space is specified, so that the table will be created in a table space selected by the system based on the rules described for the *IN tablespace-name* clause.

```
CREATE TABLE EMPLOYEE_SALARY
  (DEPTNO   CHAR(3)      NOT NULL,
   DEPTNAME VARCHAR(36)  NOT NULL,
   EMPNO    CHAR(6)      NOT NULL,
   SALARY   DECIMAL(9,2) NOT NULL WITH DEFAULT)
```

4. Create distinct types for total salary and miles and use them for columns of a table created in the default table space. In a dynamic SQL statement assume the CURRENT SCHEMA special register is JOHNDOE and the CURRENT PATH is the default ("SYSIBM", "SYSFUN", "JOHNDOE").

   If a value for SALARY is not specified it must be set to 0 and if a value for LIVING_DIST is not specified it must to set to 1 mile.

```
CREATE TYPE JOHNDOE.T_SALARY AS INTEGER

CREATE TYPE JOHNDOE.MILES  AS FLOAT

CREATE TABLE EMPLOYEE
  (ID          INTEGER NOT NULL,
   NAME        CHAR (30),
   SALARY      T_SALARY NOT NULL WITH DEFAULT,
   LIVING_DIST MILES    DEFAULT MILES(1) )
```

5. Create distinct types for image and audio and use them for columns of a table. No table space is specified, so that the table will be created in a table space selected by the system based on the rules described for the IN *tablespace-name* clause. Assume the CURRENT PATH is the default.

```
CREATE TYPE IMAGE AS BLOB (10M)

CREATE TYPE AUDIO AS BLOB (1G)

CREATE TABLE PERSON
  (SSN    INTEGER NOT NULL,
   NAME   CHAR (30),
   VOICE  AUDIO,
   PHOTO  IMAGE)
```

6. Create table EMPLOYEE in the HUMRES table space. The constraints defined on the table are the following:
   - The values of department number must lie in the range 10 to 100.
   - The job of an employee can only be either "Sales", "Mgr", or "Clerk".
   - Every employee that has been with the company since 1986 must make more than $40,500.

   Note: If the columns included in the check constraints are nullable they could also be NULL.

```
CREATE TABLE EMPLOYEE
  (ID          SMALLINT NOT NULL,
   NAME        VARCHAR(9),
   DEPT        SMALLINT CHECK (DEPT BETWEEN 10 AND 100),
   JOB         CHAR(5) CHECK (JOB IN ('Sales','Mgr','Clerk')),
   HIREDATE    DATE,
   SALARY      DECIMAL(7,2),
   COMM        DECIMAL(7,2),
   PRIMARY KEY (ID),
   CONSTRAINT  YEARSAL CHECK (YEAR(HIREDATE) > 1986
     OR SALARY > 40500)
  )
  IN HUMRES
```

7. Create a table that is wholly contained in the PAYROLL table space.

```
CREATE TABLE EMPLOYEE .....
  IN PAYROLL
```

8. Create a table with its data part in ACCOUNTING and its index part in ACCOUNT_IDX.

```
CREATE TABLE SALARY.....
  IN ACCOUNTING INDEX IN ACCOUNT_IDX
```

9. Create a table and log SQL changes in the default format.

```
CREATE TABLE SALARY1 .....
```

   or

```
CREATE TABLE SALARY1 .....
  DATA CAPTURE NONE
```

10. Create a table and log SQL changes in an expanded format.

```
CREATE TABLE SALARY2 .....
  DATA CAPTURE CHANGES
```

11. Create a table EMP_ACT in the SCHED table space. EMPNO, PROJNO, ACTNO, EMPTIME, EMSTDATE, and EMENDATE are column names. Constraints defined on the table are:
    - The value for the set of columns, EMPNO, PROJNO, and ACTNO, in any row must be unique.

- The value of PROJNO must match an existing value for the PROJNO column in the PROJECT table and if the project is deleted all rows referring to the project in EMP_ACT should also be deleted.

```
CREATE TABLE EMP_ACT
  (EMPNO       CHAR(6) NOT NULL,
   PROJNO      CHAR(6) NOT NULL,
   ACTNO       SMALLINT NOT NULL,
   EMPTIME     DECIMAL(5,2),
   EMSTDATE    DATE,
   EMENDATE    DATE,
   CONSTRAINT EMP_ACT_UNIQ UNIQUE (EMPNO,PROJNO,ACTNO),
   CONSTRAINT FK_ACT_PROJ FOREIGN KEY (PROJNO)
                          REFERENCES PROJECT (PROJNO) ON DELETE CASCADE
  )
  IN SCHED
```

A unique index called EMP_ACT_UNIQ is automatically created in the same schema to enforce the unique constraint.

12. Create a table that is to hold information about famous goals for the ice hockey hall of fame. The table will list information about the player who scored the goal, the goaltender against who it was scored, the date, and a description. The description column is nullable.

```
CREATE TABLE HOCKEY_GOALS
  ( BY_PLAYER      VARCHAR(30)   NOT NULL,
    BY_TEAM        VARCHAR(30)   NOT NULL,
    AGAINST_PLAYER VARCHAR(30)   NOT NULL,
    AGAINST_TEAM   VARCHAR(30)   NOT NULL,
    DATE_OF_GOAL   DATE          NOT NULL,
    DESCRIPTION    CLOB(5000) )
```

13. Suppose an exception table is needed for the EMPLOYEE table. One can be created using the following statement.

```
CREATE TABLE EXCEPTION_EMPLOYEE AS
  (SELECT EMPLOYEE.*,
    CURRENT TIMESTAMP AS TIMESTAMP,
    CAST ('' AS CLOB(32K)) AS MSG
  FROM EMPLOYEE
  ) WITH NO DATA
```

14. Given the following table spaces with the indicated attributes:

```
TBSPACE            PAGESIZE    USER   USERAUTH
------------------ ----------- ------ --------
DEPT4K                    4096 BOBBY  Y
PUBLIC4K                  4096 PUBLIC Y
DEPT8K                    8192 BOBBY  Y
DEPT8K                    8192 RICK   Y
PUBLIC8K                  8192 PUBLIC Y
```

- If RICK creates the following table, it is placed in table space PUBLIC4K since the byte count is less than 4005; but if BOBBY creates the same table, it is placed in table space DEPT4K, since BOBBY has USE privilege because of an explicit grant:

```
CREATE TABLE DOCUMENTS
  (SUMMARY    VARCHAR(1000),
   REPORT     VARCHAR(2000))
```

- If BOBBY creates the following table, it is placed in table space DEPT8K since the byte count is greater than 4005, and BOBBY has USE privilege because of an explicit grant. However, if DUNCAN creates the same table, it is placed in table space PUBLIC8K, since DUNCAN has no specific privileges:

```
CREATE TABLE CURRICULUM
  (SUMMARY    VARCHAR(1000),
   REPORT     VARCHAR(2000),
   EXERCISES  VARCHAR(1500))
```

15. Create a table with a LEAD column defined with the structured type EMP. Specify an INLINE LENGTH of 300 bytes for the LEAD column, indicating that any instances of LEAD that cannot fit within the 300

bytes are stored outside the table (separately from the base table row, similar to the way LOB values are handled).

```
CREATE TABLE PROJECTS (PID INTEGER,
  LEAD EMP INLINE LENGTH 300,
  STARTDATE DATE,
            ...)
```

16. Create a table DEPT with five columns named DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, and LOCATION. Column DEPT is to be defined as an IDENTITY column so that a value will always be generated for it. The values for the DEPT column should begin with 500 and increment by 1.

```
CREATE TABLE DEPT
  (DEPTNO     SMALLINT      NOT NULL
                 GENERATED ALWAYS AS IDENTITY
                   (START WITH 500, INCREMENT BY 1),
   DEPTNAME   VARCHAR(36)   NOT NULL,
   MGRNO      CHAR(6),
   ADMRDEPT   SMALLINT      NOT NULL,
   LOCATION   CHAR(30))
```

17. Create a SALES table that is distributed on the YEAR column, and that has dimensions on the REGION and YEAR columns. Data will be distributed across database partitions according to hashed values of the YEAR column. On each database partition, data will be organized into extents based on unique combinations of values of the REGION and YEAR columns on those database partitions.

```
CREATE TABLE SALES
  (CUSTOMER   VARCHAR(80),
   REGION     CHAR(5),
   YEAR       INTEGER)
DISTRIBUTE BY HASH (YEAR)
ORGANIZE BY DIMENSIONS (REGION, YEAR)
```

18. Create a SALES table with a PURCHASEYEARMONTH column that is generated from the PURCHASEDATE column. Use an expression to create a column that is monotonic with respect to the original PURCHASEDATE column, and is therefore suitable for use as a dimension. The table is distributed on the REGION column, and organized within each database partition into extents according to the PURCHASEYEARMONTH column; that is, different regions will be on different database partitions, and different purchase months will belong to different cells (or sets of extents) within those database partitions.

```
CREATE TABLE SALES
  (CUSTOMER            VARCHAR(80),
   REGION              CHAR(5),
   PURCHASEDATE        DATE,
   PURCHASEYEARMONTH   INTEGER
                          GENERATED ALWAYS AS (INTEGER(PURCHASEDATE)/100))
DISTRIBUTE BY HASH (REGION)
ORGANIZE BY DIMENSIONS (PURCHASEYEARMONTH)
```

19. Create a CUSTOMER table with a CUSTOMERNUMDIM column that is generated from the CUSTOMERNUM column. Use an expression to create a column that is monotonic with respect to the original CUSTOMERNUM column, and is therefore suitable for use as a dimension. The table is organized into cells according to the CUSTOMERNUMDIM column, so that there is a different cell in the table for every 50 customers. If a unique index were created on CUSTOMERNUM, customer numbers would be clustered in such a way that each set of 50 values would be found in a particular set of extents in the table.

```
CREATE TABLE CUSTOMER
  (CUSTOMERNUM      INTEGER,
   CUSTOMERNAME     VARCHAR(80),
   ADDRESS          VARCHAR(200),
   CITY             VARCHAR(50),
   COUNTRY          VARCHAR(50),
   CODE             VARCHAR(15),
   CUSTOMERNUMDIM   INTEGER
                       GENERATED ALWAYS AS (CUSTOMERNUM/50))
ORGANIZE BY DIMENSIONS (CUSTOMERNUMDIM)
```

20. Create a remote base table called EMPLOYEE on the Oracle server, ORASERVER. A nickname, named EMPLOYEE, which refers to this newly created remote base table, will also automatically be created.

```
CREATE TABLE EMPLOYEE
  (EMP_NO       CHAR(6)       NOT NULL,
   FIRST_NAME   VARCHAR(12)   NOT NULL,
   MID_INT      CHAR(1)       NOT NULL,
   LAST_NAME    VARCHAR(15)   NOT NULL,
   HIRE_DATE    DATE,
   JOB          CHAR(8),
   SALARY       DECIMAL(9,2),
   PRIMARY KEY (EMP_NO))
OPTIONS
  (REMOTE_SERVER 'ORASERVER',
   REMOTE_SCHEMA 'J15USER1',
   REMOTE_TABNAME 'EMPLOYEE')
```

The following CREATE TABLE statements show how to specify the table name, or the table name and the explicit remote base table name, to get the required case. The lowercase identifier, employee, is used to illustrate the implicit folding of identifiers.

Create a remote base table called EMPLOYEE (uppercase characters) on an Informix server, and create a nickname named EMPLOYEE (uppercase characters) on that table:

```
CREATE TABLE employee
  (EMP_NO CHAR(6) NOT NULL,
   ...)
OPTIONS
  (REMOTE_SERVER 'INFX_SERVER')
```

If the REMOTE_TABNAME option is not specified, and *table-name* is not delimited, the remote base table name will be in uppercase characters, even if the remote data source normally stores names in lowercase characters.

Create a remote base table called employee (lowercase characters) on an Informix server, and create a nickname named EMPLOYEE (uppercase characters) on that table:

```
CREATE TABLE employee
  (EMP_NO CHAR(6) NOT NULL,
   ...)
OPTIONS
  (REMOTE_SERVER 'INFX_SERVER',
   REMOTE_TABNAME 'employee')
```

When creating a table at a remote data source that supports delimited identifiers, use the REMOTE_TABNAME option and a character string constant that specifies the table name in the required case.

Create a remote base table called employee (lowercase characters) on an Informix server, and create a nickname named employee (lowercase characters) on that table:

```
CREATE TABLE "employee"
  (EMP_NO CHAR(6) NOT NULL,
   ...)
OPTIONS
  (REMOTE_SERVER 'INFX_SERVER')
```

If the REMOTE_TABNAME option is not specified, and *table-name* is delimited, the remote base table name will be identical to *table-name*.

21. Create a range-clustered table that can be used to locate a student using a student ID. For each student record, include the school ID, program ID, student number, student ID, student first name, student last name, and student grade point average (GPA).

```
CREATE TABLE STUDENTS
  (SCHOOL_ID     INTEGER   NOT NULL,
   PROGRAM_ID    INTEGER   NOT NULL,
   STUDENT_NUM   INTEGER   NOT NULL,
   STUDENT_ID    INTEGER   NOT NULL,
   FIRST_NAME    CHAR(30),
   LAST_NAME     CHAR(30),
```

```
    GPA           DOUBLE)
  ORGANIZE BY KEY SEQUENCE
    (STUDENT_ID
      STARTING FROM 1
      ENDING AT 1000000)
    DISALLOW OVERFLOW
```

The size of each record is the sum of the columns, plus alignment, plus the range-clustered table row header. In this case, the row size is 98 bytes: 4 + 4 + 4 + 4 + 30 + 30 + 8 + 3 (for nullable columns) + 1 (for alignment) + 10 (for the header). With a 4-KB page size (or 4096 bytes), after accounting for page overhead, there are 4038 bytes available, enough room for 41 records per page. Allowing for 1 million student records, there is a need for (1 million divided by 41 records per page) 24,391 pages. With two additional pages for table overhead, the final number of 4-KB pages that are allocated when the table is created is 24,393.

22. Create a table named DEPARTMENT with a functional dependency that has no specified constraint name.

```
CREATE TABLE DEPARTMENT
  (DEPTNO     SMALLINT      NOT NULL,
   DEPTNAME   VARCHAR(36)   NOT NULL,
   MGRNO      CHAR(6),
   ADMRDEPT   SMALLINT      NOT NULL,
   LOCATION   CHAR(30),
  CHECK (DEPTNAME DETERMINED BY DEPTNO) NOT ENFORCED)
```

23. Create a table with protected rows.

```
CREATE TABLE TOASTMASTERS
  (PERFORMANCE DB2SECURITYLABEL,
   POINTS      INTEGER,
   NAME        VARCHAR(50))
  SECURITY POLICY CONTRIBUTIONS
```

24. Create a table with protected columns.

```
CREATE TABLE TOASTMASTERS
  (PERFORMANCE CHAR(8),
   POINTS      INTEGER COLUMN SECURED WITH CLUBPOSITION,
   NAME        VARCHAR(50))
  SECURITY POLICY CONTRIBUTIONS
```

25. Create a table with protected rows and columns.

```
CREATE TABLE TOASTMASTERS
  (PERFORMANCE DB2SECURITYLABEL,
   POINTS      INTEGER COLUMN SECURED WITH CLUBPOSITION,
   NAME        VARCHAR(50))
  SECURITY POLICY CONTRIBUTIONS
```

26. Large objects for a partitioned table reside, by default, in the same table space as the data. This default behavior can be overridden by using the LONG IN clause to specify one or more table spaces for the large objects. Create a table named DOCUMENTS whose large object data is to be stored (in a round-robin fashion for each data partition) in table spaces TBSP1 and TBSP2.

```
CREATE TABLE DOCUMENTS
  (ID INTEGER,
   CONTENTS CLOB)
  LONG IN TBSP1, TBSP2
  PARTITION BY RANGE (ID)
    (STARTING 1 ENDING 1000
     EVERY 100)
```

Alternatively, use the long form of the syntax to explicitly identify a large table space for each data partition. In this example, the CLOB data for the first data partition is placed in LARGE_TBSP3, and the CLOB data for the remaining data partitions is spread across LARGE_TBSP1 and LARGE_TBSP2 in a round-robin fashion.

```
CREATE TABLE DOCUMENTS
  (ID INTEGER,
   CONTENTS CLOB)
```

```
     LONG IN LARGE_TBSP1, LARGE_TBSP2
     PARTITION BY RANGE (ID)
       (STARTING 1 ENDING 100
        IN TBSP1 LONG IN LARGE_TBSP3,
        STARTING 101 ENDING 1000
        EVERY 100)
```

27. Create a partitioned table named ACCESSNUMBERS having two data partitions. The row (10, NULL) is to be placed in the first partition, and the row (NULL, 100) is to be placed in the second (last) data partition.

```
   CREATE TABLE ACCESSNUMBERS
     (AREA INTEGER,
      EXCHANGE INTEGER)
     PARTITION BY RANGE (AREA NULLS LAST, EXCHANGE NULLS FIRST)
       (STARTING (1,1) ENDING (10,100),
        STARTING (11,1) ENDING (MAXVALUE,MAXVALUE))
```

Because null values in the second column are sorted first, the row (11, NULL) would sort below the low boundary of the last data partition (11, 1); attempting to insert this row returns an error. The row (12, NULL) would fall within the last data partition.

28. Create a table named RATIO having a single data partition and partitioning column PERCENT.

```
   CREATE TABLE RATIO
     (PERCENT INTEGER)
     PARTITION BY RANGE (PERCENT)
       (STARTING (MINVALUE) ENDING (MAXVALUE))
```

This table definition allows any integer value for column PERCENT to be inserted. The following definition for the RATIO table allows any integer value between 1 and 100 inclusive to be inserted into column PERCENT.

```
   CREATE TABLE RATIO
     (PERCENT INTEGER)
     PARTITION BY RANGE (PERCENT)
       (STARTING 0 EXCLUSIVE ENDING 100 INCLUSIVE)
```

29. Create a table named MYDOCS with two columns: one is an identifier, and the other stores XML documents.

```
   CREATE TABLE MYDOCS
     (ID INTEGER,
      DOC XML)
   IN HLTBSPACE
```

30. Create a table named NOTES with four columns, including one for storing XML-based notes.

```
   CREATE TABLE NOTES
     (ID          INTEGER,
      DESCRIPTION VARCHAR(255),
      CREATED     TIMESTAMP,
      NOTE        XML)
```

31. Create a table, EMP_INFO, that contains a phone number and address for each employee. Include a ROW CHANGE TIMESTAMP column in the table to track the modification of employee information.

```
CREATE TABLE EMP_INFO
     (EMPNO           CHAR(6) NOT NULL,
      EMP_INFOCHANGE  TIMESTAMP NOT NULL GENERATED ALWAYS
       FOR EACH ROW ON UPDATE
       AS ROW CHANGE TIMESTAMP,
      EMP_ADDRESS     VARCHAR(300),
      EMP_PHONENO     CHAR(4),
      PRIMARY KEY (EMPNO) )
```

32. Create a partitioned table named DOCUMENTS having two data partitions:

- The data object in the first partition resides in table space TBSP11. The partitioned index partition on the partition resides in table space TBSP21. The XML data object resides in table space TBSP31.

- The data object in the second partition resides in table space TBSP12. The partitioned index partition on the partition resides in table space TBSP22. The XML data object resides in table space TBSP32.

The table level INDEX IN clause has no impact on table space selection for partitioned indexes.

```
CREATE TABLE DOCUMENTS
(ID        INTEGER,
 CONTENTS  XML)  INDEX IN TBSPX
PARTITION BY (ID NULLS LAST)
(STARTING FROM 1 INCLUSIVE ENDING AT 100 INCLUSIVE
 IN TBSP11 INDEX IN TBSP21 LONG IN TBSP31,
 STARTING FROM 101 INCLUSIVE ENDING AT 200 INCLUSIVE
 IN TBSP21 INDEX IN TBSP22 LONG IN TBSP32)
```

33. Create a partitioned table named SALES having two data partitions:

- The data object in the first partition resides in table space TBSP11. The partitioned index partition on the partition resides in table space TBSP21.
- The data object in the second partition resides in table space TBSP12. The partitioned index object resides in table space TBSP22.

The table level INDEX IN clause has no impact on table space selection for partitioned indexes.

```
CREATE TABLE SALES
(SID     INTEGER,
 AMOUNT  INTEGER)  INDEX IN TBSPX
PARTITION BY RANGE (SID NULLS LAST)
(STARTING FROM 1 INCLUSIVE ENDING AT 100 INCLUSIVE
 IN TBSP11 INDEX IN TBSP21,
 STARTING FROM 101 INCLUSIVE ENDING AT 200 INCLUSIVE
 IN TBSP12 INDEX IN TBSP22)
```

34. Create a table named BOOKS with four columns, including one named DATE_ADDED, which inserts the current TIMESTAMP by default.

```
CREATE TABLE BOOKS
  (ISBN_NUM   INTEGER,
   TITLE      VARCHAR(255),
   AUTHOR     VARCHAR(255),
   DATE_ADDED TIMESTAMP WITH DEFAULT CURRENT TIMESTAMP)
```

35. Create a Unicode table called STUDENTS in a non-Unicode database. Assume that the database was created using code set 1252 and territory CA and the ALT_COLLATE database configuration parameter was updated to IDENTITY_16BIT.

```
CREATE TABLE STUDENTS (
            STUDENTID INT NOT NULL,
           FAMILY_NAME VARCHAR(36) NOT NULL,
           GIVEN_NAME VARCHAR(36) NOT NULL,
           PRIMARY KEY(STUDENTID))
      CCSID UNICODE
```

36. Create a table called TDEPT_TEMP, based on the TDEPT table that is created in Example 1.

```
CREATE TABLE TDEPT_TEMP LIKE TDEPT
```

The TDEPT_TEMP table will have the same definition as TDEPT except that the primary key will not be defined and a default table space will be implicitly chosen.

37. Create a column-organized user-maintained materialized query table on column-organized table CDE.TDEPT.

```
CREATE TABLE mqt_tdept AS
   (SELECT *
     FROM cde.tdept
     WHERE deptno BETWEEN 10 AND 20)
   DATA INITIALLY DEFERRED
   REFRESH DEFERRED
   MAINTAINED BY USER
   ORGANIZE BY COLUMN
```

38. Column security labels inherited by a materialized query table.

```
CREATE SECURITY LABEL COMPONENT level_array ARRAY ['A', 'B', 'C']

CREATE SECURITY POLICY P COMPONENTS level_array WITH DB2LBACRULES

CREATE SECURITY LABEL P.A COMPONENT level_array 'A'

CREATE SECURITY LABEL P.B COMPONENT level_array 'B'

CREATE SECURITY LABEL P.C COMPONENT level_array 'C'

CREATE TABLE t1 (c1 INT, c2 INT SECURED WITH B, c3 REAL SECURED WITH A)
   SECURITY POLICY P

CREATE TABLE t2 (c4 REAL, c5 INT SECURED WITH C, c6 DB2SECURITYLABEL)
   SECURITY POLICY P
```

Generate a materialized query table

```
CREATE TABLE m1 AS
   (SELECT  c1, c3, c5, c6 FROM  t1,t2 WHERE c2 !=100)
   DATA INITALLY DEFERRED REFRESH DEFERRED
```

The security label of t1.c2 is used to compute security labels of all columns of m1 because it appears in the predicates of the query. The label-based access control properties of the materialized query table m1 are:

- Security policy = P
- Security label of column m1.c1 = P.B
- Security label of column m1.c3 = P.A
- Security label of column m1.c5 = P.B
- Security label of column m1.c6 = P.B and it is also DB2SECURITYLABEL.

A staging table for a materialized query table is protected with label-based access control. Staging table st1 is defined as:

```
CREATE TABLE st1 FOR m1 PROPAGATE IMMEDIATE
```

The label-based access control properties of the staging table st1 are:

- Security policy = P
- Security label of column st1.c1 = P.B
- Security label of column st1.c3 = P.A
- Security label of column st1.c5 = P.B
- Security label of column st1.c6 = P.B and it is also DB2SECURITYLABEL.

39. The following example shows you how to create a shadow table called T1_SHADOW that is based on the row-organized table T1.

    a. Create the base table and define a primary key. The primary key on the base table must be included in the select list of the shadow table. The primary key on the shadow table is required to provide a one-to-one mapping for each row in the base table to the corresponding row in the shadow table. The primary key also facilitates maintenance of the shadow table.

    ```
    CREATE TABLE t1 (
      c1 INTEGER NOT NULL,
      c2 INTEGER
    ) ORGANIZE BY ROW;

    ALTER TABLE t1
      ADD CONSTRAINT t1_pk PRIMARY KEY(c1);
    ```

    b. Create the shadow table:

    ```
    CREATE TABLE t1_shadow AS
        (SELECT c1, c2 FROM t1)
    ```

```
  DATA INITIALLY DEFERRED
  REFRESH DEFERRED
  MAINTAINED BY REPLICATION
  ORGANIZE BY COLUMN;

SET INTEGRITY FOR t1_shadow ALL IMMEDIATE UNCHECKED;

ALTER TABLE t1_shadow
  ADD CONSTRAINT t1_shadow_pk PRIMARY KEY (c1);
```

40. Create a table that is named STRING_UNITS, which demonstrates each possible string unit specification.

```
CREATE TABLE string_units
   (c1 VARCHAR(10),
    c2 VARCHAR(10 OCTETS),
    c3 VARCHAR(10 CODEUNITS32),
    c4 VARGRAPHIC(10),
    c5 VARGRAPHIC(10 CODEUNITS16),
    c6 VARGRAPHIC(10 CODEUNITS32))
```

    The columns have the following string units:

    - c1 = OCTETS, if the environment string units is SYSTEM; CODEUNITS32 if the environment string units is CODEUNITS32
    - c2 = OCTETS
    - c3 = CODEUNITS32
    - c4 = CODEUNITS16, if the environment string units is SYSTEM; CODEUNITS32 if the environment string units is CODEUNITS32
    - c5 = CODEUNITS16
    - c6 = CODEUNITS32

    Environment string units can be set with the NLS_STRING_UNITS session level global variable. If the NLS_STRING_UNITS session level global variable is not set or is null, the environment string units are determined by the value of the **string_units** database configuration parameter.

41. Create a random distribution table using the random by unique method. The distribution keys are automatically set to both keys of the index: **ID** and **NAME**.

```
CREATE TABLE RAND_BY_UNIQUE (ID BIGINT NOT NULL,
                             NAME CHAR(25) NOT NULL,
                             DESCRIPTION VARCHAR(1000),
                             PRIMARY KEY(ID, NAME)) DISTRIBUTE BY RANDOM
```

42. Create a random distribution table using the random by generation method. The distribution key is set to an internal column **RANDOM_DISTRIBUTION_KEY**, which is hidden from SQL unless explicitly specified.

```
CREATE TABLE RAND_BY_GENERATION (C1 BIGINT) DISTRIBUTE BY RANDOM
```

CREATE TABLESPACE

The CREATE TABLESPACE statement defines a new table space within the database, assigns containers to the table space, and records the table space definition and attributes in the catalog.

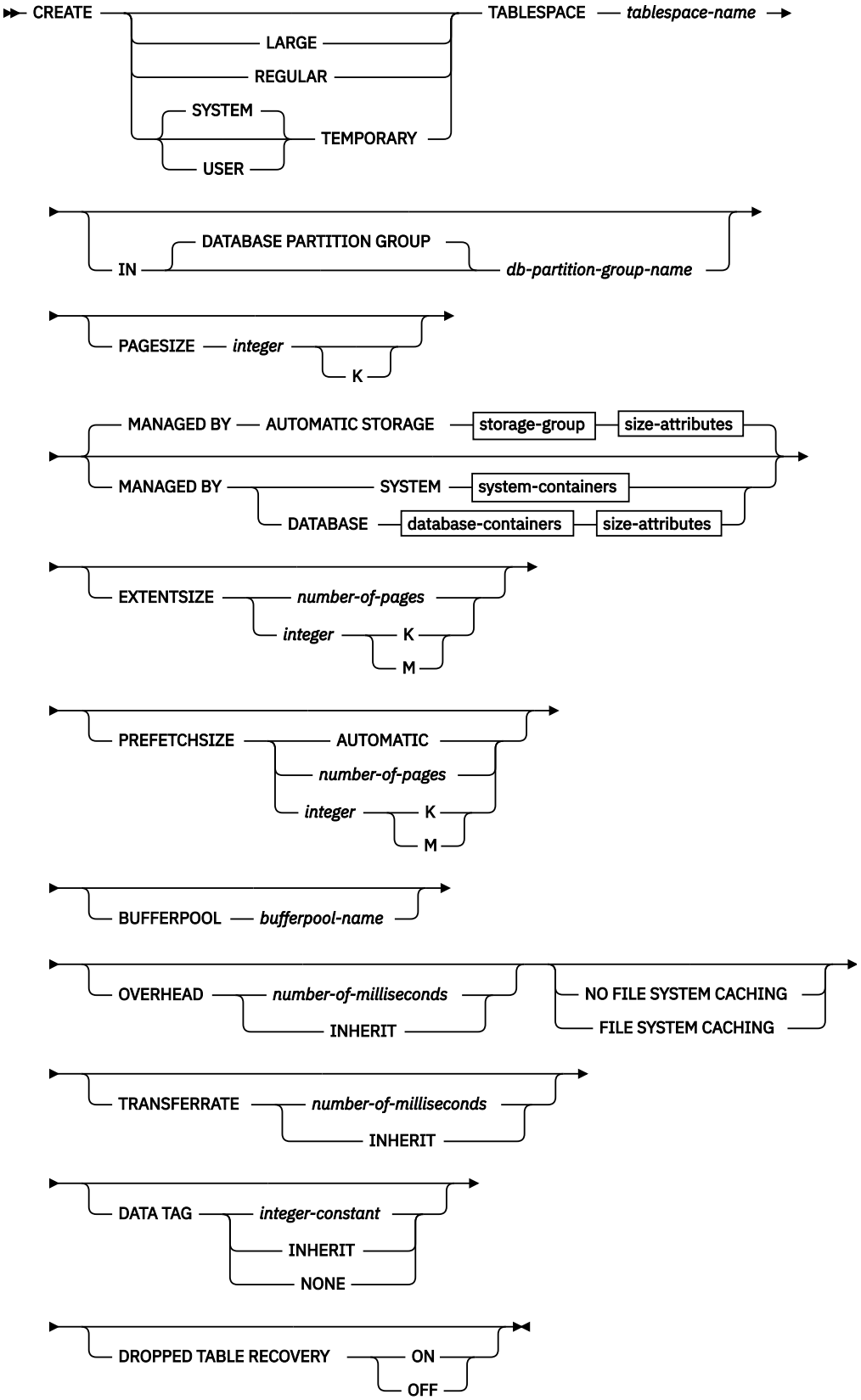
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

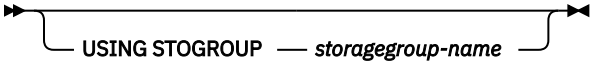
Authorization

The privileges that are held by the authorization ID of the statement must include SYSCTRL or SYSADM authority.

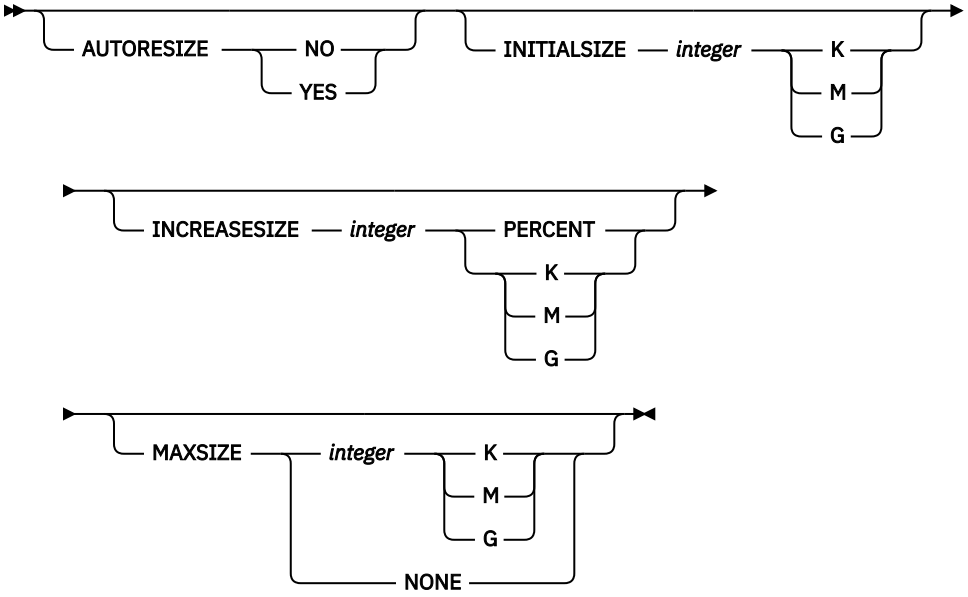
Syntax



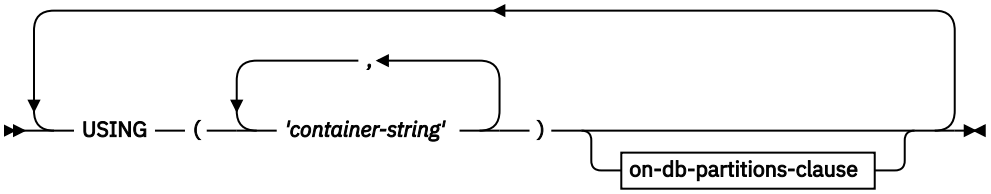
Storage-group



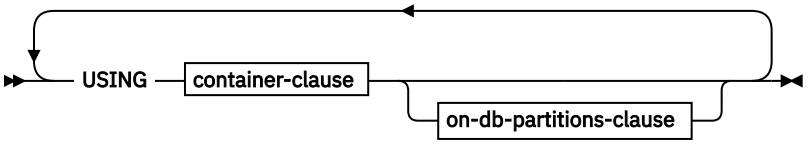
Size-attributes



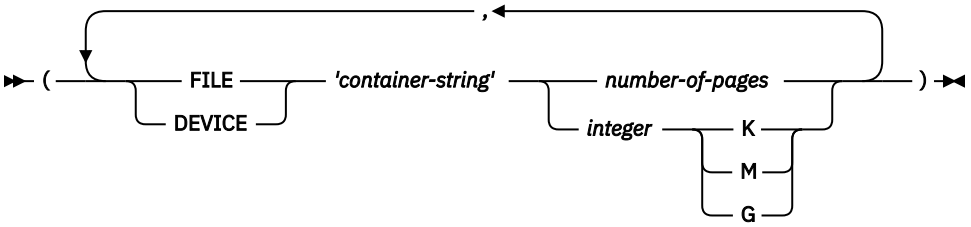
System-containers



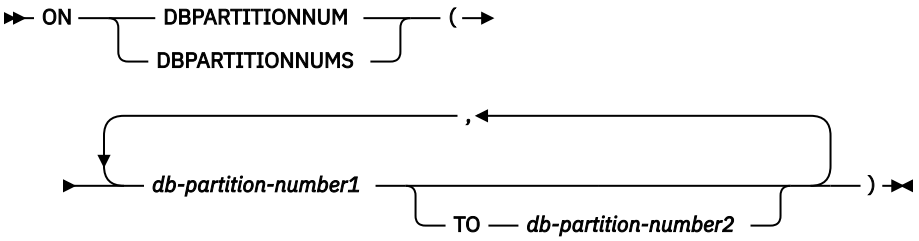
Database-containers



Container-clause



On-db-partitions-clause



Description

LARGE, REGULAR, SYSTEM TEMPORARY, or USER TEMPORARY
Specifies the type of table space that is to be created. If no type is specified, the default is determined by the MANAGED BY clause.

LARGE
Stores all permanent data. This type is only allowed on database-managed space (DMS) table spaces. It is also the default type for DMS table spaces when no type is specified. When a table is placed in a large table space:

- The table can be larger than a table in a regular table space. For more information on table and table space limits, see SQL and XML limits.
- The table can support more than 255 rows per data page, which can improve space utilization on data pages.
- Indexes that are defined on the table will require an extra 2 bytes per row entry, compared to indexes defined on a table that resides in a regular table space.

REGULAR
Stores all permanent data. This type applies to both DMS and SMS table spaces. This is the only type that is allowed for SMS table spaces, and it is also the default type for SMS table spaces when no type is specified.

SYSTEM TEMPORARY
Stores temporary tables, work areas that are used by the database manager to perform operations such as sorts or joins. A database must always have at least one SYSTEM TEMPORARY table space, because temporary tables can only be stored in such a table space. A temporary table space is created automatically when a database is created.

USER TEMPORARY
Stores created temporary tables and declared temporary tables. No user temporary table spaces exist when a database is created. To allow the definition of created temporary tables or declared temporary tables, at least one user temporary table space should be created with appropriate USE privileges.

tablespace-name
Names the table space. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *tablespace-name* must not identify a table space that already exists in the catalog (SQLSTATE 42710). The *tablespace-name* must not begin with the characters 'SYS' (SQLSTATE 42939).

IN DATABASE PARTITION GROUP *db-partition-group-name*
Specifies the database partition group for the table space. The database partition group must exist. The only database partition group that can be specified when creating a SYSTEM TEMPORARY table space is IBMTEMPGROUP. The DATABASE PARTITION GROUP keywords are optional.

If the database partition group is not specified, the default database partition group (IBMDEFAULTGROUP) is used for REGULAR, LARGE, and USER TEMPORARY table spaces. For SYSTEM TEMPORARY table spaces, the default database partition group IBMTEMPGROUP is used.

PAGESIZE *integer* [K]
Defines the size of pages that are used for the table space. The valid values for *integer* without the suffix K are 4096, 8192, 16384, or 32768. The valid values for *integer* with the suffix K are 4, 8, 16, or 32. Any number of spaces is allowed between *integer* and K, including no space. An error occurs if the page size is not one of these values (SQLSTATE 428DE), or if the page size is not the same as the page size of the buffer pool that is associated with the table space (SQLSTATE 428CB).

The default value is provided by the **pagesize** database configuration parameter, which is set when the database is created.

MANAGED BY AUTOMATIC STORAGE
Specifies that the table space is to be an automatic storage table space. If no storage groups are defined, an error is returned (SQLSTATE 55060).

The database manager automatically decides how the automatic storage table space is initially created. Temporary table spaces are initialized as system-managed space (SMS) table space and permanent table spaces are initialized as database-managed space (DMS) table space. When creating a permanent table space and the type of table space is not specified, the default behavior is to create a large table space. With an automatic storage table space, the database manager determines which containers are to be assigned to the table space, based on the storage paths that are associated with the storage group the table space uses.

storage-group
Specify the storage group for an automatic storage table space.

USING STOGROUP
For an automatic storage table space, identifies the storage group for the table space in which the table space data will be stored. If a *storagegroup-name* is not specified, then the currently designated default storage group is used. This clause only applies to automatic storage table spaces (SQLSTATE 42613).

storagegroup-name
Identifies the storage group in which table space data will be stored. *storagegroup-name* must identify a storage group that exists at the current server (SQLSTATE 42704). This is a one-part name.

size-attributes
Specify the size attributes for an automatic storage table space or a DMS table space that is not an automatic storage table space. SMS table spaces are not auto-resizable.

AUTORESIZE
Specifies whether the auto-resize capability of a DMS table space or an automatic storage table space is to be enabled. Auto-resizable table spaces automatically increase in size when they become full. The default is NO for DMS table spaces and YES for automatic storage table spaces.

NO
Specifies that the auto-resize capability of a DMS table space or an automatic storage table space is to be disabled.

YES
Specifies that the auto-resize capability of a DMS table space or an automatic storage table space is to be enabled.

INITIALSIZE *integer* K | M | G
Specifies the initial size, per database partition, of an automatic storage table space. This option is only valid for automatic storage table spaces. The integer value must be followed by K (for kilobytes), M (for megabytes), or G (for gigabytes). The actual value that is used might be slightly smaller than what was specified, because the database manager strives to maintain a consistent size across containers in the table space. Moreover, if the table space is auto-resizable and the initial size is not large enough to contain metadata that must be added to the new table space, the database manager will continue to extend the table space by the value of INCREASESIZE until enough space exists. If the INITIALSIZE clause is not specified, the database manager determines an appropriate value. The value for *integer* must be at least 48 K.

INCREASESIZE *integer* PERCENT or INCREASESIZE *integer* K | M | G
Specifies the amount, per database partition, by which a table space that is enabled for auto-resize will automatically be increased when the table space is full, and a request for space has been made. The integer value must be followed by:

- PERCENT to specify the amount as a percentage of the table space size at the time that a request for space is made. When PERCENT is specified, the integer value must be between `0` and `100` (SQLSTATE 42615).
- K (for kilobytes), M (for megabytes), or G (for gigabytes) to specify the amount in bytes.

The actual value that is used might be slightly smaller or larger than what was specified, because the database manager strives to maintain consistent growth across containers in the table space. If the table space is auto-resizable, but the INCREASESIZE clause is not specified, the database manager determines an appropriate value.

MAXSIZE *integer* K | M | G or MAXSIZE NONE
Specifies the maximum size to which a table space that is enabled for auto-resize can automatically be increased. If the table space is auto-resizable, but the MAXSIZE clause is not specified, the default is NONE.

integer
Specifies a hard limit on the size, per database partition, to which a DMS table space or an automatic storage table space can automatically be increased. The integer value must be followed by K (for kilobytes), M (for megabytes), or G (for gigabytes). The actual value that is used might be slightly smaller than what was specified, because the database manager strives to maintain consistent growth across containers in the table space.

NONE
Specifies that the table space is to be allowed to grow to file system capacity, or to the maximum table space size (described in "SQL and XML limits").

MANAGED BY SYSTEM
Specifies that the table space is to be an SMS table space.

MANAGED BY SYSTEM cannot be specified in a Db2 pureScale environment (SQLSTATE 42997).

Important: The SMS table space type is deprecated for user-defined permanent table spaces and might be removed in a future release. The SMS table space type is not deprecated for catalog and temporary table spaces. For more information, see "SMS permanent table spaces have been deprecated" at http://www.ibm.com/support/knowledgecenter/SSEPGG_10.1.0/com.ibm.db2.luw.wn.doc/doc/i0058748.html.

system-containers
Specify the containers for an SMS table space.

USING *('container-string', ...)*
For an SMS table space, identifies one or more containers that will belong to the table space and in which the table space data will be stored. The *container-string* cannot exceed 240 bytes in length.

Each *container-string* can be an absolute or relative directory name.

The directory name, if not absolute, is relative to the database directory, and can be a path name alias (or symbolic link) to storage that is not physically associated with the database directory. For example, *dbdir*`/work/c1` might be a symbolic link to a separate file system.

If any component of the directory name does not exist, it is created by the database manager. When a table space is dropped, all components created by the database manager are deleted. If the directory identified by *container-string* exists, it must not contain any files or subdirectories (SQLSTATE 428B2).

The format of *container-string* depends on the operating system.

Operating system	Format of absolute path name
Linux AIX	An absolute path name begins with a forward slash (/)
Windows	An absolute directory path name begins with a drive letter and a colon (:)

A relative path name on any platform does not begin with an operating system-dependent character.

For file-level protocols, such as NAS and CIFS, remote resources (such as LAN-redirected drives or NFS-mounted file systems) are currently supported only when the following technologies are used:

- Network Appliance Filers
- IBM Network Attached Storage

- NEC iStorage S2100, S2200, or S4100
- NEC Storage NS Series with a database server on Windows

  Note: NEC Storage NS Series is supported only with the use of an uninterrupted power supply (UPS); continuous UPS (rather than standby) is recommended.

An NFS-mounted file system on AIX must be mounted in uninterruptible mode using the **-o nointr** option.

Block-level protocols, such as iSCSI and FCP, are supported by any backend storage that has non-volatile RAM or battery backup. The storage technology must guarantee that successful writes are not lost in the event of failure, such as a power outage.

on-db-partitions-clause
Specifies the database partition or partitions on which the containers are created in a partitioned database. If this clause is not specified, then the containers are created on the database partitions in the database partition group that are not explicitly specified in any other *on-db-partitions-clauses*. For a SYSTEM TEMPORARY table space defined on database partition group IBMTEMPGROUP, when the *on-db-partitions-clause* is not specified, the containers will also be created on all new database partitions added to the database.

MANAGED BY DATABASE
Specifies that the table space is to be a DMS table space. When the type of table space is not specified, the default behavior is to create a large table space.

MANAGED BY DATABASE cannot be specified in a Db2 pureScale environment (SQLSTATE 42997).

Important: The DMS table space type is deprecated for user-defined permanent table spaces and might be removed in a future release. The DMS table space type is not deprecated for catalog and temporary table spaces. For more information, see "DMS permanent table spaces have been deprecated" at http://www.ibm.com/support/knowledgecenter/SSEPGG_10.1.0/com.ibm.db2.luw.wn.doc/doc/i0060577.html.

database-containers
Specify the containers for a DMS table space.

USING
Introduces a container-clause.

container-clause
Specifies the containers for a DMS table space.

(FILE|DEVICE *'container-string' number-of-pages, ...*)
For a DMS table space, identifies one or more containers that will belong to the table space and in which the table space data will be stored. The type of the container (either FILE or DEVICE) and its size (in PAGESIZE pages) are specified. The size can also be specified as an integer value followed by K (for kilobytes), M (for megabytes), or G (for gigabytes). If specified in this way, the floor of the number of bytes divided by the pagesize is used to determine the number of pages for the container. A mixture of FILE and DEVICE containers can be specified. The *container-string* cannot exceed 254 bytes in length.

For a FILE container, *container-string* must be an absolute or relative file name. The file name, if not absolute, is relative to the database directory. If any component of the directory name does not exist, it is created by the database manager. If the file does not exist, it will be created and initialized to the specified size by the database manager. When a table space is dropped, all components that are created by the database manager are deleted.

Note: If the file exists, it is overwritten, and if it is smaller than specified, it is extended. The file will not be truncated if it is larger than specified.

For a DEVICE container, *container-string* must be a device name. The device must already exist.

All containers must be unique across all databases. A container can belong to only one table space. The size of the containers can differ; however, optimal performance is achieved when

all containers are the same size. The exact format of *container-string* depends on the operating system.

For file-level protocols, such as NAS and CIFS, remote resources (such as LAN-redirected drives or NFS-mounted file systems) are currently supported only when the following technologies are used:

- Network Appliance Filers
- IBM Network Attached Storage
- NEC iStorage S2100, S2200, or S4100
- NEC Storage NS Series with a database server on Windows

Note: NEC Storage NS Series is supported only with the use of an uninterrupted power supply (UPS); continuous UPS (rather than standby) is recommended.

Block-level protocols, such as iSCSI and FCP, are supported by any backend storage that has non-volatile RAM or battery backup. The storage technology must guarantee that successful writes are not lost in the event of failure, such as a power outage.

on-db-partitions-clause
Specifies the database partition or partitions on which the containers are created in a partitioned database. If this clause is not specified, then the containers are created on the database partitions in the database partition group that are not explicitly specified in any other *on-db-partitions-clause*. For a SYSTEM TEMPORARY table space defined on database partition group IBMTEMPGROUP, when the *on-db-partitions-clause* is not specified, the containers will also be created on all new database partitions added to the database.

on-db-partitions-clause
Specifies the database partitions on which containers are created in a partitioned database.

ON DBPARTITIONNUMS
Keywords indicating that individual database partitions are specified. DBPARTITIONNUM is a synonym for DBPARTITIONNUMS.

db-partition-number1
Specify a database partition number.

TO *db-partition-number2*
Specify a range of database partition numbers. The value of *db-partition-number2* must be greater than or equal to the value of *db-partition-number1* (SQLSTATE 428A9). Containers are to be created on each database partition between and including the specified values. A specified database partition must be in the database partition group for the table space.

The database partition specified by number, and every database partition within the specified range of database partitions must exist in the database partition group for the table space (SQLSTATE 42729). A database partition number can only appear explicitly or within a range in exactly one *on-db-partitions-clause* for the statement (SQLSTATE 42613).

EXTENTSIZE *number-of-pages*
Specifies the number of PAGESIZE pages that will be written to a container before skipping to the next container. The extent size value can also be specified as an integer value followed by K (for kilobytes) or M (for megabytes). If specified in this way, the floor of the number of bytes divided by the page size is used to determine the value for the extent size. The database manager cycles repeatedly through the containers as data is stored.

In a Db2 pureScale environment, you should use an extent size of at least 32 pages. This minimum extent size reduces the amount of internal message traffic within the Db2 pureScale environment when extents are added for a table or index.

The default value is provided by the **dft_extent_sz** database configuration parameter, which has a valid range of 2-256 pages.

PREFETCHSIZE
Specifies to read in data that is needed by a query before it is referenced by the query, so that the query does not need to wait for I/O to be performed.

The default value is provided by the **dft_prefetch_sz** database configuration parameter.

AUTOMATIC
Specifies that the prefetch size of a table space is to be updated automatically; that is, the prefetch size will be managed by the database manager.

The prefetch size will be updated automatically whenever the number of containers in a table space changes (following successful execution of an ALTER TABLESPACE statement that adds or drops one or more containers). The prefetch size is also automatically updated at database startup.

number-of-pages
Specifies the number of PAGESIZE pages that will be read from the table space when data prefetching is being performed. The maximum value is 32767.

***integer* K | M**
Specifies the prefetch size value as an integer value followed by K (for kilobytes) or M (for megabytes). If specified in this way, the floor of the number of bytes divided by the page size is used to determine the number of pages value for prefetch size.

BUFFERPOOL *bufferpool-name*
The name of the buffer pool that is used for tables in this table space. The buffer pool must exist (SQLSTATE 42704). Furthermore, it must exist prior to the start of the transaction containing the CREATE TABLESPACE statement. If the buffer pool is created in the same UOW as the table space, it is not available to use. Instead, the table space will use the small system buffer pool with the matching page size. If not specified, the default buffer pool (IBMDEFAULTBP) is used. The page size of the buffer pool must match the page size specified (or defaulted) for the table space (SQLSTATE 428CB). The database partition group of the table space must be defined for the buffer pool (SQLSTATE 42735).

OVERHEAD *number-of-milliseconds* or OVERHEAD INHERIT
Specifies the I/O controller overhead and disk seek and latency time. This value is used to determine the cost of I/O during query optimization. If OVERHEAD is not specified for a non-automatic storage table space, the value defaults to the database creation default described later in the description for this keyword. If OVERHEAD is not specified for an automatic storage table space, the default is to INHERIT the value from the storage group it is using. If the OVERHEAD value at the storage group is undefined, the OVERHEAD defaults to the database creation default. For more information on tuning, refer to Table space impact on query optimization.

number-of-milliseconds
The value of *number-of-milliseconds* is any numeric literal (integer, decimal, or floating point). If this value is not the same for all containers, the number should be the average for all containers that belong to the table space.

INHERIT
If INHERIT is specified, the table space must be defined by using automatic storage and the OVERHEAD is dynamically inherited from the storage group. INHERIT cannot be specified if the table space is not defined by using automatic storage (SQLSTATE 42613).

For a database that was created in Db2 version 10.1 or later, the default I/O controller overhead and disk seek and latency time for 4 KB PAGESIZE table space is 6.725 milliseconds.

For a database that was upgraded from a previous version of Db2 to Db2 version 10.1 or later, the default I/O controller overhead and disk seek and latency time is as follows:

- 7.5 milliseconds for a database that is created in Db2 version 9.7 or higher

FILE SYSTEM CACHING or NO FILE SYSTEM CACHING
Specifies whether I/O operations are to be cached at the file system level or non-cached by using direct I/O. If neither option is specified, the I/O mode is determined based on operating system, file system, and in the case of SMS table spaces, data object type. For more information, see File

system caching configurations. Note that once a non-default file system caching option is chosen, it is not possible to return to the default (unspecified) behavior. Instead, the file system caching mode must be selected explicitly.

FILE SYSTEM CACHING
Specifies that all I/O operations in the target table space are to be cached at the file system level.

NO FILE SYSTEM CACHING
Specifies that all I/O operations are to bypass the file system-level cache. LOB and Long field data in SMS table spaces are excepted.

Note:

Db2 supports disk devices with physical sector sizes of 512 bytes or 4096 bytes.

Support for 4096 byte sector sizes is not enabled by default, and can be enabled using the **DB2_4K_DEVICE_SUPPORT** registry variable.

TRANSFERRATE *number-of-milliseconds* or TRANSFERRATE INHERIT
Specifies the time to read one page into memory. If TRANSFERRATE is not specified for a non-automatic storage table space, the value defaults to the database creation default described later in the description for this keyword. If TRANSFERRATE is not specified for an automatic storage table space, the default is to INHERIT the value from the storage group it is using. If the DEVICE READ RATE value at the storage group is undefined, the TRANSFERRATE defaults to the database creation default. For more information on tuning, refer to Table space impact on query optimization.

number-of-milliseconds
This value is used to determine the cost of I/O during query optimization. The value of *number-of-milliseconds* is any numeric literal (integer, decimal, or floating point). If this value is not the same for all containers, the number should be the average for all containers that belong to the table space.

INHERIT
If INHERIT is specified, the table space must be defined by using automatic storage and the TRANSFERRATE is dynamically inherited from the DEVICE READ RATE of the storage group. INHERIT cannot be specified if the table space is not defined by using automatic storage (SQLSTATE 42613).

When an automatic storage table space inherits the TRANSFERRATE setting from the storage group it is using, the DEVICE READ RATE of the storage group, which is in megabytes per second, is converted into milliseconds per page read accounting for the PAGESIZE setting of the table space. The conversion formula follows:

TRANSFERRATE = (1 / DEVICE READ RATE) * 1000 / 1024000 * PAGESIZE

For a database that was created in Db2 version 10.1 or later, the default time to read one page into memory for 4 KB PAGESIZE table space is 0.04 milliseconds.

For a database that was upgraded from a previous version of Db2 to Db2version 10.1 or later, the default time to read one page into memory is as follows:

- 0.06 milliseconds for a database that is created in Db2 version 9.7 or higher

DATA TAG *integer-constant*, DATA TAG INHERIT or DATA TAG NONE
Specifies a tag for the data in the table space. If the DATA TAG is not specified, the default for automatic storage table spaces is to INHERIT from the storage group it is using and for non-automatic table spaces it will be set to NONE. This value can be used as part of a WLM configuration in a work class definition (see “CREATE WORK CLASS SET ” on page 1560) or referenced within a threshold definition (see “CREATE THRESHOLD ” on page 1443). This clause cannot be specified if TEMPORARY is also specified (SQLSTATE 42613).

integer-constant
Valid values for *integer-constant* are integers in the range 1 - 9. If an *integer-constant* is specified and an associated storage group exists, the data tag that is specified for the table space will override any data tag value that is specified for the associated storage group.

INHERIT
If INHERIT is specified, the table space must be defined by using automatic storage and the data tag is dynamically inherited from the storage group. INHERIT cannot be specified if the table space is not defined by using automatic storage (SQLSTATE 42613).

NONE
If NONE is specified, there is no data tag.

DROPPED TABLE RECOVERY
Indicates whether dropped tables in the specified table space can be recovered by using the **RECOVER DROPPED TABLE** option of the **ROLLFORWARD DATABASE** command. This clause can only be specified for a regular or large table space (SQLSTATE 42613).

ON
Specifies that dropped tables can be recovered. This is the default.

OFF
Specifies that dropped tables cannot be recovered.

Rules

- If automatic storage is not defined for the database, an error is returned (SQLSTATE 55060).
- The INITIALSIZE clause cannot be specified with the MANAGED BY SYSTEM or MANAGED BY DATABASE clause (SQLSTATE 42601).
- The AUTORESIZE, INCREASESIZE, or MAXSIZE clause cannot be specified with the MANAGED BY SYSTEM clause (SQLSTATE 42601).
- The AUTORESIZE, INITIALSIZE, INCREASESIZE, or MAXSIZE clause cannot be specified for the creation of a temporary automatic storage table space (SQLSTATE 42601).
- The INCREASESIZE or MAXSIZE clause cannot be specified if the tables space is not auto-resizable (SQLSTATE 42601).
- AUTORESIZE cannot be enabled for DMS table spaces that are defined to use raw device containers (SQLSTATE 42601).
- A table space must initially be large enough to hold five extents (SQLSTATE 57011).
- The maximum size of a table space must be larger than its initial size (SQLSTATE 560B0).
- Container operations (ADD, EXTEND, RESIZE, DROP, or BEGIN NEW STRIPE SET) cannot be performed on automatic storage table spaces because the database manager is controlling the space management of such table spaces (SQLSTATE 42858).
- Each container definition requires 53 bytes plus the number of bytes necessary to store the container name. The combined length of all container definitions for the table space cannot exceed 208 kilobytes (SQLSTATE 54034).
- For a partitioned database, if more than one database partition resides on the same physical node, the same device or path cannot be specified for more than one database partition (SQLSTATE 42730). In this environment, either specify a unique *container-string* for each database partition, or use a relative path name.
- Only automatic storage table spaces can be created in a Db2 pureScale environment(SQLSTATE 42997).
- ***Container size limits:*** In DMS table spaces, a container must be at least two times the extent size pages in length (SQLSTATE 54039). The maximum size of a container is operating system dependent.

Notes

- Choosing between a database-managed space or a system-managed space for a table space is a fundamental choice involving tradeoffs.

- When more than one TEMPORARY table space exists in the database, they are used in round-robin fashion to balance their usage.
- The owner of the table space is granted USE privilege with the WITH GRANT OPTION on the table space when it is created.
- An automatic storage table space is created as either an SMS table space or a DMS table space. DMS is chosen for large and regular table spaces, and SMS is chosen for temporary table spaces. This behavior cannot be depended upon because it might change in a future release. When DMS is chosen and the type of table space is not specified, the default behavior is to create a large table space.
- The creation of an automatic storage table space does not include container definitions. The database manager automatically determines the location and size, if applicable, of the containers based on the storage paths that are associated with the specified storage group or the default storage group. The database manager will attempt to grow large and regular table spaces, as necessary, if the maximum size has not been reached. This might involve extending existing containers or adding containers to a new stripe set. Every time that the database is activated, the database manager automatically reconfigures the number and location of the containers for temporary table spaces that are not in an abnormal state.
- A large or regular automatic storage table space will not use new storage paths (see the description of the ALTER STOGROUP statement) until there is no more space in one of the existing storage paths that the table space is using. Temporary automatic storage table spaces can only use the new storage paths once the database has been deactivated and then reactivated.
- ***Media attributes*:** The following table shows how the media attributes of newly created table spaces are treated in upgraded and newly created Db2 version 10.1 databases.

Table 143. Media attributes across different versions of Db2

Media attributes	Upgraded Database	Newly Created Database
New automatic storage table spaces / storage group DEVICE READ RATE set to *undefined*	Defaults based on version database was created (no change)	Not applicable
New automatic storage table spaces / storage group OVERHEAD set to *undefined*	Defaults based on version database was created (no change)	Not applicable
New automatic storage table spaces / storage group DEVICE READ RATE is set	Inherit from storage group factoring in PAGESIZE	Inherit from storage group factoring in PAGESIZE
New automatic storage table spaces / storage group OVERHEAD is set	Inherit from storage group	Inherit from storage group
New non-automatic storage table spaces	Defaults based on version database was created (no change)	Db2 version 10.1 media defaults taking PAGESIZE into account

- ***Default TRANSFERRATE***: The following table shows how the default TRANSFERRATE value differs for newly created table spaces.

Table 144. Default TRANSFERRATE

PAGESIZE	TRANSFERRATE
4 KB	0.04 ms per page read
8 KB	0.08 ms per page read
16 KB	0.16 ms per page read
32 KB	0.32 ms per page read

- ***Syntax alternatives***: The following are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - NODE can be specified in place of DBPARTITIONNUM.
  - NODES can be specified in place of DBPARTITIONNUMS.
  - NODEGROUP can be specified in place of DATABASE PARTITION GROUP.
  - LONG can be specified in place of LARGE.
- For the Db2 Developer-C Edition:
  - Altering an auto-resize table space without specifying **MAXSIZE** will implicitly set **MAXSIZE** to the remaining capacity, up to the defined storage size.
  - An attempt to resize, add, or extend the container size of all table spaces larger than the defined storage size results in a fail.
  - Altering a table space fails if there exists a subsequent CREATE TABLESPACE that hasn't been committed.

Examples

1. Create a large DMS table space on a Linux system using three devices of 10 000 4K pages each. Specify their I/O characteristics.

```
CREATE TABLESPACE PAYROLL
  MANAGED BY DATABASE
  USING (DEVICE'/dev/rhdisk6' 10000,
    DEVICE '/dev/rhdisk7' 10000,
    DEVICE '/dev/rhdisk8' 10000)
  OVERHEAD 12.67
  TRANSFERRATE 0.18
```

2. Create a regular SMS table space on Windows using three directories on three separate drives, with a 64-page extent size, and a 32-page prefetch size.

```
CREATE TABLESPACE ACCOUNTING
  MANAGED BY SYSTEM
  USING ('/tbsp/acc1', '/tbsp/acc2', '/tbsp/acc3')
  EXTENTSIZE 64
  PREFETCHSIZE 32
```

3. Create a system temporary DMS table space on a Linux system by using two files of 50 000 pages each, and a 256-page extent size.

```
CREATE TEMPORARY TABLESPACE TEMPSPACE2
  MANAGED BY DATABASE
  USING (FILE 'dbtmp/tempspace2.f1' 50000,
    FILE 'dbtmp/tempspace2.f2' 50000)
  EXTENTSIZE 256
```

4. Create a large DMS table space in database partition group ODDNODEGROUP (database partitions 1, 3, and 5) on a Linux system. Use the device `/dev/rhdisk0` for 10 000 4K pages on each database partition. Specify a database partition-specific device with 40 000 4K pages for each database partition.

```
CREATE TABLESPACE PLANS
  MANAGED BY DATABASE
  USING (DEVICE '/dev/rhdisk0' 10000, DEVICE '/dev/rn1hd01' 40000)
  ON DBPARTITIONNUM (1)
  USING (DEVICE '/dev/rhdisk0' 10000, DEVICE '/dev/rn3hd03' 40000)
  ON DBPARTITIONNUM (3)
  USING (DEVICE '/dev/rhdisk0' 10000, DEVICE '/dev/rn5hd05' 40000)
  ON DBPARTITIONNUM (5)
```

5. Create a large automatic storage table space that is named DATATS, allowing the system to make all decisions concerning table space size and growth.

```
CREATE TABLESPACE DATATS
```

or

```
CREATE TABLESPACE DATATS
  MANAGED BY AUTOMATIC STORAGE
```

6. Create a system temporary automatic storage table space named TEMPDATA.

```
CREATE TEMPORARY TABLESPACE TEMPDATA
```

or

```
CREATE TEMPORARY TABLESPACE TEMPDATA
  MANAGED BY AUTOMATIC STORAGE
```

7. Create a large automatic storage table space that is named USERSPACE3 with an initial size of 100 megabytes and a maximum size of 1 gigabyte.

```
CREATE TABLESPACE USERSPACE3
  INITIALSIZE 100 M
  MAXSIZE 1 G
```

8. Create a large automatic storage table space that is named LARGEDATA with a growth rate of 10 percent (that is, its total size increases by 10 percent each time that it is automatically resized) and a maximum size of 512 megabytes. Instead of specifying the INITIALSIZE clause, let the database manager determine an appropriate initial size for the table space.

```
CREATE LARGE TABLESPACE LARGEDATA
  INCREASESIZE 10 PERCENT
  MAXSIZE 512 M
```

9. Create a large DMS table space that is named USERSPACE4 with two file containers (each container being 1 megabyte in size), a growth rate of 2 megabytes, and a maximum size of 100 megabytes.

```
CREATE TABLESPACE USERSPACE4
  MANAGED BY DATABASE USING (FILE '/db/file1' 1 M, FILE '/db/file2' 1 M)
  AUTORESIZE YES
  INCREASESIZE 2 M
  MAXSIZE 100 M
```

10. Create large DMS table spaces, using RAW devices on a Windows operating system.
    - To specify entire physical drives, use the \\.*physical-drive* format:

      ```
      CREATE TABLESPACE TS1
        MANAGED BY DATABASE USING (DEVICE '\\.\PhysicalDrive5' 10000,
          DEVICE '\\.\PhysicalDrive6' 10000)
      ```

    - To specify logical partitions by using drive letters:

      ```
      CREATE TABLESPACE TS2
        MANAGED BY DATABASE USING (DEVICE '\\.\G:' 10000,
          DEVICE '\\.\H:' 10000)
      ```

    - To specify logical partitions by using volume global unique identifiers (GUIDs), use the **db2listvolumes** utility to retrieve the volume GUID for each local partition, then copy the GUID for the logical partition that you want into the table space container clause:

      ```
      CREATE TABLESPACE TS3
        MANAGED BY DATABASE USING (
          DEVICE '\\?\Volume{2ca6a0c1-8542-11d8-9734-00096b5322d2}\' 20000M)
      ```

      You might prefer to use volume GUIDs over the drive letter format if you have more partitions than available drive letters on the machine.
    - To specify logical partitions by using junction points (or volume mount points), mount the RAW partition to another NTFS-formatted volume as a junction point, then specify the path to the junction point on the NTFS volume as the container path. For example:

```
CREATE TABLESPACE TS4
  MANAGED BY DATABASE USING (DEVICE 'C:\JUNCTION\DISK_1' 10000,
    DEVICE 'C:\JUNCTION\DISK_2' 10000)
```

The partition is queried first to see whether there is a file system on it; if yes, the partition is not treated as a RAW device, and normal file system I/O operations are performed on the partition.

Related information

Best practices: Database storage

CREATE THRESHOLD

The CREATE THRESHOLD statement defines a threshold.

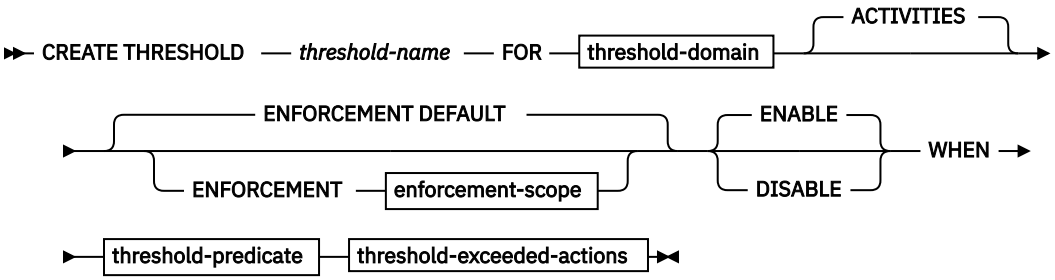
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

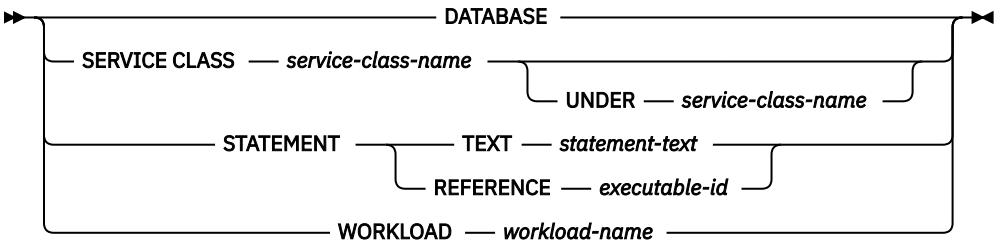
Authorization

The privileges held by the authorization ID of the statement must include WLMADM or DBADM authority.

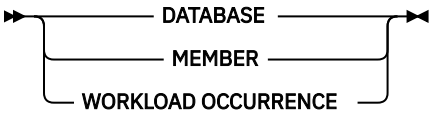
Syntax



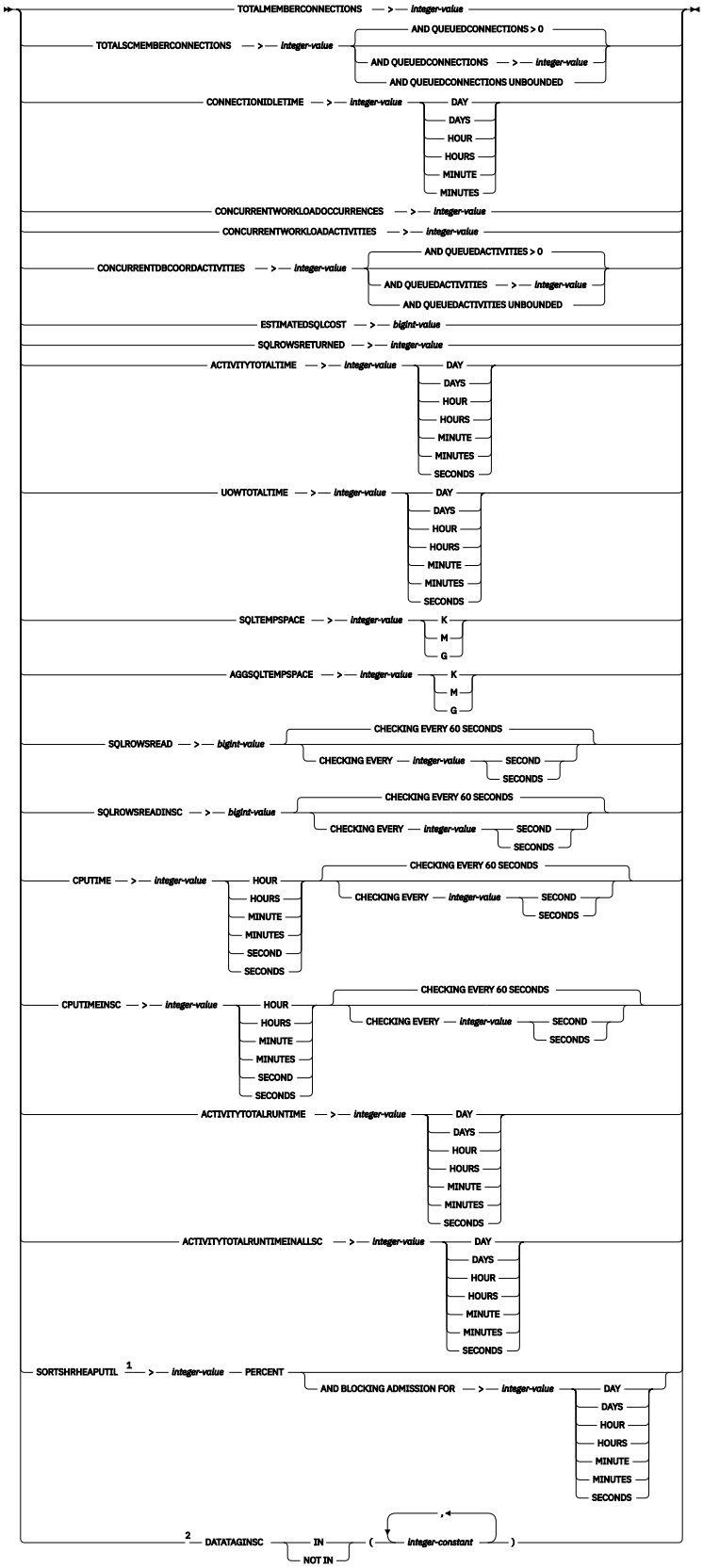
threshold-domain



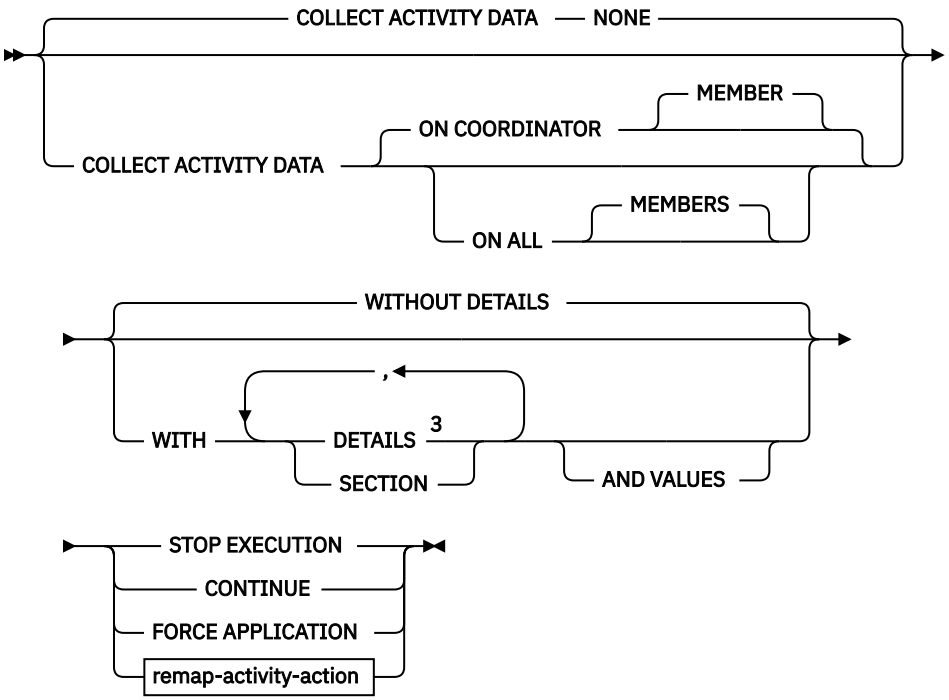
enforcement-scope



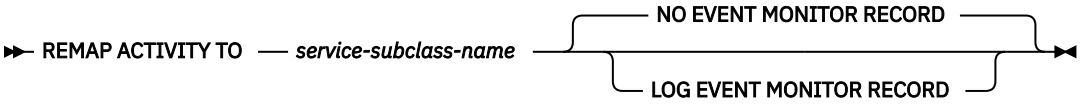
threshold-predicate



threshold-exceeded-actions



remap-activity-action



Notes:

[1] This feature is available in Db2 Version 11.5 Mod Pack 2 and later versions.

[2] Each data tag value can be specified only once.

[3] The DETAILS keyword is the minimum to be specified, followed by the option separated by a comma.

Description

threshold-name

Names the threshold. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *threshold-name* must not identify a threshold that already exists at the current server (SQLSTATE 42710). The name must not begin with the characters 'SYS' (SQLSTATE 42939).

FOR *threshold-domain* ACTIVITIES

Specifies the definition domain of the threshold.

DATABASE

This threshold applies to any activity in the database.

SERVICE CLASS *service-class-name*

This threshold applies to activities executing in service class *service-class-name*. If UNDER is not specified, *service-class-name* must identify an existing service superclass (SQLSTATE 42704). If UNDER is specified, *service-class-name* must identify an existing service subclass of the service superclass specified after the UNDER keyword (SQLSTATE 42704). The *service-class-name* cannot be the SYSDEFAULTSYSTEMCLASS service class or the SYSDEFAULTMAINTENANCECLASS service class (SQLSTATE 5U032).

UNDER *service-class-name*

Specifies a service superclass. The *service-class-name* must identify an existing service superclass (SQLSTATE 42704).

STATEMENT
This threshold applies to activities for a specific SQL statement. You identify the statement to use for the threshold by specifying the statement text or the statement's executable ID .

TEXT *statement-text*
This threshold applies to statements matching the text specified in *statement-text*. Both static and dynamic SQL statements are considered when the condition for the threshold is evaluated. At run time, the text specified for *statement-text* must be an exact match of the text of a statement in the package cache for the threshold to be violated. Differences in letter case or use of white space prevent a match from occurring between *statement-text* and any running SQL statement. The text for *statement-text* must be specified as a string constant. As such, the maximum length for the text of a statement for a statement threshold is 32 672 bytes, and not the usual 2 MB upper limit for statements.

Access plan differences do not affect statement matching. It is possible for multiple cached statements with same text but different access plans to match the threshold text defined by *statement-text*.

If a statement that otherwise matches the statement supplied for *statement-text* is altered or transformed during compilation in such a way that it differs from *statement-text*, the statements will not match. For example, if the statement concentrator is enabled, literal values might be replaced by parameter markers. No such transformation is applied to text supplied for the *statement-text* in the CREATE THRESHOLD statement. The text supplied to CREATE THRESHOLD must match exactly the transformed text of any statement of interest. You can determine the exact text of statements as they are executed using monitoring table functions such as **MON_GET_PKG_CACHE_STMT** and **MON_GET_ACTIVITY_DETAILS**.

The following predicates can be used with a statement threshold:

- **ACTIVITYTOTALRUNTIME**
- **ACTIVITYTOTALTIME**
- **CPUTIME**
- **ESTIMATEDSQLCOST**
- **SQLROWSREAD**
- **SQLROWSRETURNED**
- **SQLTEMPSPACE**

REFERENCE *executable-id*
This threshold applies to statements with text that matches the text of the statement with the specified executable ID. The database manager uses the executable ID to locate text of the statement from its section in the package cache. The text of the statement that is used for the threshold is that which was cached for the section at the time the threshold was created. For dynamic SQL, the statement referenced by the executable ID must be in the package cache. For static SQL, if the statement is not in the cache, the database manager retrieves it from the system catalogs.

Once the statement text is retrieved from the package cache, there is no direct relationship between the threshold and the specified executable ID; the cached section can even be evicted from the cache without impact on any threshold that was derived from it. Once the text associated with the executable ID is determined, the threshold created by this clause behaves in exactly the same way as one created by the STATEMENT TEXT clause.

WORKLOAD *workload-name*
This threshold applies to the specified workload. The *workload-name* must identify an existing workload (SQLSTATE 42704).

ENFORCEMENT *enforcement-scope*
The enforcement scope of the threshold.

DEFAULT
The default enforcement scope of the threshold will be used.

DATABASE
The threshold is enforced across all members within the definition domain; that is, all members of the database, and all members of the service class.

MEMBER
The threshold is enforced on a per member basis. There is no coordination across all members to enforce the threshold.

WORKLOAD OCCURRENCE
The threshold is enforced only within a workload occurrence. Two workload occurrences running concurrently on the same member will each have their own running count for this threshold.

ENABLE or DISABLE
Specifies whether or not the threshold is enabled for use by the database manager.

ENABLE
The threshold is used by the database manager to restrict the execution of database activities.

DISABLE
The threshold is not used by the database manager to restrict the execution of database activities.

WHEN *threshold-predicate*
Specifies the condition of the threshold.

TOTALMEMBERCONNECTIONS > *integer-value*
This condition defines an upper bound on the number of coordinator connections that can run concurrently on a member. This value can be zero or any positive integer (SQLSTATE 42820). A value of zero means that any new coordinator connection will be prevented from connecting. All currently running or queued connections will continue. The definition domain for this condition must be DATABASE, and the enforcement scope must be MEMBER (SQLSTATE 5U037). This threshold is not enforced for users with DBADM or WLMADM authority.

TOTALSCMEMBERCONNECTIONS > *integer-value*
This condition defines an upper bound on the number of coordinator connections that can run concurrently on a member in a specific service superclass. This value can be zero or any positive integer (SQLSTATE 42820). A value of zero means that any new connection will be prevented from joining the service class. All currently running or queued connections will continue. The definition domain for this condition must be SERVICE SUPERCLASS, and the enforcement scope must be MEMBER (SQLSTATE 5U037).

AND QUEUEDCONNECTIONS > *integer-value* or AND QUEUEDCONNECTIONS UNBOUNDED
Specifies a queue size for when the maximum number of coordinator connections is exceeded. This value can be any positive integer, including zero (SQLSTATE 42820). A value of zero means that no coordinator connections are queued. Specifying UNBOUNDED will queue every connection that exceeds the specified maximum number of coordinator connections, and the *threshold-exceeded-actions* will never be executed. The default is zero.

CONNECTIONIDLETIME > *integer-value* DAY | DAYS | HOUR | HOURS | MINUTE | MINUTES
This condition defines an upper bound for the amount of time the database manager will allow a connection to remain idle. This value can be any positive integer (not zero) (SQLSTATE 42820). Use a valid duration keyword to specify an appropriate unit of time for *integer-value*. The definition domain for this condition must be DATABASE or SERVICE SUPERCLASS, and the enforcement scope must be DATABASE (SQLSTATE 5U037). This condition is enforced at the coordinator member.

If you specify the STOP EXECUTION action with CONNECTIONIDLETIME thresholds, the connection for the application is dropped when the threshold is exceeded. Any subsequent attempt by the application to access the data server will receive SQLSTATE 5U026.

The maximum value for this threshold is 2 147 483 640 seconds. Any value specified that has a seconds equivalent larger than 2 147 483 640 seconds will be set to this number of seconds.

CONCURRENTWORKLOADOCCURRENCES > *integer-value*
This condition defines an upper bound on the number of concurrent occurrences for the workload on each member. This value can be any positive integer (not zero) (SQLSTATE 42820). The

definition domain for this condition must be WORKLOAD and the enforcement scope must be MEMBER (SQLSTATE 5U037).

CONCURRENTWORKLOADACTIVITIES > *integer-value*
This condition defines an upper bound on the number of concurrent coordinator activities and nested activities for the workload on each member. This value can be any positive integer (not zero) (SQLSTATE 42820). The definition domain for this condition must be WORKLOAD and the enforcement scope for this condition must be WORKLOAD OCCURRENCE (SQLSTATE 5U037).

Each nested activity must satisfy the following conditions:

- It must be a recognized coordinator activity. Any nested coordinator activity that does not fall within the recognized types of activities will not be counted. Similarly, nested subagent activities, such as remote node requests, are not counted.
- It must be directly invoked from user logic, such as a user-written procedure issuing SQL statements.

Internal SQL activities, such as those initiated by the setting of a constraint or the refreshing of a materialized query table, are also not counted by this threshold, because they are initiated by the database manager and not directly invoked by user logic.

CONCURRENTDBCOORDACTIVITIES > *integer-value*
This condition defines an upper bound on the number of recognized database coordinator activities that can run concurrently on all members in the specified domain. This value can be zero or any positive integer (SQLSTATE 42820). A value of zero means that any new database coordinator activities will be prevented from executing. All currently running or queued database coordinator activities will continue. The definition domain for this condition must be DATABASE, work action (a threshold for a work action definition domain is created using a CREATE WORK ACTION SET or ALTER WORK ACTION SET statement, and the work action set must be applied to a workload or a database), SERVICE SUPERCLASS, or SERVICE SUBCLASS. Also, the enforcement scope must be DATABASE (SQLSTATE 5U037) in environments other than Db2 pureScale, where the condition is enforced across the entire database, and MEMBER (SQLSTATE 5U037) in Db2 pureScale where the condition is enforced at each coordinator member. All activities are tracked by this condition, except for the following items:

- CALL statements are not controlled by this threshold, but all nested child activities started within the called routine are under this threshold's control. Anonymous blocks and autonomous routines are classified as CALL statements.
- User-defined functions are controlled by this threshold, but child activities nested in a user-defined function are not controlled. If an autonomous routine is called from within a user defined function, neither the autonomous routine nor any child activities of the autonomous routine are under threshold control.
- Trigger actions that invoke CALL statements and the child activities of these CALL statements are not controlled by this threshold. INSERT, UPDATE, or DELETE statements that can cause a trigger to activate continue to be under threshold control.
- To manage concurrency with a CALL statement, you may be able to use the TOTALSCPARTITIONCONNECTIONS threshold. The TOTALSCPARTITIONCONNECTIONS threshold is effective for controlling concurrency of CALL statements when your workload consists of transient connections. Transient connections are connections that are established only during the procedure invocation. The TOTALSCPARTITIONCONNECTIONS threshold is not appropriate if your workload consists of long-lived connections.

When a threshold is defined as part of a work action set, the enforcement scope is determined automatically based on the current environment (MEMBER, if the current environment is Db2 pureScale; DATABASE, if it is otherwise).

Important: Before using CONCURRENTDBCOORDACTIVITIES thresholds, be sure to become familiar with the effects that they can have on the database system.

For more information, refer to "CONCURRENTDBCOORDACTIVITIES threshold" in *Db2 Workload Management Guide and Reference*.

AND QUEUEDACTIVITIES > *integer-value* or AND QUEUEDACTIVITIES UNBOUNDED
Specifies a queue size for when the maximum number of database coordinator activities is exceeded. This value can be zero or any positive integer (SQLSTATE 42820). A value of zero means that no database coordinator activities are queued. Specifying UNBOUNDED will queue every database coordinator activity that exceeds the specified maximum number of database coordinator activities, and the *threshold-exceeded-actions* will never be executed. The default is zero.

Note: If a threshold action of CONTINUE is specified for a queuing threshold, it effectively makes the size of the queue unbounded, regardless of any hard value specified for the queue size.

ESTIMATEDSQLCOST > *bigint-value*
This condition defines an upper bound for the optimizer-assigned cost (in timerons) of an activity. This value can be any positive big integer (not zero) (SQLSTATE 42820). The definition domain for this condition must be DATABASE, work action (a threshold for a work action definition domain is created using a CREATE WORK ACTION SET or ALTER WORK ACTION SET statement, and the work action set must be applied to a workload or a database), SERVICE SUPERCLASS, SERVICE SUBCLASS, or WORKLOAD, and the enforcement scope must be DATABASE (SQLSTATE 5U037). This condition is enforced at the coordinator member. Activities tracked by this condition are:

- Coordinator activities of type data manipulation language (DML).
- Nested DML activities that are invoked from user logic. Consequently, DML activities that can be initiated by the database manager through internal SQL are not tracked by this condition (unless their cost is included in the parent's estimate, in which case they are indirectly tracked).

SQLROWSRETURNED > *integer-value*
This condition defines an upper bound for the number of rows returned to a client application from the application server. This value can be any positive integer (not zero) (SQLSTATE 42820). The definition domain for this condition must be DATABASE, work action (a threshold for a work action definition domain is created using a CREATE WORK ACTION SET or ALTER WORK ACTION SET statement, and the work action set must be applied to a workload or a database), SERVICE SUPERCLASS, SERVICE SUBCLASS, or WORKLOAD, and the enforcement scope must be DATABASE (SQLSTATE 5U037). This condition is enforced at the coordinator member. Activities tracked by this condition are:

- Coordinator activities of type DML.
- Nested DML activities that are derived from user logic. Activities that are initiated by the database manager through internal SQL are not affected by this condition.

Result sets returned from within a procedure are treated separately as individual activities. There is no aggregation of the rows that are returned by the procedure itself.

ACTIVITYTOTALTIME > *integer-value* DAY | DAYS | HOUR | HOURS | MINUTE | MINUTES | SECONDS
This condition defines an upper bound for the amount of time the database manager will allow an activity to execute, including the time the activity was queued. The definition domain for this condition must be DATABASE, work action (a threshold for a work action definition domain is created using a CREATE WORK ACTION SET or ALTER WORK ACTION SET statement, and the work action set must be applied to a workload or a database), SERVICE SUPERCLASS, SERVICE SUBCLASS, or WORKLOAD, and the enforcement scope must be DATABASE (SQLSTATE 5U037). This condition is logically enforced at the coordinator member.

The specified *integer-value* must be an integer that is greater than zero (SQLSTATE 42820). Use a valid duration keyword to specify an appropriate unit of time for *integer-value*. If the specified time unit is SECONDS, the value must be a multiple of 10 (SQLSTATE 42615). The maximum value that can be specified for this threshold is 2 147 483 640 seconds. If any value (using the DAY, HOUR, MINUTE, or SECONDS time unit) has a seconds equivalent larger than the maximum value, an error is returned (SQLSTATE 42615).

UOWTOTALTIME > *integer-value* DAY | DAYS | HOUR | HOURS | MINUTE | MINUTES | SECONDS

This condition defines an upper bound for the amount of time the database manager will allow a unit of work to execute. This value can be any non-zero positive integer (SQLSTATE 42820). Use a valid duration keyword to specify an appropriate unit of time for *integer-value*. If the specified time unit is SECONDS, the value must be a multiple of 10 (SQLSTATE 42615). The definition domain for this condition must be DATABASE, SERVICE SUPERCLASS, or WORKLOAD, and the enforcement scope must be DATABASE (SQLSTATE 5U037). This condition is enforced at the coordinator member.

The maximum value that can be specified for this threshold is 2 147 483 640 seconds. If any value (using the DAY, HOUR, MINUTE, or SECONDS time unit) has a seconds equivalent larger than the maximum value, an error is returned (SQLSTATE 42615).

SQLTEMPSPACE > *integer-value* K | M | G

This condition defines the maximum amount of system temporary space that can be consumed by an SQL statement on a member. This value can be any positive integer (not zero) (SQLSTATE 42820).

If *integer-value K* (in either upper- or lowercase) is specified, the maximum size is 1024 times *integer-value*. If *integer-value M* is specified, the maximum size is 1 048 576 times *integer-value*. If *integer-value G* is specified, the maximum size is 1 073 741 824 times *integer-value*.

The definition domain for this condition must be DATABASE, work action (a threshold for a work action definition domain is created using a CREATE WORK ACTION SET or ALTER WORK ACTION SET statement, and the work action set must be applied to a workload or a database), SERVICE SUPERCLASS, SERVICE SUBCLASS, or WORKLOAD, and the enforcement scope must be MEMBER (SQLSTATE 5U037). Activities tracked by this condition are:

- Coordinator activities of type DML and corresponding subagent work (subsection execution).
- Nested DML activities that are derived from user logic and their corresponding subagent work (subsection execution). Activities that are initiated by the database manager through an internal SQL are not affected by this condition.

AGGSQLTEMPSPACE > *integer-value* K | M | G

This condition defines the maximum amount of system temporary space that can be consumed by a set of statements in a service class on a member. This value can be any positive integer (not zero) (SQLSTATE 42820).

If *integer-value* K (in either upper- or lowercase) is specified, the maximum size is 1024 times *integer-value*. If *integer-value* M is specified, the maximum size is 1 048 576 times *integer-value*. If *integer-value* G is specified, the maximum size is 1 073 741 824 times *integer-value*.

The definition domain for this condition must be SERVICE SUBCLASS and the enforcement scope must be MEMBER (SQLSTATE 5U037).

Activities contributing to the aggregate that is tracked by this condition are:

- Coordinator activities of type DML and corresponding subagent work like subsection execution.
- Nested DML activities that are derived from user logic and their corresponding subagent work like subsection execution. Activities initiated by the database manager through an internal SQL statement are not affected by this condition.

SQLROWSREAD > *bigint-value*

This condition defines an upper bound on the number of rows that may be read by an activity during its lifetime on a particular member. This value can be any positive big integer (not zero) (SQLSTATE 42820). Note that the number of rows read is different from the number of rows returned, which is controlled by the SQLROWSRETURNED condition.

The definition domain for this condition must be DATABASE, SERVICE CLASS, a service subclass (SERVICE CLASS specifying the UNDER clause), WORKLOAD or a work action (a threshold for a work action definition domain is created using a CREATE WORK ACTION SET or ALTER WORK ACTION SET statement, and the work action set must be applied to a workload or a database),

and the enforcement scope must be MEMBER (SQLSTATE 5U037). This condition is enforced independently at each member.

Activities tracked by this condition are:

- Coordinator activities of type DML and corresponding subagent work (like subsection execution).
- Internal SQL activities like those initiated by the setting of a constraint, or the refreshing of a materialized query table, are also not tracked by this threshold, because they are initiated by the database manager and not directly invoked by user logic.

CHECKING EVERY *integer-value* SECOND | SECONDS
Specifies how frequently the threshold condition is checked for an activity. The threshold is checked at the end of each request (like a fetch operation, for example) and on the interval defined by the CHECKING clause. The CHECKING clause defines an upper bound on how long a threshold violation may go undetected. The default is 60 seconds. The value can be any positive integer (not zero) with a maximum value of 86400 seconds (SQLSTATE 42820). Setting a low value may impact system performance negatively.

SQLROWSREADINSC > *bigint-value*
This condition defines an upper bound on the number of rows that may be read by an activity on a particular member while it is executing in a service subclass. Rows read before executing in the service subclass specified are not counted. This value can be any positive big integer (not zero) (SQLSTATE 42820). Note that the number of rows read is different from the number of rows returned, which is controlled by the SQLROWSRETURNED condition.

The definition domain for this condition must be a service subclass (SERVICE CLASS specifying the UNDER clause) and the enforcement scope must be MEMBER (SQLSTATE 5U037). This condition is enforced independently at each member.

Activities tracked by this condition are:

- Coordinator activities of type DML and corresponding subagent work (like subsection execution).
- Internal SQL activities like those initiated by the setting of a constraint, or the refreshing of a materialized query table, are also not tracked by this threshold, because they are initiated by the database manager and not directly invoked by user logic.

CHECKING EVERY *integer-value* SECOND | SECONDS
Specifies how frequently the threshold condition is checked for an activity. The threshold is checked at the end of each request (like a fetch operation, for example) and on the interval defined by the CHECKING clause. The CHECKING clause defines an upper bound on how long a threshold violation may go undetected. The default is 60 seconds. The value can be any positive integer (not zero) with a maximum value of 86400 seconds (SQLSTATE 42820).Setting a low value may impact system performance negatively.

CPUTIME > *integer-value* HOUR | HOURS | MINUTE | MINUTES | SECOND | SECONDS
This condition defines an upper bound for the amount of processor time that an activity may consume during its lifetime on a particular member. The processor time tracked by this threshold is measured from the time that the activity starts executing. This value can be any positive integer (not zero) (SQLSTATE 42820).

The definition domain for this condition must be DATABASE, a service superclass (SERVICE CLASS), a service subclass (SERVICE CLASS specifying the UNDER clause), WORKLOAD or work action (a threshold for a work action definition domain is created using a CREATE WORK ACTION SET or ALTER WORK ACTION SET statement, and the work action set must be applied to a workload or a database), and the enforcement scope must be MEMBER (SQLSTATE 5U037). This condition is enforced independently at each member.

Activities tracked by this condition are:

- Coordinator activities of type DML and corresponding subagent work (like subsection execution).
- Internal SQL activities, like those initiated by the setting of a constraint or the refreshing of a materialized query table, are also not tracked by this threshold, because they are initiated by the database manager and not directly invoked by user logic.

- Activities of type CALL. For CALL activities, the processor time tracked for the procedure does not include the processor time used by any child activities or by any fenced mode processes. The threshold condition will be checked only upon return from user logic to the database engine. For example: During the execution of a trusted routine, the threshold condition will be checked only when the routine issues a request to the database engine).

CHECKING EVERY *integer-value* SECOND | SECONDS
Specifies how frequently the threshold condition is checked for an activity. The granularity of the CPUTIME threshold is approximately this number multiplied by the degree of parallelism for the activity. For example: If the threshold is checked every 60 seconds and the degree of parallelism is 2, the activity might use an extra 2 minutes of processor time instead of 1 minute before the threshold violation is detected. The default is 60 seconds. The value can be any positive integer (not zero) with a maximum value of 86400 seconds (SQLSTATE 42820). Setting a low value may impact system performance negatively.

CPUTIMEINSC > *integer-value* HOUR | HOURS | MINUTE | MINUTES | SECOND | SECONDS
This condition defines an upper bound for the amount of processor time that an activity may consume on a particular member while it is executing in a particular service subclass. The processor time tracked by this threshold is measured from the time that the activity starts executing in the service subclass identified in the threshold domain. Any processor time used before that point is not counted toward the limit imposed by this threshold. This value can be any positive integer (not zero) (SQLSTATE 42820).

The definition domain for this condition must be a service subclass (SERVICE CLASS specifying the UNDER clause), and the enforcement scope must be MEMBER (SQLSTATE 5U037). This condition is enforced independently at each member.

Activities tracked by this condition are:

- Coordinator activities of type DML and corresponding subagent work (like subsection execution).
- Internal SQL activities, like those initiated by the setting of a constraint or the refreshing of a materialized query table, are also not tracked by this threshold, because they are initiated by the database manager and not directly invoked by user logic.
- Activities of type CALL. For CALL activities, the processor time tracked for the procedure does not include the processor time used by any child activities or by any fenced mode processes. The threshold condition will be checked only upon return from user logic to the database engine. For example: During the execution of a trusted routine, the threshold condition will be checked only when the routine issues a request to the database engine).

CHECKING EVERY *integer-value* SECOND | SECONDS
Specifies how frequently the threshold condition is checked for an activity. The granularity of the CPUTIMEINSC threshold is approximately this number multiplied by the degree of parallelism for the activity. For example: If the threshold is checked every 60 seconds and the degree of parallelism is 2, the activity might use an extra 2 minutes of processor time instead of 1 minute before the threshold violation is detected. The default is 60 seconds. The value can be any positive integer (not zero) with a maximum value of 86400 seconds (SQLSTATE 42820). Setting a low value may impact system performance negatively.

ACTIVITYTOTALRUNTIME > *integer-value* DAY | DAYS | HOUR | HOURS | MINUTE | MINUTES | SECONDS
This condition is used to define an upper bound for the amount of time the database manager allows an activity to run. The amount of time does not include the time that the activity was queued by a WLM concurrency threshold. The definition domain for this condition must be one of the following thresholds (SQLSTATE 5U037):

- Database
- Service superclass
- Service subclass
- Statement
- Workload

- Work action [1]

1. A threshold for a work action definition domain is created by using a CREATE WORK ACTION SET or ALTER WORK ACTION SET statement. The work action set must be applied to a workload or a database.

The enforcement scope must be DATABASE (SQLSTATE 5U037).

The specified *integer-value* must be an integer that is greater than zero (SQLSTATE 42820). Use a valid duration keyword to specify an appropriate unit of time for *integer-value*. If the specified time unit is SECONDS, the value must be a multiple of 10 (SQLSTATE 42615). The maximum value that can be specified for this threshold is 2 147 483 640 seconds. If any value for the DAY, HOUR, MINUTE, or SECONDS time unit has a seconds equivalent larger than the maximum value, an error is returned (SQLSTATE 42615).

ACTIVITYTOTALRUNTIMEINALLSC > *integer-value* DAY | DAYS | HOUR | HOURS | MINUTE | MINUTES | SECONDS

This condition is used to define an upper bound for the amount of time the database manager allows an activity to run. The amount of time does not include the time that the activity was queued by a WLM concurrency threshold. The execution time that is tracked by this threshold is measured from the time that the activity starts running.

The definition domain for this condition must be service subclass (SERVICE CLASS specifying the UNDER clause), and the enforcement scope must be DATABASE (SQLSTATE 5U037).

The specified *integer-value* must be an integer that is greater than zero (SQLSTATE 42820). Use a valid duration keyword to specify an appropriate unit of time for *integer-value*. If the specified time unit is SECONDS, the value must be a multiple of 10 (SQLSTATE 42615). The maximum value that can be specified for this threshold is 2 147 483 640 seconds. If any value for the DAY, HOUR, MINUTE, or SECONDS time unit has a seconds equivalent larger than the maximum value, an error is returned (SQLSTATE 42615).

SORTSHRHEAPUTIL > *integer-value* PERCENT

Attention: This feature is available in Db2 Version 11.5 Mod Pack 2 and later versions.

This condition defines the maximum shared sort memory that may be requested by a query as a percentage of the total database shared sort memory (sheapthres_shr). When the adaptive workload manager is enabled, the threshold considers both estimated and actual memory requirements for a query. Any positive integer between 1 to 100 can be specified as a percent value. The execution time that is tracked by this threshold is measured from the time that the activity starts running.

The definition domain for this condition must be DATABASE, work action (a threshold for a work action definition domain is created using a CREATE WORK ACTION SET or ALTER WORK ACTION SET statement, and the work action set must be applied to a workload or a database), SERVICE SUPERCLASS, SERVICE SUBCLASS, STATEMENT or WORKLOAD, and the enforcement scope must be MEMBER (SQLSTATE 5U037).

Activities tracked by this condition are:

- Coordinator activities of type data manipulation language (DML).
- Nested DML activities that are directly invoked from user logic.

AND BLOCKING ADMISSION FOR *integer-value*

Specifies that action will only be taken if the sort memory requirements are exceeded, work is currently queued behind the violating activity, the queued work is blocking on sort memory and WLM admission control has not admitted any requests for the specified amount of time. For work inside the WLM admission queue, this condition will only be evaluated once a request reaches the front of the admission queue. Every time a request is allowed by admission control, the queue time will be resset. If multiple requests violate this threshold a cascading effect will be observed until something that doesn't violate this threshold is found or the last request is reached (as in, no other requests behind).

The maximum value for this threshold is 2147483640 seconds. Any value specified that has a seconds equivalent larger than 2147483640 seconds will be set to this number of seconds. The time specified has a minimum accuracy of 10 seconds, so any value specified is subject to accuracy of this amount. A value of zero is equivalent to not specifying a BLOCKING ADMISSION FOR clause.

DATATAGINSC IN *(integer-constant, ...)*

This condition defines one or more data tag values specified on a table space that the activity touches. The data tag on a table space, or its underlying storage group (where applicable), can be either not be set or set to a value from 1 to 9. If the activity touches a table space that has no data tag set (at either the table space or the storage group level), this threshold will not have any affect on that activity. The definition domain for this condition must be a service subclass (SERVICE CLASS specifying the UNDER clause), and the enforcement scope must be DATABASE PARTITION (SQLSTATE 5U037). This condition is enforced independently at each database partition.

Activities tracked by this condition are:

- Coordinator activities of type data manipulation language (DML).
- Nested DML activities that are directly invoked from user logic.

DML activities that can be initiated by the database manager through internal SQL are not tracked by this condition.

This threshold is checked only when a scan is opened on a table or when an insert is performed into a table. Fetching data from a table after a scan has been opened will not violate the threshold.

DATATAGINSC NOT IN *(integer-constant, ...)*

This condition defines one or more data tag values not specified on a table space that the activity touches. The data tag on a table space, or its underlying storage group (where applicable), can be either not be set or set to a value from 1 to 9. If the activity touches a table space that has no data tag set (either at the table space or the storage group level), this threshold will not have any affect on that activity. The definition domain for this condition must be a service subclass (SERVICE CLASS specifying the UNDER clause) and the enforcement scope must be DATABASE PARTITION (SQLSTATE 5U037). This condition is enforced independently at each database partition.

Activities tracked by this condition are:

- Coordinator activities of type data manipulation language (DML).
- Nested DML activities that are directly invoked from user logic.

DML activities that can be initiated by the database manager through internal SQL are not tracked by this condition.

This threshold is checked only when a scan is opened on a table or when an insert is performed into a table. Fetching data from a table after a scan has been opened will not violate the threshold.

threshold-exceeded-actions

Specifies what action is to be taken when a condition is exceeded. Each time that a condition is exceeded, an event is recorded in the threshold violations event monitor, if one is active.

COLLECT ACTIVITY DATA

Specifies that data about each activity that exceeded the threshold is to be sent to any active activities event monitor, when the activity completes. The default is COLLECT ACTIVITY DATA NONE. If COLLECT ACTIVITY DATA is specified, the default is WITHOUT DETAILS. The COLLECT ACTIVITY DATA setting does not apply to non-activity thresholds, such as the following: CONNECTIONIDLETIME, TOTALDBPARTITIONCONNECTIONS, TOTALSCPARTITIONCONNECTIONS, CONCURRENTWORKLOADOCCURRENCES, UOWTOTALTIME.

NONE

Specifies that activity data should not be collected for each activity that exceeds the threshold.

ON COORDINATOR MEMBER

Specifies that the activity data is to be collected only at the coordinator member of the activity.

ON ALL MEMBERS
Specifies that the activity data is to be collected at all members on which the activity is processed. On remote members, a record for the activity may be captured multiple times as the activity comes and goes on those members. For predictive thresholds, activity information is collected at all members only if you also specify the CONTINUE action for exceeded thresholds. For reactive thresholds, the ON ALL MEMBERS clause has no effect and activity information is always collected only at the coordinator member. For both predictive and reactive thresholds, any activity details, section information, or values will be collected only at the coordinator member.

WITHOUT DETAILS
Specifies that data about each activity associated with the work class for which this work action is defined is to be sent to any active activities event monitor, when the activity completes execution. Details about statement, compilation environment, and section environment data are not sent.

WITH

DETAILS
Specifies that statement and compilation environment data is to be sent to any active activities event monitor, for those activities that have them. Section environment data is not sent.

SECTION
Specifies that statement, compilation environment, section environment data, and section actuals are to be sent to any active activities event monitor for those activities that have them. DETAILS must be specified if SECTION is specified. For predictive thresholds, section actuals will be collected on any member where the activity data is collected. For reactive thresholds, section actuals will be collected only on the coordinator member.

AND VALUES
Specifies that input data values are to be sent to any active activities event monitor, for those activities that have them. This data does not include SQL statements that are compiled by using the REOPT ALWAYS bind option.

STOP EXECUTION
The execution of the activity is stopped and an error is returned (SQLSTATE 5U026). In the case of the UOWTOTALTIME threshold, the unit of work is rolled back.

CONTINUE
The execution of the activity is not stopped.

FORCE APPLICATION
The application is forced off the system (SQLSTATE 55032). This action can only be specified for the UOWTOTALTIME threshold.

remap-activity-action

REMAP ACTIVITY TO *service-subclass-name*
The activity is mapped to *service-subclass-name*. The execution of the activity is not stopped. This action is valid only for in-service-class and in-all-service-class thresholds like CPUTIMEINSC, SQLROWSREADINSC, DATATAGINSC IN and DATATAGINSC NOT IN and ACTIVITYTOTALRUNTIMEINALLSC thresholds (SQLSTATE 5U037). The service-subclass-name must identify an existing service subclass under the same superclass associated with the threshold (SQLSTATE 5U037). The service-subclass-name cannot be the same as the associated service subclass of the threshold (SQLSTATE 5U037).

NO EVENT MONITOR RECORD
Specifies that no threshold violation record will be written.

LOG EVENT MONITOR RECORD
Specifies that if a THRESHOLD VIOLATIONS event monitor exists and is active, a threshold violation record is written to it.

Notes

- Thresholds can be defined on different aspects of database behavior to monitor and control that behavior. When a threshold is defined on activities, unless otherwise specified, it will be enforced only during the actual execution of SQL statements, not including compilation time, and the load utility.
- The CONCURRENTWORKLOADOCCURRENCES threshold and the CONCURRENTWORKLOADACTIVITIES threshold differ in scope. CONCURRENTWORKLOADOCCURRENCES controls how many connections can map to a workload definition simultaneously, and CONCURRENTWORKLOADACTIVITIES controls how many activities each connection that is mapped to the workload definition can submit concurrently.
- Changes are written to the system catalog, but do not take effect until after a COMMIT statement, even for the connection that issues the statement.
- Only one uncommitted WLM-exclusive SQL statement at a time is allowed across all partitions. If an uncommitted WLM-exclusive SQL statement is executing, subsequent WLM-exclusive SQL statements will wait until the current WLM-exclusive SQL statement commits or rolls back.
- ***Threshold exceeded action of CONTINUE and event monitor data:*** Event monitor data is collected only once per member when a threshold condition has been exceeded. If the threshold exceeded action is CONTINUE, the activity continues executing and no further event monitor data is collected for that threshold at the affected member. For example, consider a time threshold of 10 minutes with an action of CONTINUE. After an activity exceeds the 10-minute upper bound, event monitor data is collected for the threshold at the affected member.
- ***Quiescing a service class:*** The TOTALSCPARTITIONCONNECTIONS threshold condition can be used to simulate quiescing service classes that cannot normally be quiesced (for example, the default user class, or the default system class). This is useful, because thresholds do not apply to users with DBADM authority running in the SYSDEFAULTADMWORKLOAD, whereas a quiesced service class is not available to anyone. Consequently, default service classes cannot be quiesced directly but only through a threshold that allows users with DBADM authority to join them when connected to the database using the SYSDEFAULTADMWORKLOAD.
- ***Syntax alternatives***: The following are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - DATABASE PARTITION can be specified in place of MEMBER, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
  - DATABASE PARTITIONS can be specified in place of MEMBERS, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
  - TOTALDBPARTITIONCONNECTIONS can be specified in place of TOTALMEMBERCONNECTIONS, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
  - TOTALSCPARTITIONCONNECTIONS can be specified in place of TOTALSCMEMBERCONNECTIONS, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.

Examples

- *Example 1:* Create a threshold that enforces a maximum temporary table space usage of 50M (per database partition) to any activity in the database. Any activity that violates this threshold is to be stopped.

```
CREATE THRESHOLD DBMAX50MEGTEMPSPACE
  FOR DATABASE ACTIVITIES
  ENFORCEMENT MEMBER
  WHEN SQLTEMPSPACE > 50 M
  STOP EXECUTION
```

- *Example 2:* Create a second threshold to limit the default runtime of any activity in the database to a maximum of 1 hour. Any activity that violates this threshold is to be stopped.

```
CREATE THRESHOLD DBMAX1HOURRUNTIME
  FOR DATABASE
  WHEN ACTIVITYTOTALTIME > 1 HOUR
  STOP EXECUTION
```

- *Example 3:* Assume that a service superclass named BIGQUERIES was created to host queries using more temporary space than average and running longer than 1 hour. The thresholds defined inside this service class will override the values that were set in the previous example at the database level. Note how activities violating the thresholds inside this superclass are allowed to continue executing, but detailed information is collected for further analysis.

```
CREATE THRESHOLD BIGQUERIESMAX500MEGTEMPSPACE
  FOR SERVICE CLASS BIGQUERIES ACTIVITIES
  ENFORCEMENT DATABASE MEMBER
  WHEN SQLTEMPSPACE  > 500 M
  COLLECT ACTIVITY DATA WITH DETAILS AND VALUES
  CONTINUE

CREATE THRESHOLD BIGQUERIESLONGRUNNINGTIME
  FOR SERVICE CLASS BIGQUERIES ACTIVITIES
  ENFORCEMENT DATABASE
  WHEN ACTIVITYTOTALTIME   > 10 HOURS
  COLLECT ACTIVITY DATA WITH DETAILS AND VALUES
  CONTINUE
```

- *Example 4:* Assuming the existence of a workload named PAYROLL, create a threshold that enforces the maximum number of activities within the workload to be less than or equal to 10.

```
CREATE THRESHOLD MAXACTIVITIESINPAYROLL
  FOR WORKLOAD PAYROLL ACTIVITIES
  ENFORCEMENT WORKLOAD OCCURRENCE
  WHEN CONCURRENTWORKLOADACTIVITIES > 10
  STOP EXECUTION
```

- *Example 5:* Create a threshold that enforces a maximum concurrency of 2 activities in the service class BIGQUERIES.

```
CREATE THRESHOLD MAXBIGQUERIESCONCURRENCY
  FOR SERVICE CLASS BIGQUERIES  ACTIVITIES
  ENFORCEMENT DATABASE
  WHEN CONCURRENTDBCOORDACTIVITIES > 2
  STOP EXECUTION
```

- *Example 6:* Create a threshold that captures activity information for a specific statement that runs for longer than one minute, but do not cease statement execution.

```
CREATE THRESHOLD TH1
  FOR STATEMENT
  TEXT 'SELECT DISTINCT PARTS_BIN FROM STOCK WHERE PART_NUMBER = ?'
  ACTIVITIES ENFORCEMENT DATABASE
  WHEN ACTIVITYTOTALTIME > 1 MINUTE
  COLLECT ACTIVITY DATA WITH DETAILS, SECTION AND VALUES
  CONTINUE
```

CREATE TRANSFORM

The CREATE TRANSFORM statement defines transformation functions, identified by a group name, that are used to exchange structured type values with host language programs and with external functions.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

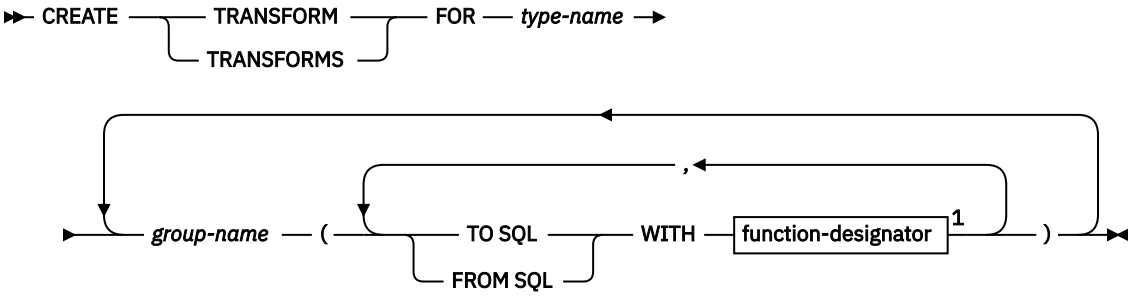
Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

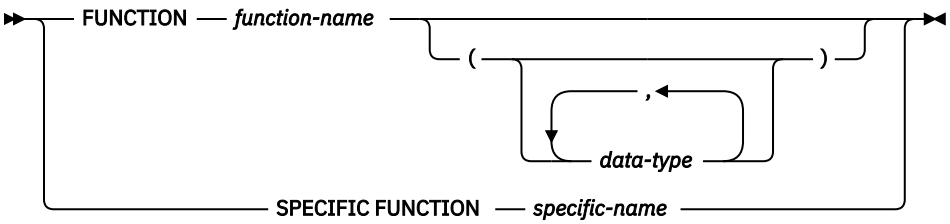
- Owner of the type identified by *type-name*, and EXECUTE privilege on every specified function

- Owner of the type identified by type-name, and EXECUTEIN privilege on the schema containing all the specified functions
- Owner of the type identified by type-name, and DATAACCESS authority on the schema containing all the specified functions
- SCHEMAADM on the schema containing the type-name
- DBADM authority

Syntax



function-designator



Notes:

[1] The same clause must not be specified more than once.

Description

TRANSFORM or TRANSFORMS
Indicates that one or more transform groups is being defined. Either version of the keyword can be specified.

FOR *type-name*
Specifies a name for the user-defined structured type for which the transform group is being defined.

In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified *type-name*. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for an unqualified *type-name*. The *type-name* must be the name of an existing user-defined type (SQLSTATE 42704), and it must be a structured type (SQLSTATE 42809). The structured type or any other structured type in the same type hierarchy must not have transforms already defined with the given group-name (SQLSTATE 42739).

group-name
Names the transform group. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *group-name* must not identify a transform group that already exists in the catalog for the specified *type-name* (SQLSTATE 42739). The *group-name* must not begin with the characters 'SYS' (SQLSTATE 42939). At most, one of each of the FROM SQL and TO SQL function designations can be specified for any given group (SQLSTATE 42628).

TO SQL
Defines the specific function used to transform a value to the SQL user-defined structured type format. The function must have all its parameters as built-in data types and the returned type is *type-name*.

FROM SQL
Defines the specific function used to transform a value to a built in data type value representing the SQL user-defined structured type. The function must have one parameter of data type *type-name*, and return a built-in data type (or set of built-in data types).

WITH *function-designator*
Uniquely identifies the transform function.

If FROM SQL is specified, *function-designator* must identify a function that meets the following requirements:

- There is one parameter of type *type-name*.
- The return type is a built-in type, or a row whose columns all have built-in types.
- The signature specifies either LANGUAGE SQL or the use of another FROM SQL transform function that has LANGUAGE SQL.

If TO SQL is specified, *function-designator* must identify a function that meets the following requirements:

- All parameters have built-in types.
- The return type is *type-name*.
- The signature specifies either LANGUAGE SQL or the use of another TO SQL transform function that has LANGUAGE SQL.

If *function-designator* identifies a function that does not meet these requirements (according to its use as a FROM SQL or a TO SQL transform function), an error is raised (SQLSTATE 428DC).

Methods (even if specified with FUNCTION ACCESS) cannot be specified as transforms through *function-designator*. Instead, only functions that are defined by the CREATE FUNCTION statement can act as transforms (SQLSTATE 42704 or 42883).

For more information, see "Function, method, and procedure designators" on page 745.

Rules

- The one or more built-in types that are returned from the FROM SQL function should directly correspond to the one or more built-in types that are parameters of the TO SQL function. This is a logical consequence of the inverse relationship between these two functions.

Notes

- When a transform group is not specified in an application program (using the TRANSFORM GROUP precompile or bind option for static SQL, or the SET CURRENT DEFAULT TRANSFORM GROUP statement for dynamic SQL), the transform functions in the transform group 'DB2_PROGRAM' are used (if defined) when the application program is retrieving or sending host variables that are based on the user-defined structured type identified by *type-name*. When retrieving a value of data type *type-name*, the FROM SQL transform is invoked to transform the structured type to the built-in data type returned by the transform function. Similarly, when sending a host variable that will be assigned to a value of data type *type-name*, the TO SQL transform is invoked to transform the built-in data type value to the structured type value. If a user-defined transform group is not specified, or a 'DB2_PROGRAM' group is not defined (for the given structured type), an error is raised (SQLSTATE 42741).
- The built-in data type representation for a structured type host variable must be assignable:
  - from the result of the FROM SQL transform function for the structured type as defined by the specified TRANSFORM GROUP option of the precompile command (using retrieval assignment rules) and
  - to the parameter of the TO SQL transform function for the structured type as defined by the specified TRANSFORM GROUP option of the precompile command (using storage assignment rules).

If a host variable is not assignment compatible with the type required by the applicable transform function, an error is raised (for bind-in: SQLSTATE 42821; for bind-out: SQLSTATE 42806). For errors that result from string assignments, see "String Assignments".

- The transform functions identified in the default transform group named 'DB2_FUNCTION' are used whenever a user-defined function not written in SQL is invoked using the data type *type-name* as a parameter or returns type. This applies when the function does not specify the TRANSFORM GROUP clause. When invoking the function with an argument of data type *type-name*, the FROM SQL transform is executed to transform the structured type to the built-in data type returned by the transform function. Similarly, when the returns data type of the function is of data type *type-name*, the TO SQL transform is invoked to transform the built-in data type value returned from the external function program into the structured type value.
- If a structured type contains an attribute that is also a structured type, the associated transform functions must recursively expand (or assemble) all nested structured types. This means that the results or parameters of the transform functions consist only of the set of built-in types representing all base attributes of the subject structured type (including all its nested structured types). There is no "cascading" of transform functions for handling nested structured types.
- The functions identified in this statement are resolved according to the rules outlined previously at the execution of this statement. When these functions are used (implicitly) in subsequent SQL statements, they do not undergo another resolution process. The transform functions defined in this statement are recorded exactly as they are resolved in this statement.
- When attributes or subtypes of a given type are created or dropped, the transform functions for the user-defined structured type must also be changed.
- For a given transform group, the FROM SQL and TO SQL transforms can be specified in either the same *group-name* clause, in separate *group-name* clauses, or in separate CREATE TRANSFORM statements. The only restriction is that a given FROM SQL or TO SQL transform designation may not be redefined without first dropping the existing group definition. This allows you to define, for example, a FROM SQL transform for a given group first, and the corresponding TO SQL transform for the same group at a later time.

Example

Create two transform groups that associate the user-defined structured type polygon with transform functions customized for C and Java, respectively.

```
CREATE TRANSFORM FOR POLYGON
  mystruct1 (FROM SQL WITH FUNCTION myxform_sqlstruct,
             TO SQL WITH FUNCTION myxform_structsql)
  myjava1   (FROM SQL WITH FUNCTION myxform_sqljava,
             TO SQL WITH FUNCTION myxform_javasql)
```

CREATE TRIGGER

The CREATE TRIGGER statement defines a trigger in the database. Triggers can be created to support general forms of integrity or business rules. A trigger defines a set of actions that are executed with, or triggered by, an INSERT, UPDATE, or DELETE statement.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- ALTER privilege on the table on which the BEFORE or AFTER trigger is defined

- CONTROL privilege on the view on which the INSTEAD OF trigger is defined
- Owner of the view on which the INSTEAD OF trigger is defined
- ALTERIN privilege on the schema of the table or view on which the trigger is defined
- SCHEMAADM authority on the schema containing the table or view on which the trigger is defined
- DBADM authority

and one of:

- IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the trigger does not exist
- CREATEIN privilege on the schema, if the schema name of the trigger refers to an existing schema
- SCHEMAADM authority on the schema, if the schema name of the trigger refers to an existing schema
- DBADM authority

If the authorization ID of the statement does not have DATAACCESS authority, the privileges (excluding group privileges) held by the authorization ID of the statement must include all of the following authorities, as long as the trigger exists:

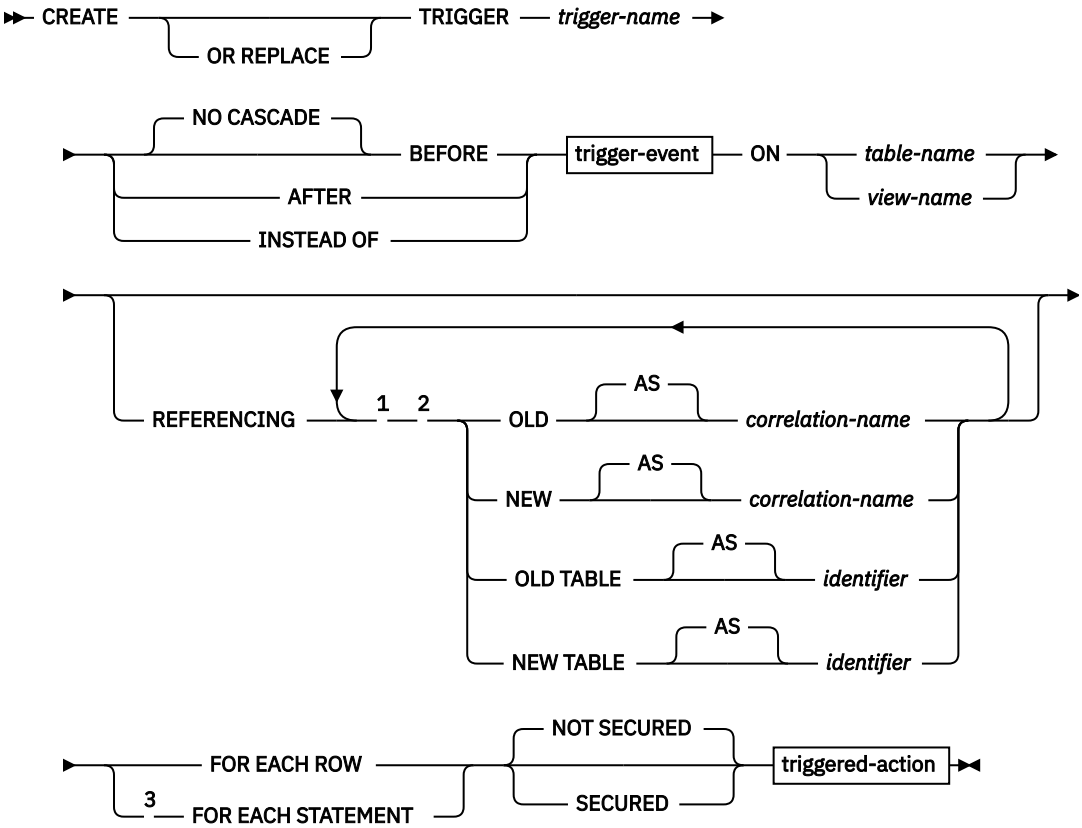
- On the table on which the trigger is defined, if any transition variables or tables are specified:
  - SELECT privilege on the table on which the trigger is defined, if any transition variables or tables are specified
  - SELECTIN privilege on the schema containing the table on which the trigger is defined, if any transition variables or tables are specified
  - CONTROL privilege on the table on which the trigger is defined, if any transition variables or tables are specified
  - DATAACCESS authority on the schema containing the table on which the trigger is defined, if any transition variables or tables are specified
  - DATAACCESS authority
- On any table or view referenced in the triggered action condition:
  - SELECT privilege on any table or view referenced in the triggered action condition
  - SELECTIN privilege on the schema containing any table or view referenced in the triggered action condition
  - CONTROL privilege on any table or view referenced in the triggered action condition
  - DATAACCESS authority on the schema containing any table or view referenced in the triggered action condition
  - DATAACCESS authority
- Necessary privileges to invoke the triggered SQL statements specified.

Group privileges are not considered for any table or view specified in the CREATE TRIGGER statement.

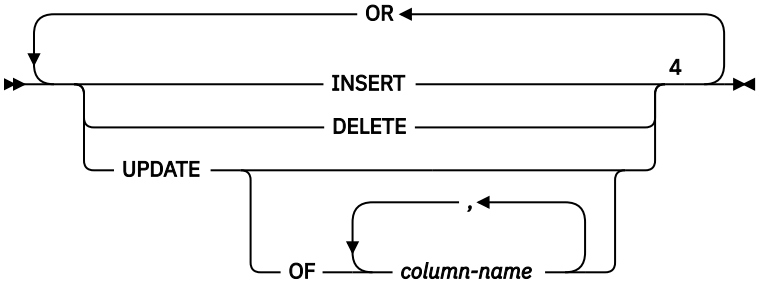
To replace an existing trigger, the authorization ID of the statement must be the owner of the existing trigger (SQLSTATE 42501).

If the SECURED option is specified, the privileges held by the authorization ID of the statement must additionally include SECADM or CREATE_SECURE_OBJECT authority (SQLSTATE 42501).

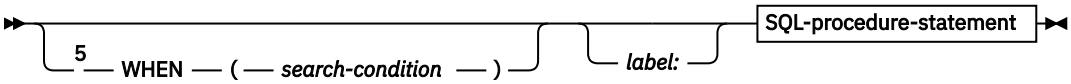
Syntax



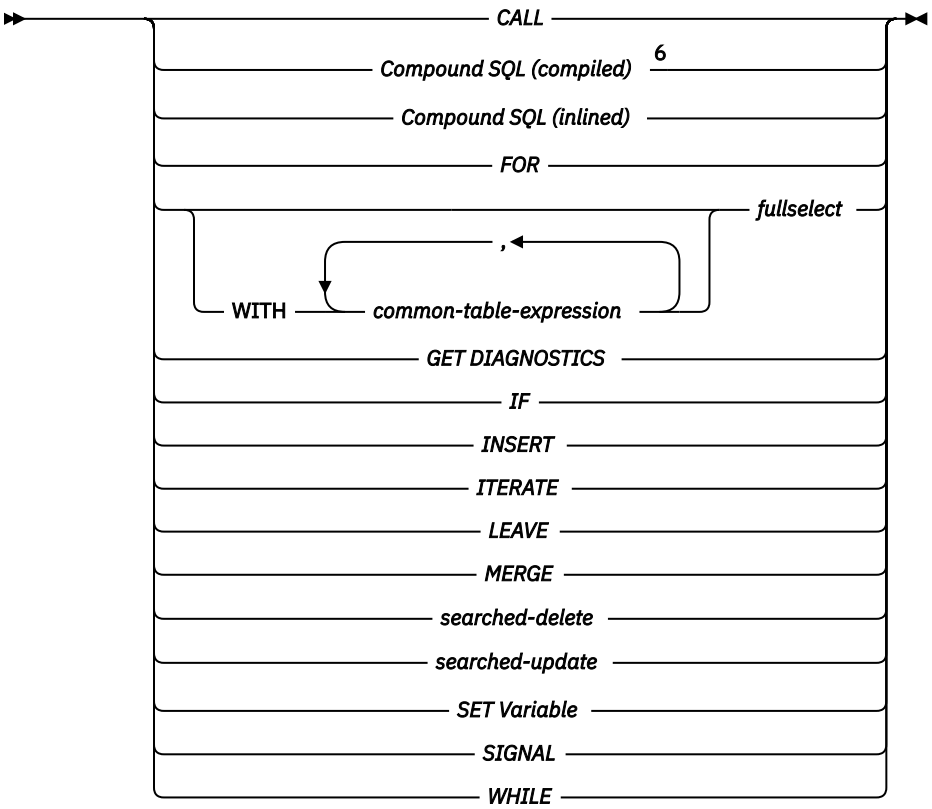
trigger-event



triggered-action



SQL-procedure-statement



Notes:

[1] OLD and NEW can only be specified once each.

[2] OLD TABLE and NEW TABLE can only be specified once each, and only for AFTER triggers or INSTEAD OF triggers.

[3] FOR EACH STATEMENT may not be specified for BEFORE triggers or INSTEAD OF triggers.

[4] A trigger event must not be specified more than once for the same operation. For example, INSERT OR DELETE is allowed, but INSERT OR INSERT is not allowed.

[5] WHEN condition may not be specified for INSTEAD OF triggers.

[6] A compound SQL (compiled) statement cannot be specified if the trigger definition includes a REFERENCING OLD TABLE clause or a REFERENCING NEW TABLE clause. A compound SQL (compiled) statement also cannot be specified for a trigger definition in a partitioned database environment.

Description

OR REPLACE

Specifies to replace the definition for the trigger if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog. This option is ignored if a definition for the trigger does not exist at the current server. This option can be specified only by the owner of the object.

trigger-name

Names the trigger. The name, including the implicit or explicit schema name, must not identify a trigger already described in the catalog (SQLSTATE 42710). If a two-part name is specified, the schema name cannot begin with "SYS" (SQLSTATE 42939).

NO CASCADE BEFORE

Specifies that the associated triggered action is to be applied before any changes caused by the actual update of the subject table are applied to the database. It also specifies that the triggered action of the trigger will not cause other triggers to be activated.

AFTER
Specifies that the associated triggered action is to be applied after the changes caused by the actual update of the subject table are applied to the database.

INSTEAD OF
Specifies that the associated triggered action replaces the action against the subject view. Only one INSTEAD OF trigger is allowed for each kind of operation on a given subject view (SQLSTATE 428FP).

trigger-event
Specifies that the triggered action associated with the trigger is to be executed whenever one of the events is applied to the subject table or subject view. Any combination of the events can be specified, but each event (INSERT, DELETE, and UPDATE) can only be specified once (SQLSTATE.42613). If multiple events are specified, the triggered action must be a compound SQL (compiled) statement (SQLSTATE 42601).

INSERT
Specifies that the triggered action associated with the trigger is to be executed whenever an INSERT operation is applied to the subject table or subject view.

DELETE
Specifies that the triggered action associated with the trigger is to be executed whenever a DELETE operation is applied to the subject table or subject view.

UPDATE
Specifies that the triggered action associated with the trigger is to be executed whenever an UPDATE operation is applied to the subject table or subject view, subject to the columns specified or implied.

If the optional *column-name* list is not specified, every column of the table or view is implied. Therefore, omission of the *column-name* list implies that the trigger will be activated by the update of any column of the table or view.

OF *column-name,...*
Each *column-name* specified must be a column of the base table (SQLSTATE 42703). If the trigger is a BEFORE trigger, the *column-name* specified cannot be a generated column other than the identity column (SQLSTATE 42989). No *column-name* can appear more than once in the *column-name* list (SQLSTATE 42711). The trigger will only be activated by the update of a column that is identified in the *column-name* list. This clause cannot be specified for an INSTEAD OF trigger (SQLSTATE 42613).

ON

table-name
Designates the subject table of the BEFORE trigger or AFTER trigger definition. The name must specify a base table or an alias that resolves to a base table (SQLSTATE 42704 or 42809). The name must not specify a catalog table (SQLSTATE 42832), a materialized query table (SQLSTATE 42997), a created temporary table, a declared temporary table (SQLSTATE 42995), or a nickname (SQLSTATE 42809).

view-name
Designates the subject view of the INSTEAD OF trigger definition. The name must specify an untyped view or an alias that resolves to an untyped view with no columns of type XML (SQLSTATE 42704 or 42809). The name must not specify a catalog view (SQLSTATE 42832). The name must not specify a view that is defined using WITH CHECK OPTION (a symmetric view), or a view on which a symmetric view has been defined, directly or indirectly (SQLSTATE 428FQ).

NOT SECURED or SECURED
Specifies whether the trigger is considered secure. The default is NOT SECURED.

NOT SECURED
Specifies the trigger is considered not secure.

SECURED
Specifies the trigger is considered secure. SECURED must be specified for a trigger whose subject table is a table on which row level or column level access control has been activated (SQLSTATE

428H8). Similarly, SECURED must be specified for a trigger that is created on a view and one or more of the underlying tables in that view definition has row level or column level access control activated (SQLSTATE 428H8).

REFERENCING

Specifies the correlation names for the *transition variables* and the table names for the *transition tables*. Correlation names identify a specific row in the set of rows affected by the triggering SQL operation. Table names identify the complete set of affected rows. Each row affected by the triggering SQL operation is available to the triggered action by qualifying columns with *correlation-names* specified as follows.

OLD AS *correlation-name*

Specifies a correlation name which identifies the row state before the triggering SQL operation.

NEW AS *correlation-name*

Specifies a correlation name which identifies the row state as modified by the triggering SQL operation and by any SET statement in a BEFORE trigger that has already executed.

The complete set of rows affected by the triggering SQL operation is available to the triggered action by using a temporary table name specified as follows.

OLD TABLE AS *identifier*

Specifies the name of a temporary table that identifies the values in the complete set of affected rows prior to the triggering SQL operation. If the trigger event is INSERT, the temporary table is empty.

NEW TABLE AS *identifier*

Specifies the name of a temporary table that identifies the state of the complete set of affected rows as modified by the triggering SQL operation and by any SET statement in a before trigger that has already been executed. If the trigger event is DELETE, the temporary table is empty.

The following rules apply to the REFERENCING clause:

- None of the OLD and NEW correlation names and the OLD TABLE and NEW TABLE names can be identical (SQLSTATE 42712).
- Only one OLD and one NEW *correlation-name* may be specified for a trigger (SQLSTATE 42613).
- Only one OLD TABLE and one NEW TABLE *identifier* may be specified for a trigger (SQLSTATE 42613).
- OLD TABLE or NEW TABLE identifiers cannot be defined in a BEFORE trigger (SQLSTATE 42898).
- A NEW transition variable can only be the target of an assignment in a BEFORE trigger. Otherwise, transition variables cannot be the target of an assignment (SQLSTATE 42703 or 42987).
- OLD or NEW correlation names cannot be defined in a FOR EACH STATEMENT trigger (SQLSTATE 42899).
- Transition tables cannot be modified (SQLSTATE 42807).
- The total of the references to the transition table columns and transition variables in the triggered-action cannot exceed the limit for the number of columns in a table or the sum of their lengths cannot exceed the maximum length of a row in a table (SQLSTATE 54040).
- The scope of each *correlation-name* and each *identifier* is the entire trigger definition.
- If the triggered-action includes a compound SQL (compiled) statement:
  - OLD TABLE or NEW TABLE identifiers cannot be defined.
  - If the operation is a DELETE operation, OLD *correlation-name* captures the value of the deleted row. If it is an UPDATE operation, it captures the value of the row before the UPDATE operation. For an insert operation, OLD *correlation-name* captures null values for each column of a row.
  - For an insert operation or an update operation, the value of NEW captures the new state of the row as provided by the original operation and as modified by any BEFORE trigger that has executed to this point. For a delete operation, NEW *correlation-name* captures null values for each column of a row. In a BEFORE DELETE trigger, any non-null values assigned to the new transition variables persist only within the trigger where the assignment occurred.

- If the triggered-action does not include a compound SQL (compiled) statement:
  - The OLD *correlation-name* and the OLD TABLE *identifier* can only be used if the trigger event is either a DELETE operation or an UPDATE operation (SQLSTATE 42898). If the operation is a DELETE operation, OLD *correlation-name* captures the value of the deleted row. If it is an UPDATE operation, it captures the value of the row before the UPDATE operation. The same applies to the OLD TABLE *identifier* and the set of affected rows.
  - The NEW *correlation-name* and the NEW TABLE *identifier* can only be used if the trigger event is either an INSERT operation or an UPDATE operation (SQLSTATE 42898). In both operations, the value of NEW captures the new state of the row as provided by the original operation and as modified by any BEFORE trigger that has executed to this point. The same applies to the NEW TABLE *identifier* and the set of affected rows.

FOR EACH ROW
Specifies that the triggered action is to be applied once for each row of the subject table or subject view that is affected by the triggering SQL operation.

FOR EACH STATEMENT
Specifies that the triggered action is to be applied only once for the whole statement. This type of trigger granularity cannot be specified for a BEFORE trigger or an INSTEAD OF trigger (SQLSTATE 42613). If specified, an UPDATE or DELETE trigger is activated, even if no rows are affected by the triggering UPDATE or DELETE statement.

triggered-action
Specifies the action to be performed when a trigger is activated. A triggered action is composed of an *SQL-procedure-statement* and by an optional condition for the execution of the *SQL-procedure-statement*.

Trigger event predicates can be used anywhere in the triggered action of a CREATE TRIGGER statement that uses a compound SQL (compiled) statement as the *SQL-procedure-statement*.

WHEN

(*search-condition*)
Specifies a condition that is true, false, or unknown. The *search-condition* provides a capability to determine whether or not a certain triggered action should be executed. The associated action is performed only if the specified search condition evaluates as true. If the WHEN clause is omitted, the associated *SQL-procedure-statement* is always performed.

The WHEN clause cannot be specified for INSTEAD OF triggers (SQLSTATE 42613).

A reference to a transition variable with an XML data type can be used only in a VALIDATED predicate.

label:
Specifies the label for an SQL procedure statement. The label must be unique within a list of SQL procedure statements, including any compound statements nested within the list. Note that compound statements that are not nested can use the same label. A list of SQL procedure statements is possible in a number of SQL control statements.

Only the FOR statement, WHILE statement, and the compound SQL statement can include a label.

SQL-procedure-statement
Specifies the SQL statement that is to be part of the triggered action. A searched update, searched delete, insert, or merge operation on nicknames inside compound SQL is not supported.

The triggered action of a BEFORE trigger on a column of type XML can invoke the XMLVALIDATE function through a SET statement, leave values of type XML unchanged, or assign them to NULL using a SET statement.

The *SQL-procedure-statement* must not contain a statement that is not supported (SQLSTATE 42987).

The *SQL-procedure-statement* cannot reference an undefined transition variable (SQLSTATE 42703), a federated object (SQLSTATE 42997), or a declared temporary table (SQLSTATE 42995). or the start and end columns of the BUSINESS_TIME period (SQLSTATE 42808).

The *SQL-procedure-statement* in a BEFORE trigger cannot:

- Contain any INSERT, DELETE, or UPDATE operations, nor invoke any routine defined with MODIFIES SQL DATA, if it is not a compound SQL (compiled).
- Contain any DELETE or UPDATE operations on the trigger subject table, nor invoke any routine containing such operations, if it is a compound SQL (compiled).
- Reference a materialized query table defined with REFRESH IMMEDIATE (SQLSTATE 42997)
- Reference a generated column other than the identity column in the NEW transition variable (SQLSTATE 42989).

Notes

- Adding a trigger to a table that already has rows in it will not cause any triggered actions to be activated. Thus, if the trigger is designed to enforce constraints on the data in the table, those constraints may not be satisfied by the existing rows.
- If the events for two triggers occur simultaneously (for example, if they have the same event, activation time, and subject tables), then the first trigger created is the first to execute. If the OR REPLACE option is used to replace a previously created trigger, the create time is changed and therefore could affect the order of trigger execution.
- If a column is added to the subject table after triggers have been defined, the following rules apply:
  - If the trigger is an UPDATE trigger that was specified without an explicit column list, then an update to the new column will cause the activation of the trigger.
  - The column will not be visible in the triggered action of any previously defined trigger.
  - The OLD TABLE and NEW TABLE transition tables will not contain this column. Thus, the result of performing a "SELECT *" on a transition table will not contain the added column.
- If a column is added to any table referenced in a triggered action, the new column will not be visible to the triggered action.
- If an object referenced in the trigger body does not exist or is marked invalid, or the definer temporarily doesn't have privileges to access the object, and if the database configuration parameter **auto_reval** is set to DEFERRED_FORCE, then the trigger will still be created successfully. The trigger will be marked invalid and will be revalidated the next time it is invoked.
- The result of a fullselect specified in a *SQL-procedure-statement* is not available inside or outside of the trigger.
- A procedure called within a triggered compound statement must not issue a COMMIT or a ROLLBACK statement (SQLSTATE 42985).
- A procedure that contains a reference to a nickname in a searched UPDATE statement, a searched DELETE statement, or an INSERT statement is not supported (SQLSTATE 25000).
- ***Table access restrictions:***: If a procedure is defined as READS SQL DATA or MODIFIES SQL DATA, no statement in the procedure can access a table that is being modified by the compound statement that invoked the procedure (SQLSTATE 57053). If the procedure is defined as MODIFIES SQL DATA, no statement in the procedure can modify a table that is being read or modified by the compound statement that invoked the procedure (SQLSTATE 57053).
- A BEFORE DELETE trigger defined on a table involved in a cycle of cascaded referential constraints should not include references to the table on which it is defined or any other table modified by cascading during the evaluation of the cycle of referential integrity constraints. The results of such a trigger are data dependent and therefore may not produce consistent results.

  In its simplest form, this means that a BEFORE DELETE trigger on a table with a self-referencing referential constraint and a delete rule of CASCADE should not include any references to the table in the *triggered-action*.

- The creation of a trigger causes certain packages to be marked invalid:
  - If an UPDATE trigger without an explicit column list is created, then packages with an update usage on the target table or view are invalidated.
  - If an UPDATE trigger with a column list is created, then packages with update usage on the target table are only invalidated if the package also has an update usage on at least one column in the *column-name* list of the CREATE TRIGGER statement.
  - If an INSERT trigger is created, packages that have an insert usage on the target table or view are invalidated.
  - If a delete trigger is created, packages that have a delete usage on the target table or view are invalidated.
- A package remains invalid until the application program is explicitly bound or rebound, or it is executed and the database manager automatically rebinds it.
- ***Inoperative triggers***: An *inoperative trigger* is a trigger that is no longer available and is therefore never activated. A trigger becomes inoperative if:
  - a privilege that the creator of the trigger is required to have for the trigger to execute is revoked
  - an object such as a table, view or alias, upon which the triggered action is dependent, is dropped
  - a view, upon which the triggered action is dependent, becomes inoperative
  - an alias that is the subject table of the trigger is dropped.

  In practical terms, an inoperative trigger is one in which a trigger definition has been dropped as a result of cascading rules for DROP or REVOKE statements. For example, when a view is dropped, any trigger with an *SQL-procedure-statement* that contains a reference to that view is made inoperative.

  When a trigger is made inoperative, all packages with statements performing operations that were activating the trigger will be marked invalid. When the package is rebound (explicitly or implicitly) the inoperative trigger is completely ignored. Similarly, applications with dynamic SQL statements performing operations that were activating the trigger will also completely ignore any inoperative triggers.

  The trigger name can still be specified in the DROP TRIGGER and COMMENT ON TRIGGER statements.

  An inoperative trigger may be re-created by issuing a CREATE TRIGGER statement using the definition text of the inoperative trigger. This trigger definition text is stored in the TEXT column of the SYSCAT.TRIGGERS catalog view. Note that there is no need to explicitly drop the inoperative trigger in order to re-create it. Issuing a CREATE TRIGGER statement with the same *trigger-name* as an inoperative trigger will cause that inoperative trigger to be replaced with a warning (SQLSTATE 01595).

  Inoperative triggers are indicated by an X in the VALID column of the SYSCAT.TRIGGERS catalog view.
- ***Errors executing triggers***: Errors that occur during the execution of triggered SQL statements are returned using SQLSTATE 09000 unless the error is considered severe. If the error is severe, the severe error SQLSTATE is returned. The SQLERRMC field of the SQLCA for non-severe error will include the trigger name, SQLCODE, SQLSTATE and as many tokens as will fit from the tokens of the failure.

  The *SQL-procedure-statement* could include a SIGNAL SQLSTATE statement or a RAISE_ERROR function. In both these cases, the SQLSTATE returned is the one specified in the SIGNAL SQLSTATE statement or the RAISE_ERROR condition.
- Creating a trigger with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- ***DB2SECURITYLABEL column***: A DB2SECURITYLABEL column can be referenced in the trigger body of a BEFORE TRIGGER but it cannot be changed in the body of a BEFORE trigger (SQLSTATE 42989).
- ***BUSINESS_TIME period columns***: The start and end columns of a BUSINESS_TIME period cannot be changed in the body of BEFORE UPDATE trigger (SQLSTATE 42808).
- ***Read-only views***: The addition of an INSTEAD OF trigger for a view affects the read only characteristic of the view. If a read-only view has a dependency relationship with an INSTEAD OF trigger, the type of

operation that is defined for the INSTEAD OF trigger defines whether the view is deletable, insertable, or updatable.

- ***Transition variable values and INSTEAD OF triggers***: The initial values for new transition variables or new transition table columns that are visible in an INSTEAD OF INSERT trigger are set as follows:
  - If a value is explicitly specified for a column in the insert operation, the corresponding new transition variable is that explicitly specified value.
  - If a value is not explicitly specified for a column in the insert operation or the DEFAULT clause is specified, the corresponding new transition variable is:
    - the default value of the underlying table column if the view column is updatable (without the INSTEAD OF trigger)
    - otherwise, the null value

  The initial values for new transition variables that are visible in an INSTEAD OF UPDATE trigger are set as follows:
  - If a value is explicitly specified for a column in the update operation, the corresponding new transition variable is that explicitly specified value.
  - If the DEFAULT clause is explicitly specified for a column in the update operation, the corresponding new transition variable is:
    - the default value of the underlying table column if the view column is updatable (without the INSTEAD OF trigger)
    - otherwise, the null value
  - Otherwise, the corresponding new transition variable is the existing value of the column in the row.
- ***Triggers and typed tables***: A BEFORE or AFTER trigger can be attached to a typed table at any level of a table hierarchy. If an SQL statement activates multiple triggers, the triggers will be executed in their creation order, even if they are attached to different tables in the typed table hierarchy.

  When a trigger is activated, its transition variables (OLD, NEW, OLD TABLE and NEW TABLE) may contain rows of subtables. However, they will contain only columns defined on the table to which they are attached.

  Effects of INSERT, UPDATE, and DELETE statements:
  - Row triggers: When an SQL statement is used to INSERT, UPDATE, or DELETE a table row, it activates row-triggers attached to the most specific table containing the row, and all supertables of that table. This rule is always true, regardless of how the SQL statement accesses the table. For example, when issuing an UPDATE EMP command, some of the updated rows may be in the subtable MGR. For EMP rows, the row-triggers attached to EMP and its supertables are activated. For MGR rows, the row-triggers attached to MGR and its supertables are activated.
  - Statement triggers: An INSERT, UPDATE, or DELETE statement activates statement-triggers attached to tables (and their supertables) that could be affected by the statement. This rule is always true, regardless of whether any actual rows in these tables were affected. For example, on an INSERT INTO EMP command, statement-triggers for EMP and its supertables are activated. As another example, on either an UPDATE EMP or DELETE EMP command, statement triggers for EMP and its supertables and subtables are activated, even if no subtable rows were updated or deleted. Likewise, a UPDATE ONLY (EMP) or DELETE ONLY (EMP) command will activate statement-triggers for EMP and its supertables, but not statement-triggers for subtables.

  Effects of DROP TABLE statements: A DROP TABLE statement does not activate any triggers that are attached to the table being dropped. However, if the dropped table is a subtable, all the rows of the dropped table are considered to be deleted from its supertables. Therefore, for a table T:
  - Row triggers: DROP TABLE T activates row-type delete-triggers that are attached to all supertables of T, for each row of T.
  - Statement triggers: DROP TABLE T activates statement-type delete-triggers that are attached to all supertables of T, regardless of whether T contains any rows.

Actions on Views: To predict what triggers are activated by an action on a view, use the view definition to translate that action into an action on base tables. For example:

1. An SQL statement performs UPDATE V1, where V1 is a typed view with a subview V2. Suppose V1 has underlying table T1, and V2 has underlying table T2. The statement could potentially affect rows in T1, T2, and their subtables, so statement triggers are activated for T1 and T2 and all their subtables and supertables.
2. An SQL statement performs UPDATE V1, where V1 is a typed view with a subview V2. Suppose V1 is defined as SELECT ... FROM ONLY(T1) and V2 is defined as SELECT ... FROM ONLY(T2). Since the statement cannot affect rows in subtables of T1 and T2, statement triggers are activated for T1 and T2 and their supertables, but not their subtables.
3. An SQL statement performs UPDATE ONLY(V1), where V1 is a typed view defined as SELECT ... FROM T1. The statement can potentially affect T1 and its subtables. Therefore, statement triggers are activated for T1 and all its subtables and supertables.
4. An SQL statement performs UPDATE ONLY(V1), where V1 is a typed view defined as SELECT ... FROM ONLY(T1). In this case, T1 is the only table that can be affected by the statement, even if V1 has subviews and T1 has subtables. Therefore, statement triggers are activated only for T1 and its supertables.

- ***MERGE statement and triggers***: The MERGE statement can execute update, delete, and insert operations. The applicable UPDATE, DELETE, or INSERT triggers are activated for the MERGE statement when an update, delete, or insert operation is executed.
- ***Obfuscation***: The CREATE TRIGGER statement can be submitted in obfuscated form. In an obfuscated statement, only the trigger name is readable. The rest of the statement is encoded in such a way that is not readable but can be decoded by the database server. Obfuscated statements can be produced by calling the DBMS_DDL.WRAP function.
- ***Creating a trigger with the SECURED option***: Normally users with SECADM authority do not have privileges to create database objects such as triggers or user-defined functions. Typically, they will examine the data accessed by a trigger, ensure it is secure, then grant the CREATE_SECURE_OBJECT authority to someone who has the required privileges to create the secure trigger. After the trigger is created, they will revoke the CREATE_SECURE_OBJECT authority from the trigger owner.

  The trigger is considered secure. The database manager treats the SECURED attribute as an assertion that declares the user has established an audit procedure for all activities in the trigger body. If a secure trigger references user-defined functions, the database manager assumes those functions are secure without validation. If those functions can access sensitive data, the user with SECADM authority needs to ensure those functions are allowed to access those data and that all subsequent ALTER FUNCTION statements or changes to external packages are being reviewed by this audit process.

  A trigger must be secure if its subject table has row level or column level access control activated. Similarly, a trigger must be secure if its subject table is a view and one or more of the underlying tables in the view definition has row level or column level access control activated.
- ***Creating a trigger with the NOT SECURED option***: The CREATE TRIGGER statement returns an error if the trigger's subject table has row level or column level access control activated. Similarly, the CREATE TRIGGER statement fails if the trigger is defined on a view and one or more of the underlying tables in that view definition has row level or column level access control activated.
- ***Row and column access control that is not enforced for transition variables and transition tables***: Triggers are used for database integrity, and as such a balance between security and database integrity is needed. If row level or column level access control is activated on the subject table or an underlying table of the subject view, row permissions and column masks are not applied to the initial values of transition variables and transition tables. Row level and column level access control that is enforced for the subject table or an underlying table of the subject view is also ignored for transition variables and transition tables that are referenced in the trigger body or are passed as arguments to user-defined functions invoked in the trigger body. To ensure there is no security concern for SQL statements in the trigger action to access sensitive data in transition variables and transition tables, the trigger must be created with the SECURED option. If a trigger is not secure, the CREATE TRIGGER statement returns an error.

- ***Considerations for implicitly hidden columns***: A transition variable exists for any column defined as implicitly hidden. In the body of a trigger, a transition variable that corresponds to an implicitly hidden column can be referenced.
- ***Rebinding dependent packages***: Every compiled trigger has a dependent package. The package can be rebound at any time by using the REBIND_ROUTINE_PACKAGE procedure. Explicitly rebinding the dependent package does not revalidate an invalid trigger. Revalidate an invalid trigger by using automatic revalidation or explicitly by using the ADMIN_REVALIDATE_DB_OBJECTS procedure. Trigger revalidation automatically rebinds the dependent package.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - OLD_TABLE can be specified in place of OLD TABLE, and NEW_TABLE can be specified in place of NEW TABLE
  - MODE DB2SQL can be specified following FOR EACH ROW or FOR EACH STATEMENT

Examples

- *Example 1:* Create two triggers that will result in the automatic tracking of the number of employees a company manages. The triggers will interact with the following tables:
  - EMPLOYEE table with these columns: ID, NAME, ADDRESS, and POSITION.
  - COMPANY_STATS table with these columns: NBEMP, NBPRODUCT, and REVENUE.

  The first trigger increments the number of employees each time a new person is hired; that is, each time a new row is inserted into the EMPLOYEE table:

```
CREATE TRIGGER NEW_HIRED
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

  The second trigger decrements the number of employees each time an employee leaves the company; that is, each time a row is deleted from the table EMPLOYEE:

```
CREATE TRIGGER FORMER_EMP
  AFTER DELETE ON EMPLOYEE
  FOR EACH ROW
  UPDATE COMPANY_STATS SET NBEMP = NBEMP - 1
```

- *Example 2:* Create a trigger that ensures that whenever a parts record is updated, the following check and (if necessary) action is taken:
  - If the on-hand quantity is less than 10% of the maximum stocked quantity, then issue a shipping request ordering the number of items for the affected part to be equal to the maximum stocked quantity minus the on-hand quantity.

  The trigger will interact with the PARTS table with these columns: PARTNO, DESCRIPTION, ON_HAND, MAX_STOCKED, and PRICE.

  ISSUE_SHIP_REQUEST is a user-defined function that sends an order form for additional parts to the appropriate company.

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N
  FOR EACH ROW
  WHEN (N.ON_HAND < 0.10 * N.MAX_STOCKED)
  BEGIN ATOMIC
  VALUES(ISSUE_SHIP_REQUEST(N.MAX_STOCKED - N.ON_HAND, N.PARTNO));
  END
```

- *Example 3:* Repeat the scenario in Example 2 except use a fullselect instead of a VALUES statement to invoke the user-defined function. This example also shows how to define the trigger as a statement

trigger instead of a row trigger. For each row in the transition table that evaluates to true for the WHERE clause, a shipping request is issued for the part.

```
CREATE TRIGGER REORDER
   AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
   REFERENCING NEW TABLE AS NTABLE
   FOR EACH STATEMENT
      BEGIN ATOMIC
         SELECT ISSUE_SHIP_REQUEST(MAX_STOCKED - ON_HAND, PARTNO)
            FROM NTABLE
         WHERE (ON_HAND < 0.10 * MAX_STOCKED);
   END
```

- *Example 4:* Create a trigger that will cause an error when an update occurs that would result in a salary increase greater than ten percent of the current salary.

```
CREATE TRIGGER RAISE_LIMIT
  AFTER UPDATE OF SALARY ON EMPLOYEE
  REFERENCING NEW AS N OLD AS O
  FOR EACH ROW
  WHEN (N.SALARY > 1.1 * O.SALARY)
        SIGNAL SQLSTATE '75000' SET MESSAGE_TEXT='Salary increase>10%'
```

- *Example 5:* Consider an application which records and tracks changes to stock prices. The database contains two tables, CURRENTQUOTE and QUOTEHISTORY.

```
Tables: CURRENTQUOTE (SYMBOL, QUOTE, STATUS)
        QUOTEHISTORY (SYMBOL, QUOTE, QUOTE_TIMESTAMP)
```

When the QUOTE column of CURRENTQUOTE is updated, the new quote should be copied, with a timestamp, to the QUOTEHISTORY table. Also, the STATUS column of CURRENTQUOTE should be updated to reflect whether the stock is:

1. rising in value;
2. at a new high for the year;
3. dropping in value;
4. at a new low for the year;
5. steady in value.

CREATE TRIGGER statements that accomplish this are as follows.

– Trigger Definition to set the status:

```
   CREATE TRIGGER STOCK_STATUS
     NO CASCADE BEFORE UPDATE OF QUOTE ON CURRENTQUOTE
     REFERENCING NEW AS NEWQUOTE OLD AS OLDQUOTE
     FOR EACH ROW
     BEGIN ATOMIC
        SET NEWQUOTE.STATUS =
          CASE
             WHEN NEWQUOTE.QUOTE >
                   (SELECT MAX(QUOTE) FROM QUOTEHISTORY
                   WHERE SYMBOL = NEWQUOTE.SYMBOL
                   AND YEAR(QUOTE_TIMESTAMP) = YEAR(CURRENT DATE) )
                THEN 'High'
             WHEN NEWQUOTE.QUOTE <
(SELECT MIN(QUOTE) FROM QUOTEHISTORY
                   WHERE SYMBOL = NEWQUOTE.SYMBOL
                   AND YEAR(QUOTE_TIMESTAMP) = YEAR(CURRENT DATE) )
                THEN 'Low'
             WHEN NEWQUOTE.QUOTE > OLDQUOTE.QUOTE
                THEN 'Rising'
             WHEN NEWQUOTE.QUOTE < OLDQUOTE.QUOTE
                THEN 'Dropping'
             WHEN NEWQUOTE.QUOTE = OLDQUOTE.QUOTE
                THEN 'Steady'
          END;
     END
```

– Trigger Definition to record change in QUOTEHISTORY table:

```
CREATE TRIGGER RECORD_HISTORY
  AFTER UPDATE OF QUOTE ON CURRENTQUOTE
  REFERENCING NEW AS NEWQUOTE
  FOR EACH ROW
  BEGIN ATOMIC
    INSERT INTO QUOTEHISTORY
      VALUES (NEWQUOTE.SYMBOL, NEWQUOTE.QUOTE, CURRENT TIMESTAMP);
  END
```

- *Example 6:*Create a trigger that overrides any changes to the location field in the employee record in the org table. This trigger would be useful if new employee records acquired when a smaller company was purchased are processed and the target location allocated to the employee is "Toronto" and the new target location is "Los Angeles". The before trigger will ensure that regardless what value the application allocates for this field, that the final resultant value is "Los Angeles".

```
CREATE TRIGGER LOCATION_TRIGGER
  NO CASCADE
   BEFORE UPDATE ON ORG
   REFERENCING
     OLD AS PRE
     NEW AS POST
   FOR EACH ROW
   WHEN (POST.LOCATION = 'Toronto')
      SET POST.LOCATION = 'Los Angeles';
   END
```

- *Example 7:* Create a BEFORE trigger that automatically validates XML documents containing new product descriptions before they are inserted into the PRODUCT table of the SAMPLE database:

```
CREATE TRIGGER NEWPROD NO CASCADE BEFORE INSERT ON PRODUCT
   REFERENCING NEW AS N
   FOR EACH ROW
   BEGIN ATOMIC
      SET (N.DESCRIPTION) = XMLVALIDATE(N.DESCRIPTION
         ACCORDING TO XMLSCHEMA ID product);
   END
```

- *Example 8:* Create a multiple-event trigger that tracks of the number and salary of employees a company manages. The triggers will interact with the following columns and tables:
  - ID, NAME, ADDRESS, SALARY, and POSITION columns in the EMPLOYEE table
  - NBEMP, NBPRODUCT, and REVENUE columns in the COMPANY_STATS table

  The trigger increments the number of employees each time a new employee is hired; decrements the number of employees each time an employee leaves the company, and raises an error when an update occurs that would result in a salary increase greater than ten percent of the current salary:

```
CREATE OR REPLACE TRIGGER HIRED
   AFTER INSERT OR DELETE OR UPDATE OF SALARY ON EMPLOYEE
   REFERENCING NEW AS N OLD AS O FOR EACH ROW
   BEGIN
      IF INSERTING THEN UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
      ELSEIF DELETING THEN UPDATE COMPANY_STATS SET NBEMP = NBEMP - 1;
      ELSEIF (UPDATING AND (N.SALARY > 1.1 * O.SALARY))
         THEN SIGNAL SQLSTATE '75000' SET MESSAGE_TEXT='Salary increase>10%';
      END IF;
   END;
```

- *Example 9:* Create a trigger that ensures that the following check and (if necessary) action is taken, before a parts record is updated:
  - If the on-hand quantity is less than 10% of the maximum stocked quantity, then place a new order record into the ORDER table and issue a shipping request ordering the number of items for the affected part to be equal to the maximum stocked quantity minus the on-hand quantity.

  The trigger interacts with the following columns and tables:
  - PARTNO, DESCRIPTION, ON_HAND, MAX_STOCKED, and PRICE columns in the PARTS table
  - PARTNO and PRICE columns in the ORDER table

ISSUE_SHIP_REQUEST is a user-defined SQL data modification stored procedure that sends an order form for additional parts to the supply company, and deletes the corresponding row from the ORDER table after the order form is confirmed by the supply company.

```
CREATE TRIGGER REORDER
   BEFORE UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
   REFERENCING NEW AS N
   FOR EACH ROW
   WHEN (N.ON_HAND < 0.10 * N.MAX_STOCKED)
   BEGIN
      INSERT INTO ORDERS VALUES (N.MAX_STOCKED - N.ON_HAND, N.PARTNO);
      CALL ISSUE_SHIP_REQUEST(N.MAX_STOCKED - N.ON_HAND, N.PARTNO);
   END;
```

CREATE TRUSTED CONTEXT

The CREATE TRUSTED CONTEXT statement defines a trusted context at the current server.

Important: The DATA_ENCRYPT authentication type is deprecated and might be removed in a future release. To encrypt data in-transit between clients and Db2 databases, we recommend that you use the Db2 database system support of Transport Layer Security (TLS). For more information, see Encryption of data in transit

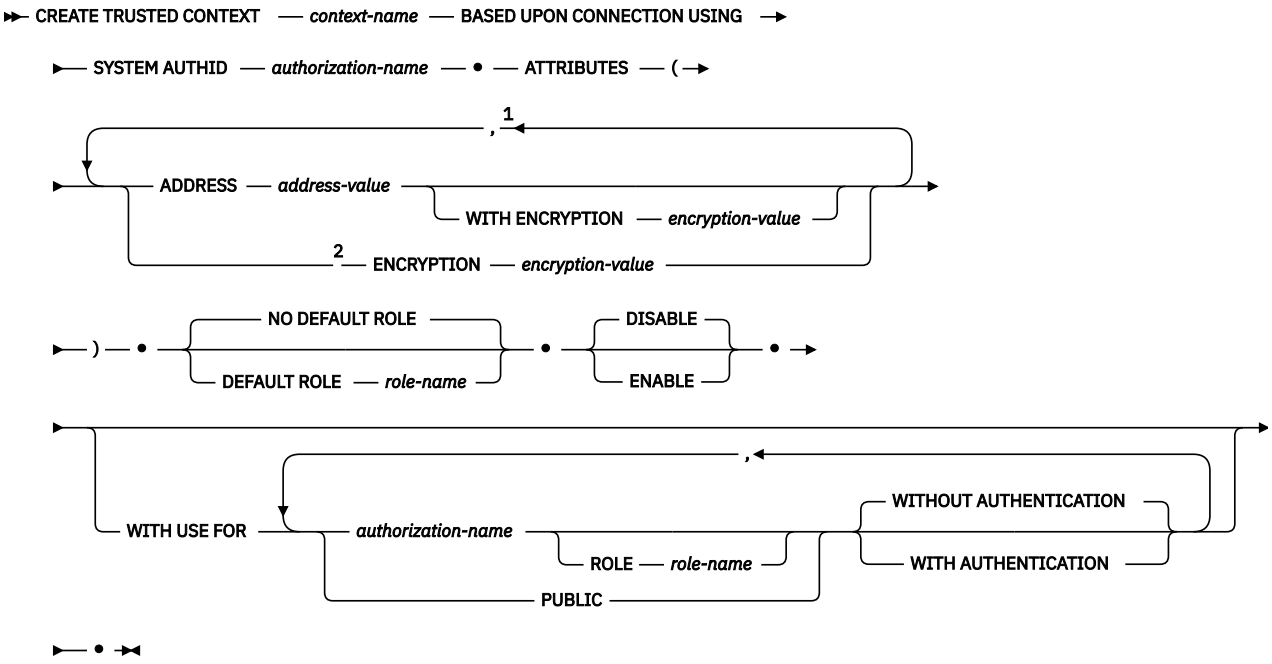
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include SECADM authority.

Syntax



Notes:

[1] Each combination of an attribute name and its corresponding value, as a pair, must be unique (SQLSTATE 4274D).

[2] ENCRYPTION cannot be specified more than once (SQLSTATE 42614); however, WITH ENCRYPTION can be specified for each ADDRESS that is specified.

Description

context-name
Names the trusted context. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The name must not identify a trusted context that already exists at the current server (SQLSTATE 42710). The name must not begin with the characters 'SYS' (SQLSTATE 42939).

BASED UPON CONNECTION USING SYSTEM AUTHID *authorization-name*
Specifies that the context is a connection established by system authorization ID *authorization-name*, which must not be associated with an existing trusted context (SQLSTATE 428GL). It cannot be the authorization ID of the statement (SQLSTATE 42502).

ATTRIBUTES (...)
Specifies a list of one or more connection trust attributes upon which the trusted context is defined.

ADDRESS *address-value*
Specifies the actual communication address used by the client to communicate with the database server. The only protocol supported is TCP/IP. The ADDRESS attribute can be specified multiple times, but each *address-value* pair must be unique for the set of attributes (SQLSTATE 4274D).

When establishing a trusted connection, if multiple values are defined for the ADDRESS attribute of a trusted context, a candidate connection is considered to match this attribute if the address used by the connection matches any of the defined values for the ADDRESS attribute of the trusted context.

address-value
Specifies a string constant that contains the value to be associated with the ADDRESS trust attribute. The *address-value* must be an IPv4 address, an IPv6 address, or a secure domain name.

- An IPv4 address must not contain leading spaces and is represented as a dotted decimal address. An example of an IPv4 address is 9.112.46.111. The value 'localhost' or its equivalent representation '127.0.0.1' will not result in a match; the real IPv4 address of the host must be specified instead.
- An IPv6 address must not contain leading spaces and is represented as a colon hexadecimal address. An example of an IPv6 address is 2001:0DB8:0000:0000:0008:0800:200C:417A. IPv4-mapped IPv6 addresses (for example, ::ffff:192.0.2.128) will not result in a match. Similarly, 'localhost' or its IPv6 short representation '::1' will not result in a match.
- A domain name is converted to an IP address by the domain name server where a resulting IPv4 or IPv6 address is determined. An example of a domain name is corona.torolab.ibm.com. When a domain name is converted to an IP address, the result of this conversion could be a set of one or more IP addresses. In this case, an incoming connection is said to match the ADDRESS attribute of a trusted context object if the IP address from which the connection originates matches any of the IP addresses to which the domain name was converted. When creating a trusted context object, it is advantageous to provide domain name values for the ADDRESS attribute instead of static IP addresses, particularly in Dynamic Host Configuration Protocol (DHCP) environments. With DHCP, a device can have a different IP address each time it connects to the network. So, if a static IP address is provided for the ADDRESS attribute of a trusted context object, some device might acquire a trusted connection unintentionally. Providing domain names for the ADDRESS attribute of a trusted context object avoids this problem in DHCP environments.

WITH ENCRYPTION *encryption-value*
Specifies the minimum level of encryption of the data stream or network encryption for this specific *address-value*. This *encryption-value* overrides the global ENCRYPTION attribute setting for this specific *address-value*.

encryption-value
Specifies a string constant that contains the value to be associated with the ENCRYPTION trust attribute for this specific *address-value*. The *encryption-value* must be one of the following values (SQLSTATE 42615):

- NONE, no specific level of encryption is required
- LOW, a minimum of light encryption is required; the authentication type on the database manager must be DATA_ENCRYPT if an incoming connection is to match the encryption setting for this specific address
- HIGH, Secure Sockets Layer (SSL) encryption, or equivalent, must be used for data communication between the database client and the database server if an incoming connection is to match the encryption setting for this specific address

ENCRYPTION *encryption-value*
Specifies the minimum level of encryption of the data stream or network encryption. The default is NONE.

encryption-value
Specifies a string constant that contains the value to be associated with the ENCRYPTION trust attribute for this specific *address-value*. The *encryption-value* must be one of the following values (SQLSTATE 42615):

- NONE, no specific level of encryption is required for an incoming connection to match the ENCRYPTION attribute of this trusted context object
- LOW, a minimum of light encryption is required; the authentication type on the database manager must be DATA_ENCRYPT if an incoming connection is to match the ENCRYPTION attribute of this trusted context object
- HIGH, Secure Sockets Layer (SSL) encryption, or equivalent, must be used for data communication between the database client and the database server if an incoming connection is to match the ENCRYPTION attribute of this trusted context object

The following table summarizes when a trusted context can be used, depending on the encryption used by the existing connection. If the trusted context cannot be used for the connection, a warning is returned (SQLSTATE 01679) and the SQLWARN8 field of the SQLCA is set to 'Y', indicating that the connection is a regular (non-trusted) connection.

Table 145. Encryption and trusted contexts

Encryption used by existing connection	**ENCRYPTION value for trusted context**	**Can the trusted context be used for the connection?**
No encryption	'NONE'	Yes
No encryption	'LOW'	No
No encryption	'HIGH'	No
Low encryption (DATA_ENCRYPT)	'NONE'	Yes
Low encryption (DATA_ENCRYPT)	'LOW'	Yes
Low encryption (DATA_ENCRYPT)	'HIGH'	No
High encryption (SSL)	'NONE'	Yes
High encryption (SSL)	'LOW'	Yes
High encryption (SSL)	'HIGH'	Yes

NO DEFAULT ROLE or DEFAULT ROLE *role-name*

Specifies whether or not a default role is associated with a trusted connection that is based on this trusted context. The default is NO DEFAULT ROLE.

NO DEFAULT ROLE

Specifies that the trusted context does not have a default role.

DEFAULT ROLE *role-name*

Specifies that *role-name* is the default role for the trusted context. The *role-name* must identify a role that exists at the current server (SQLSTATE 42704). This role is used with the user in a trusted connection, based on this trusted context, when the user does not have a user-specific role defined as part of the definition of the trusted context.

DISABLE or ENABLE

Specifies whether the trusted context is created in the enabled or disabled state. The default is DISABLE.

DISABLE

Specifies that the trusted context is created in the disabled state. A trusted context that is disabled is not considered when a trusted connection is established.

ENABLE

Specifies that the trusted context is created in the enabled state.

WITH USE FOR

Specifies who can use a trusted connection that is based on this trusted context.

authorization-name

Specifies that the trusted connection can be used by the specified *authorization-name*. The *authorization-name* must not be specified more than once in the WITH USE FOR clause (SQLSTATE 428GM). It must also not be the authorization ID of the statement (SQLSTATE 42502). If the definition of a trusted context allows access by both PUBLIC and a list of users, the specifications for a user override the specifications for PUBLIC. For example, assume that a trusted context is defined that allows access by both PUBLIC WITH AUTHENTICATION and JOE WITHOUT AUTHENTICATION. If the trusted context is used by JOE, authentication is not required. However, if the trusted context is used by GEORGE, authentication is required.

ROLE *role-name*

Specifies that *role-name* is the role to be used for the user when a trusted connection is using the trusted context. The *role-name* must identify a role that exists at the current server (SQLSTATE 42704). The role explicitly specified for the user overrides any default role associated with the trusted context.

PUBLIC

Specifies that a trusted connection that is based on this trusted context can be used by any user. PUBLIC must not be specified more than once (SQLSTATE 428GM). All users using such a trusted connection make use of the privileges associated with the default role for the associated trusted context. If a default role is not defined for the trusted context, there is no role associated with the users that use a trusted connection based on this trusted context.

WITHOUT AUTHENTICATION or WITH AUTHENTICATION

Specifies whether or not switching the user on a trusted connection requires authentication of the user. The default is WITHOUT AUTHENTICATION.

WITHOUT AUTHENTICATION

Specifies that switching the current user on a trusted connection to this user does not require authentication.

WITH AUTHENTICATION

Specifies that switching the current user on a trusted connection to this user requires authentication.

Rules

- A trusted context-exclusive SQL statement must be followed by a COMMIT or a ROLLBACK statement (SQLSTATE 5U021). Trusted context-exclusive SQL statements are:
  - CREATE TRUSTED CONTEXT, ALTER TRUSTED CONTEXT, or DROP (TRUSTED CONTEXT)
- A trusted context-exclusive SQL statement cannot be issued within a global transaction; for example, an XA transaction or a global transaction that is initiated as part of two-phase commit for federated transactions (SQLSTATE 51041).

Notes

- When providing an IP address as part of a trusted context definition, the address must be in the format that is in effect for the network. For example, providing an address in an IPv6 format when the network is IPv4 will not result in a match. In a mixed environment, it is advantageous to specify both the IPv4 and the IPv6 representations of the address, or better yet, to specify a secure domain name (for example, corona.torolab.ibm.com), which hides the address format details.
- ***Specifying a role in the definition of a trusted context:*** The definition of a trusted context can designate a role for a specific authorization ID, and a default role to be used for authorization IDs for which a specific role has not been specified in the definition of the trusted context. This role can be used with a trusted connection based on the trusted context, but it does not make the role available outside of a trusted connection based on the trusted context.
- When issuing a data manipulation language (DML) SQL statement using a trusted connection, the privileges held by a context-assigned role in effect for the authorization ID within the definition of the associated trusted context are considered in addition to other privileges directly held by the authorization ID of the statement, or indirectly by other roles held by the authorization ID of the statement.
- The privileges held by a context-assigned role in effect for the authorization ID within the definition of the associated trusted context are not considered for data definition language (DDL) SQL statements. For example, to create an object, the authorization ID of the statement must be able to do so without including the privileges held by the context-assigned role.
- When installing a new application that authenticates to the database server using the same credentials as an existing application on the same machine, and which takes advantage of a trusted context, the new application might also take advantage of the same trusted context object (inheriting the trusted context role, for example). This might not be the security administrator's intention. The security administrator might want to turn on the database audit facility to find out what applications are taking advantage of trusted context objects.
- Only one uncommitted trusted context-exclusive SQL statement is allowed at a time across all database partitions. If an uncommitted trusted context-exclusive SQL statement is executing, subsequent trusted context-exclusive SQL statements will wait until the current trusted context-exclusive SQL statement commits or rolls back.
- Changes are written to the system catalog, but do not take effect until they are committed, even for the connection that issues the statement.

Examples

- *Example 1:* Create a trusted context such that the current user on a trusted connection based on this trusted context can be switched to two different user IDs. When the current user of the connection is switched to user ID JOE, authentication is not required. However, authentication is required when the current user of the connection is switched to user ID BOB. Note that the trusted context has a default role called *context-role*. This implies that users working within the confines of this trusted context inherit the privileges associated with role *context-role*.

```
CREATE TRUSTED CONTEXT APPSERVER
  BASED UPON CONNECTION USING SYSTEM AUTHID WRJAIBI
  DEFAULT ROLE CONTEXT_ROLE
  ENABLE
  ATTRIBUTES (ADDRESS '9.26.113.204')
```

```
  WITH USE FOR JOE WITHOUT AUTHENTICATION
    BOB WITH AUTHENTICATION
```

- *Example 2:* Create a trusted context such that the current user of a trusted connection based on this trusted context can be switched to any user ID without authentication.

```
CREATE TRUSTED CONTEXT SECUREROLE
  BASED UPON CONNECTION USING SYSTEM AUTHID PBIRD
  ENABLE
  ATTRIBUTES (ADDRESS '9.26.113.204')
  WITH USE FOR PUBLIC WITHOUT AUTHENTICATION
```

- *Example 3:* Create a trusted context such that the current user of a trusted connection based on this trusted context can be switched to any user ID without authentication. The difference between this trusted context and the trusted context created in example 2, is that this trusted context has an additional attribute called ENCRYPTION. The ENCRYPTION attribute setting for trusted context SECUREROLEENCRYPT states that the encryption setting used by a connection must be at least "low encryption" (see Table 145 on page 1476) to match this trusted context attribute.

```
CREATE TRUSTED CONTEXT SECUREROLEENCRYPT
  BASED UPON CONNECTION USING SYSTEM AUTHID SHARPER
  ENABLE
  ATTRIBUTES (ADDRESS '9.26.113.204'
    ENCRYPTION 'LOW')
  WITH USE FOR PUBLIC WITHOUT AUTHENTICATION
```

- *Example 4:* Create a trusted context, such that connections made by user WRJAIBI from addresses 9.26.146.201 and 9.26.146.203 are trusted when no encryption is used, but a connection made by user WRJAIBI from address 9.26.146.202 requires a LOW level of encryption to be trusted.

```
CREATE TRUSTED CONTEXT WALIDLOCSENSITIVE
  BASED UPON CONNECTION USING SYSTEM AUTHID WRJAIBI
  ENABLE
  ATTRIBUTES (ADDRESS '9.26.146.201',
    ADDRESS '9.26.146.202' WITH ENCRYPTION 'LOW',
    ADDRESS '9.26.146.203'
    ENCRYPTION 'NONE')
```

CREATE TYPE

The CREATE TYPE statement defines a user-defined data type at the current server.

Five different kinds of user-defined data types can be created using this statement. Each of these types is described separately.

- Array. A user-defined data type that is an ordinary array or an associative array. The elements of an array type are based on one of the built-in data types or a user-defined type other than a cursor type or structured type.
- Cursor. A user-defined data type that is a cursor type.
- Distinct. A user-defined data type that is sourced on one of the built-in data types and can be defined to use strong type rules or weak type rules.. Functions that cast between the user-defined distinct type and the source built-in data type are generated when a strongly typed distinct type is created. Optionally, support for comparison operations to use with the strongly typed distinct type can be generated when the user-defined distinct type is created.
- Row. A user-defined data type that represents a row. It includes one or more fields with associated data types that make up a row of data.
- Structured. A user-defined data type that represents an object and associated methods. It may include zero or more attributes and may be a subtype allowing attributes to be inherited from a supertype. Some methods are generated when the user-defined structured type is created and others can be specified as part of the definition.

CREATE TYPE (array)

The CREATE TYPE (array) statement defines an array type. The elements of an array type are based on one of the built-in data types or a user-defined distinct type.

Invocation

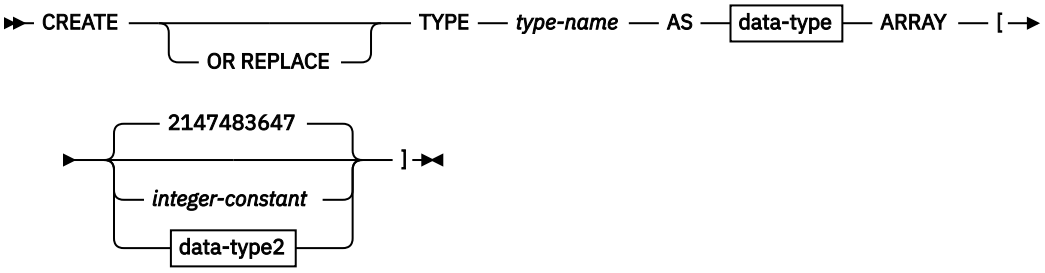
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

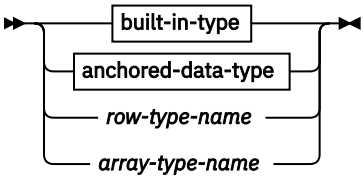
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- IMPLICIT_SCHEMA authority on the database, if the schema name of the array type does not refer to an existing schema
- CREATEIN privilege on the schema, if the schema name of the array type refers to an existing schema
- SCHEMAADM authority on the schema, if the schema name of the array type refers to an existing schema
- DBADM authority

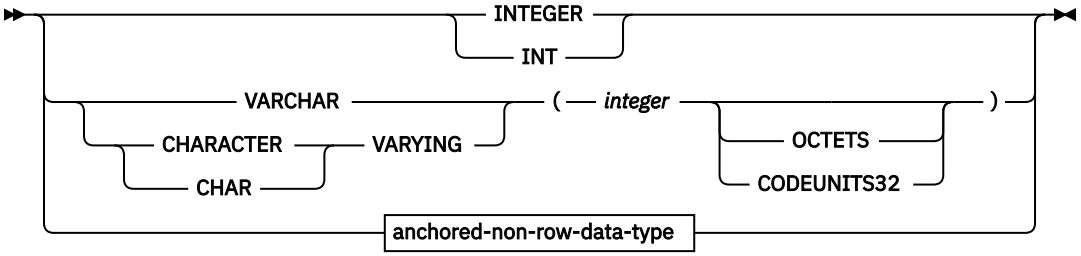
Syntax



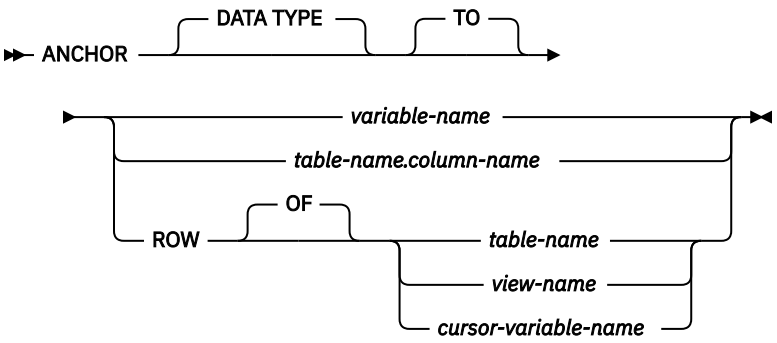
data-type



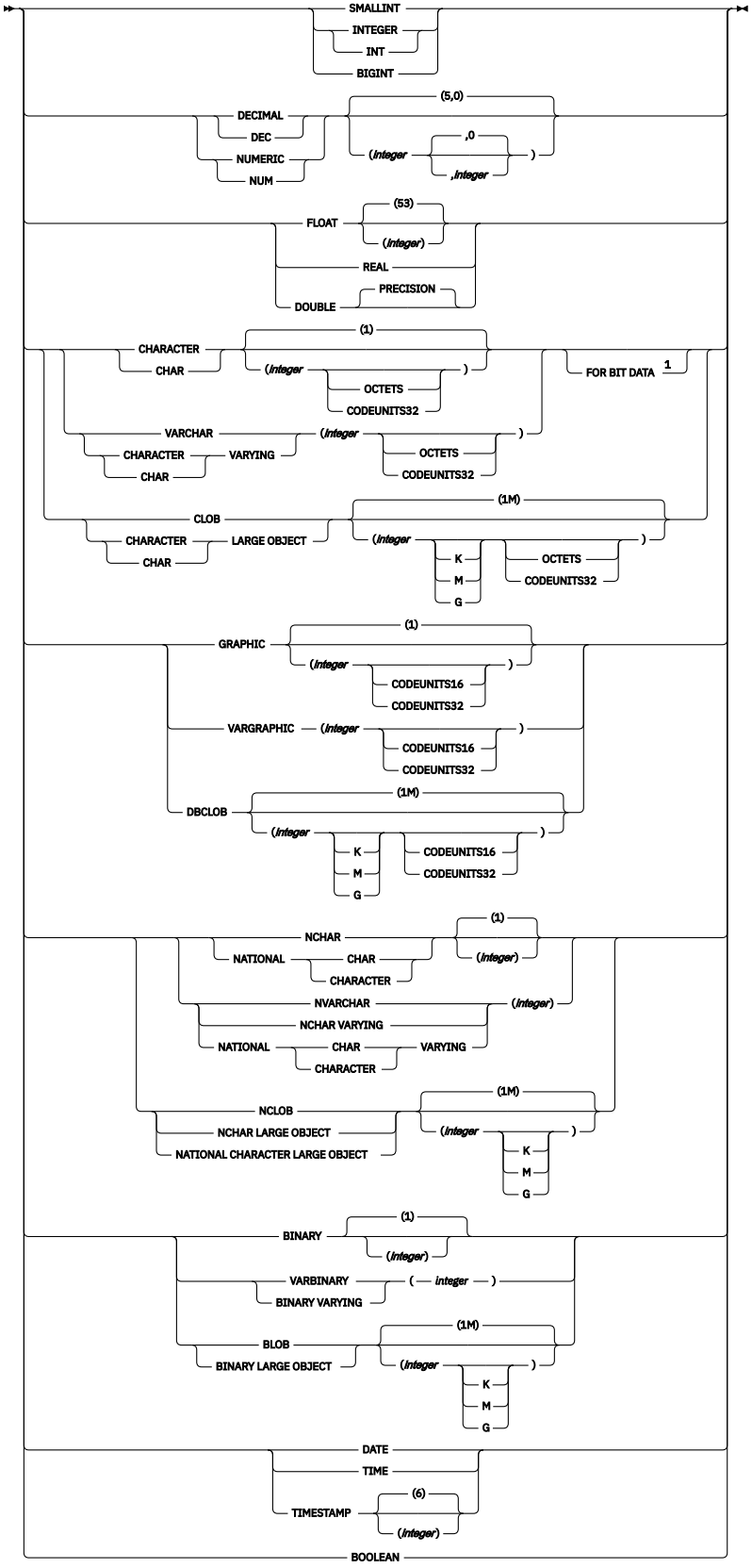
data-type2



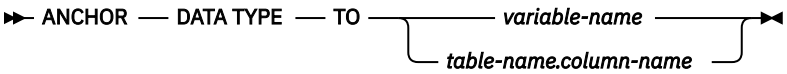
anchored-data-type



built-in-type



anchored-non-row-data-type



Notes:

[1] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

Description

OR REPLACE
Specifies to replace the definition for the data type if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog, with the exception that functions and methods are invalidated instead of dropped when they have parameters or a return value defined with the data type being replaced. The existing definition must not be a structured type (SQLSTATE 42809). This option is ignored if a definition for the data type does not exist at the current server.

type-name
Names the type. The name, including the implicit or explicit qualifier, must not identify any other type (built-in or user-defined) that already exists at the current server. The unqualified name must not be the same as the name of a built-in data type or VARBINARY (SQLSTATE 42918).

A number of names used as keywords in predicates are reserved for system use, and cannot be used as a *type-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators.

If a two-part *type-name* is specified, the schema name must not begin with the characters 'SYS' (SQLSTATE 42939).

data-type
Specifies the data type of the array elements.

built-in-type
Specifies a built-in data type. See "CREATE TABLE" for the description of built-in data types. Built-in types include the data types described in "CREATE TABLE", other than reference, SYSPROC.DB2SECURITYLABEL, XML, or user-defined types (SQLSTATE 429C2).

row-type-name
Specifies the name of a user-defined row type. If a *row-type-name* is specified without a schema name, the *row-type-name* is resolved by searching the schemas in the SQL path. Row types can be nested as elements in other array types with a maximum nesting level of sixteen.

array-type-name
Specifies an array type. If an *array-type-name* is specified without a schema name, the *array-type-name* is resolved by searching the schemas in the SQL path. Array types can be nested as elements in other array types with a maximum nesting level of sixteen.

anchored-data-type
Identifies another object used to determine the data type. The data type of the anchor object is bound by the same limitations that apply when specifying the data type directly, or in the case of a row, to creating a row type.

ANCHOR DATA TYPE TO
Indicates that an anchored data type is used to specify the data type.

variable-name
Identifies a global variable. The data type of the global variable is used as the data type for the array elements.

table-name.column-name
Identifies a column name of an existing table or view. The data type of the column is used as the data type for the array elements.

ROW OF *table-name* or *view-name*
Specifies a row of fields with names and data types that are based on the column names and column data types of the table identified by *table-name* or the view identified by *view-name*.The data type of the array elements is an unnamed row type.

ROW OF *cursor-variable-name*
Specifies a row of fields with names and data types that are based on the field names and field data types of the cursor variable identified by *cursor-variable-name*. The specified cursor variable must be one of the following elements (SQLSTATE 428HS):

- A global variable with a strongly typed cursor data type
- A global variable with a weakly typed cursor data type that was created or declared with a CONSTANT clause specifying a *select-statement* where all the result columns are named.

If the cursor type of the cursor variable is not strongly-typed using a named row type, the data type of the array elements is an unnamed row type.

anchored-non-row-data-type
Identifies another object used to determine the data type. The data type of the anchor object is bound by the same limitations that apply when specifying the data type directly.

ANCHOR DATA TYPE TO
Indicates that an anchored data type is used to specify the data type.

variable-name
Identifies a global variable with a data type that is an INTEGER or VARCHAR data type. The data type of the global variable is used as the data type for the array index.

table-name.column-name
Identifies a column name of an existing table or view with a data type that is an INTEGER or VARCHAR data type. The data type of the column is used as the data type for the array index.

ARRAY [*integer-constant*]
Specifies that the type is an array with a maximum cardinality of *integer-constant*. The value must be a positive integer (not zero) and less than the largest positive integer value (SQLSTATE 42820). The default is the largest positive integer value (2 147 483 647). The cardinality of an array value is determined by the highest element position assigned to the array value.

The maximum cardinality of an array on a given system is limited by the total amount of memory available to database applications. As such, although arrays of large cardinalities can be created, not all elements might be available for use.

ARRAY[*data-type2*]
Specifies that the type is an associative array that is indexed with values of data type *data-type2*. The data type must be either the INTEGER or VARCHAR data type (SQLSTATE 429C2). The values specified as the index when assigning an array element must be assignable to a value of *data-type2*. The cardinality of an array value is determined by the number of unique index values used when assigning array elements.

Rules

- ***Use of anchored data types:*** An anchored data type cannot refer to the following objects (SQLSTATE 428HS): a nickname, typed table, typed view, statistical view that is associated with an expression-based index, declared temporary table, row definition that is associated with a weakly typed cursor, object with a code page or collation that is different from the database code page or database collation.

Notes

- ***Array type usage:*** An array type can only be used as the data type of:
  - A local variable in a compound SQL (compiled) statement
  - A parameter of an SQL routine
  - A parameter of a Java procedure (non-nested ordinary arrays only)
  - The returns type of an SQL function
  - A global variable

- A variable or parameter defined with an array type can only be used in compound SQL (compiled) statements

Examples

Example 1: Create an array type named PHONENUMBERS with a maximum of 50 elements that are of the DECIMAL(10, 0) data type.

```
CREATE TYPE PHONENUMBERS AS DECIMAL(10,0)
  ARRAY[50]
```

Example 2: Create an array type named NUMBERS with the default number of elements in the schema GENERIC.

```
CREATE TYPE GENERIC.NUMBERS AS DECFLOAT(34)
  ARRAY[]
```

Example 3: Create an associative array named PERSONAL_PHONENUMBERS with elements that are DECIMAL(16, 0) that is indexed by strings like 'Home', 'Work', or 'Mom'.

```
CREATE TYPE PERSONALPHONENUMBERS AS DECIMAL(16, 0) ARRAY[VARCHAR(8)]
```

Example 4: Create an associative array type where the indexes are province, territory, or country names and the elements are capital cities:

```
CREATE TYPE CAPITALSARRAY AS VARCHAR(30) ARRAY[VARCHAR(20)]
```

Example 5: Create an associative array type for product descriptions of up to 40 characters long, where the indexes are the product numbers, which are a maximum of 12 characters long:

```
CREATE TYPE PRODUCTS AS VARCHAR(40) ARRAY[VARCHAR(12)]
```

CREATE TYPE (cursor)

The CREATE TYPE (cursor) statement defines a user-defined cursor type.

Invocation

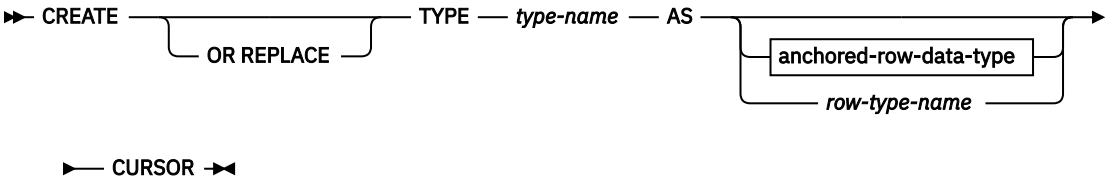
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

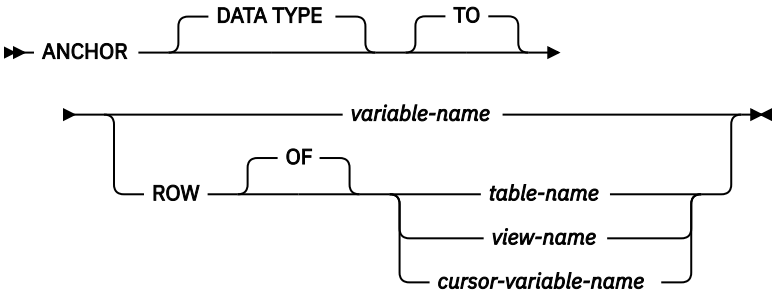
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- IMPLICIT_SCHEMA authority on the database, if the schema name of the cursor type does not refer to an existing schema
- CREATEIN privilege on the schema, if the schema name of the cursor type refers to an existing schema
- SCHEMAADM authority on the schema, if the schema name of the cursor type refers to an existing schema
- DBADM authority

Syntax



anchored-row-data-type



Description

OR REPLACE

Specifies to replace the definition for the data type if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog, with the exception that functions and methods are invalidated instead of dropped when they have parameters or a return value defined with the data type being replaced. The existing definition must not be a structured type (SQLSTATE 42809). This option is ignored if a definition for the data type does not exist at the current server.

type-name

Names the type. The name, including the implicit or explicit qualifier, must not identify any other type (built-in or user-defined) that already exists at the current server. The unqualified name must not be the same as the name of a built-in data type or BOOLEAN, BINARY or VARBINARY (SQLSTATE 42918).

A number of names used as keywords in predicates are reserved for system use, and cannot be used as a type-name (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators. If a two-part type-name is specified, the schema name must not begin with the characters 'SYS' (SQLSTATE 42939).

anchored-row-data-type

Identifies row information from another object used to determine the row type associated with the cursor type. The data type of the anchor object has the same limitations that apply to creating a row type.

ANCHOR DATA TYPE TO

Indicates an anchored data type is used to specify the data type.

variable-name

Identifies a global variable. The data type of the referenced variable must be a row type and is used as the row type associated with the cursor type.

ROW OF *table-name* or *view-name*

Specifies a row of fields with names and data types that are based on the column names and column data types of the table identified by *table-name* or the view identified by *view-name*. The data types of the anchor object columns have the same limitations that apply to field data types. The row type associated with the cursor type is an unnamed row type.

ROW OF *cursor-variable-name*
Specifies a row of fields with names and data types that are based on the field names and field data types of the cursor variable identified by *cursor-variable-name*. The specified cursor variable must be one of the following objects (SQLSTATE 428HS):

- A global variable with a strongly typed cursor data type
- A global variable with a weakly typed cursor data type that was created or declared with a CONSTANT clause specifying a *select-statement* where all the result columns are named.

If the cursor type of the cursor variable is not strongly-typed using a named row type, the row type associated with the cursor type is an unnamed row type.

row-type-name
Specifies the row type that will be used to check the row type of the result table of the *select-statement* assigned to a variable of the cursor type. The assignment fails if the type check fails (SQLSTATE 42821). If *row-type-name* is specified without a schema name, the row type is resolved by searching the schemas in the SQL path.

Rules

- ***Use of anchored data types:*** An anchored data type cannot refer to the following objects (SQLSTATE 428HS): a nickname, typed table, typed view, statistical view that is associated with an expression-based index, declared temporary table, row definition that is associated with a weakly typed cursor, object with a code page or collation that is different from the database code page or database collation.

Notes

- ***Cursor type usage:*** A cursor type can only be used as the data type of:
  - A local variable in a compound SQL (compiled) statement
  - A parameter of an SQL routine
  - The returns type of an SQL function
  - A global variable
- A variable or parameter defined with a cursor type can only be used in compound SQL (compiled) statements
- A variable or parameter that has a strongly-typed cursor type must not be used to assign cursor values that are based on a *statement-name* instead of a *select-statement*
- A user-defined cursor type with an associated row type is a strongly-typed cursor type; otherwise, it is a weakly-typed cursor type.

Examples

- *Example 1:* Create a cursor type that can be used with any cursor.

```
CREATE TYPE EMPCURSOR AS CURSOR
```

- *Example 2:* Create a strongly-typed cursor type that is based on the row data type DEPTROW:

```
CREATE TYPE DEPTCURSOR AS DEPTROW CURSOR
```

CREATE TYPE (distinct)

The CREATE TYPE (distinct) statement defines a distinct type. The distinct type is always sourced on one of the built-in data types and can be defined to use strong type or weak type rules..

Successful execution of the statement that defines a strongly typed distinct type also generates functions to cast between the distinct type and its source type and, optionally, generates support for the comparison operators (=, <>, <, <=, >, and >=) for use with the distinct type. Successful execution of the statement that defines a weakly typed distinct type does not generate any functions.

Invocation

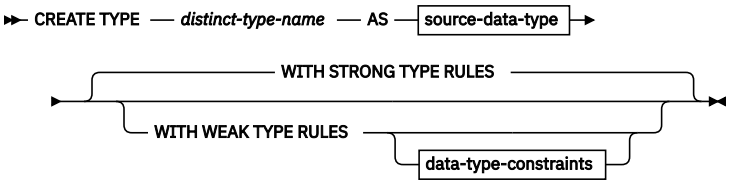
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

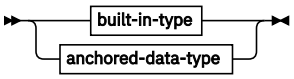
The privileges held by the authorization ID of the statement must include as least one of the following authorities:

- IMPLICIT_SCHEMA authority on the database, if the schema name of the distinct type does not refer to an existing schema
- CREATEIN privilege on the schema, if the schema name of the distinct type refers to an existing schema
- SCHEMAADM authority on the schema, if the schema name of the distinct type refers to an existing schema
- DBADM authority

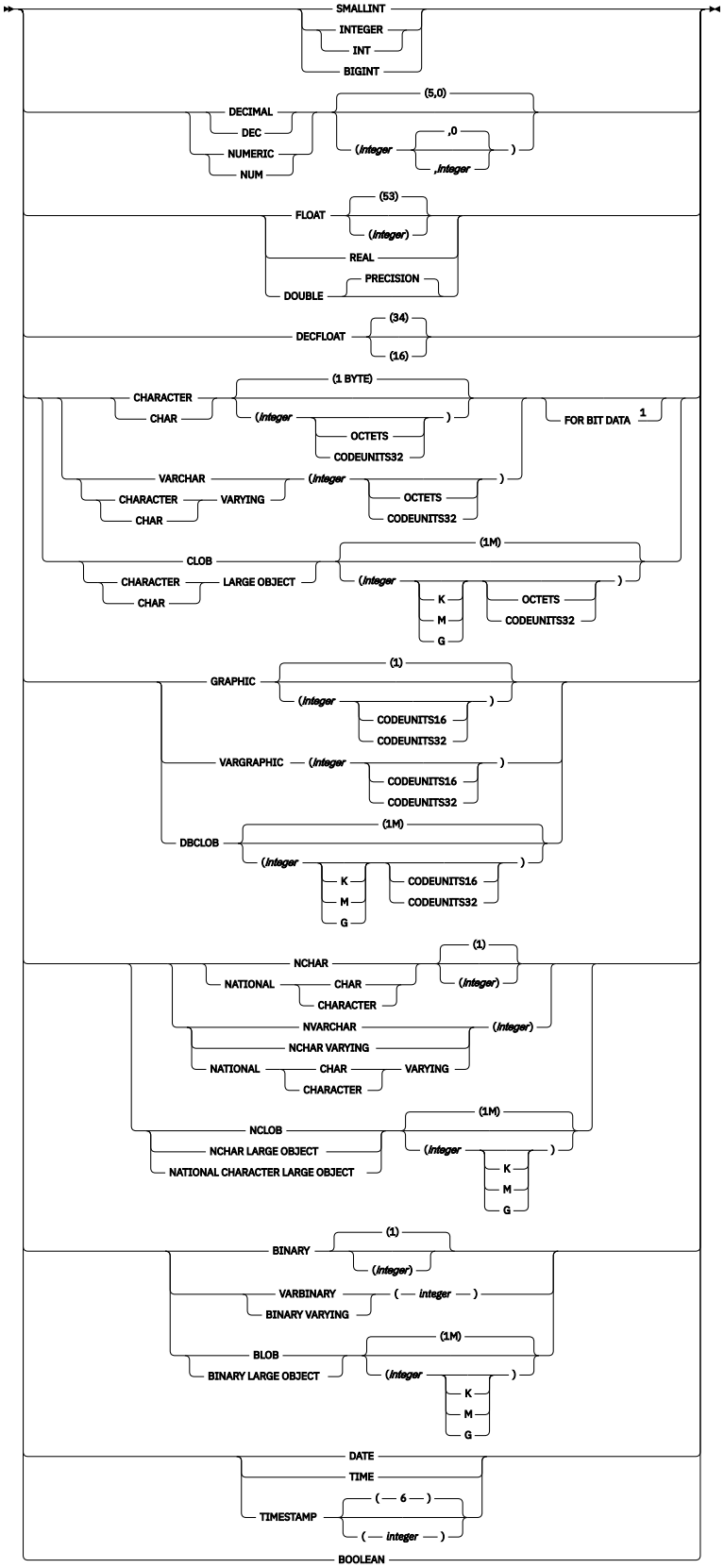
Syntax



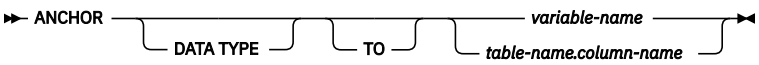
source-data-type



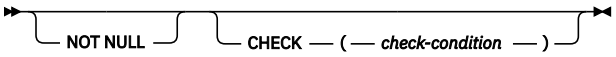
built-in-type



anchored-data-type



data-type-constraints



Notes:

[1] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

Description

distinct-type-name

Names the distinct type. The name, including the implicit or explicit qualifier, must not identify any other type (built-in or user-defined) that already exists at the current server. The unqualified name cannot be the same as the name of a built-in data type (SQLSTATE 42918), and cannot be ARRAY, INTERVAL, or ROWID.

In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The qualified form is a schema name followed by a period and an SQL identifier.

Several names used as keywords in predicates are reserved for system use and cannot be used as a distinct type name (SQLSTATE 42939). These names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators.

If a two-part name is specified, the schema name must not begin with the characters SYS (SQLSTATE 42939).

source-data-type

Specifies the data type used as the basis for the internal representation of the distinct type. The data type must be a built-in data type. For more information on built-in data types, see "CREATE TABLE". The source data type cannot be of type XML or an ARRAY type (SQLSTATE 42601). For portability of applications across platforms, use the following recommended data type names:

- DOUBLE or REAL instead of FLOAT
- DECIMAL instead of NUMERIC
- VARCHAR, BLOB, or CLOB instead of LONG VARCHAR
- VARGRAPHIC or DBCLOB instead of LONG VARGRAPHIC

anchored-data-type

Identifies another object used to determine the data type. The data type of the anchor object is bound by the same limitations that apply when specifying the data type directly.

ANCHOR DATA TYPE TO

Indicates that an anchored data type is used to specify the data type.

variable-name

Identifies a global variable with a data type that is a built-in type other than ROW or CURSOR. The data type of the global variable is used as the source data type for the distinct type.

table-name.column-name

Identifies a column name of an existing table or view with a data type that must be specified as a built-in-type. The data type of the column is used as the source data type for the distinct type.

WITH STRONG TYPE RULES

Specifies that strong typing rules are used for operations where this data type is an operand including assignments and comparisons. This is the default.

WITH WEAK TYPE RULES

Specifies that weak typing rules are used for operations where this data type is an operand including assignments, comparisons, and function resolution. When values of a weakly typed distinct type are used, the data type is effectively treated as the specified *source-data-type* when processing the operation.

data-type-constraints
Defines constraints on the distinct type that are applied when values are assigned or cast to the distinct type.

NOT NULL
Prevents a value with this distinct type from having a null value. If NOT NULL is not specified, a value with this distinct type can have the null value.

CHECK (*check-condition*)
Defines a data type check constraint. At any time, the check-condition must be true or unknown for every value with this data type. The *check-condition* is a form of the *search-condition* that conforms to the rules of table check constraints (SQLSTATE 426211) with the addition that the VALUE keyword is used to reference a value that is assigned or cast to the distinct type in the same way that a column name is referenced in a table check constraint. Note that the *check-condition* cannot reference global variables.

built-in-type
See "CREATE TABLE" for the description of built-in data types.

Rules

- ***Use of anchored data types:*** An anchored data type cannot refer to the following objects (SQLSTATE 428HS): a nickname, typed table, typed view, statistical view that is associated with an expression-based index, declared temporary table, row definition that is associated with a weakly typed cursor, object with a code page or collation that is different from the database code page or database collation.

Notes

- ***Privileges***: The definer of the user-defined type always receives the EXECUTE privilege WITH GRANT OPTION on all functions automatically generated for the distinct type.

  EXECUTE privilege on all functions automatically generated during the CREATE TYPE (Distinct) statement is granted to PUBLIC.
- Creating a distinct type with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- ***Additional generated functions***: When a strongly typed distinct type is created, the following functions are generated to cast to and from the source type:
  - One function to convert from the distinct type to the source type
  - One function to convert from the source type to the distinct type
  - One function to convert from INTEGER to the distinct type if the source type is SMALLINT
  - One function to convert from VARCHAR to the distinct type if the source type is CHAR
  - One function to convert from VARCHAR to the distinct type if the source type is BINARY
  - One function to convert from VARGRAPHIC to the distinct type if the source type is GRAPHIC.

  In general these functions will have the following format:

```
CREATE FUNCTION source-type-name (distinct-type-name)
  RETURNS source-type-name ...

CREATE FUNCTION distinct-type-name (source-type-name)
  RETURNS distinct-type-name ...
```

  In cases in which the source type is a parameterized type, the function to convert from the distinct type to the source type will have as function name the name of the source type without the parameters (see Table 146 on page 1492 for details). The type of the return value of this function will include the parameters given on the CREATE TYPE (Distinct) statement. The function to convert from the source

type to the distinct type will have an input parameter whose type is the source type including its parameters. For example,

```
CREATE TYPE T_SHOESIZE AS CHAR(2)

CREATE TYPE T_MILES AS DOUBLE
```

will generate the following functions:

```
FUNCTION CHAR (T_SHOESIZE) RETURNS CHAR (2)

FUNCTION T_SHOESIZE (CHAR (2))
RETURNS T_SHOESIZE

FUNCTION DOUBLE (T_MILES) RETURNS DOUBLE

FUNCTION T_MILES (DOUBLE) RETURNS T_MILES
```

The schema of the generated cast functions is the same as the schema of the distinct type. No other function with this name and with the same signature may already exist in the database (SQLSTATE 42710).

The following table gives the names of the functions to convert from the distinct type to the source type and from the source type to the distinct type for all predefined data types.

Table 146. CAST functions on distinct types

Source Type Name	Function Name	Parameter	Return-type
SMALLINT	*distinct-type-name*	SMALLINT	*distinct-type-name*
SMALLINT	*distinct-type-name*	INTEGER	*distinct-type-name*
SMALLINT	SMALLINT	*distinct-type-name*	SMALLINT
INTEGER	*distinct-type-name*	INTEGER	*distinct-type-name*
INTEGER	INTEGER	*distinct-type-name*	INTEGER
BIGINT	*distinct-type-name*	BIGINT	*distinct-type-name*
BIGINT	BIGINT	*distinct-type-name*	BIGINT
DECIMAL	*distinct-type-name*	DECIMAL (*p,s*)	*distinct-type-name*
DECIMAL	DECIMAL	*distinct-type-name*	DECIMAL (*p,s*)
NUMERIC	*distinct-type-name*	DECIMAL (*p,s*)	*distinct-type-name*
NUMERIC	DECIMAL	*distinct-type-name*	DECIMAL (*p,s*)
REAL	*distinct-type-name*	REAL	*distinct-type-name*
REAL	*distinct-type-name*	DOUBLE	*distinct-type-name*
REAL	REAL	*distinct-type-name*	REAL
FLOAT(*n*) where *n*<=24	*distinct-type-name*	REAL	*distinct-type-name*
FLOAT(*n*) where *n*<=24	*distinct-type-name*	DOUBLE	*distinct-type-name*
FLOAT(*n*) where *n*<=24	REAL	*distinct-type-name*	REAL
FLOAT(*n*) where *n*>24	*distinct-type-name*	DOUBLE	*distinct-type-name*
FLOAT(*n*) where *n*>24	DOUBLE	*distinct-type-name*	DOUBLE
FLOAT	*distinct-type-name*	DOUBLE	*distinct-type-name*
FLOAT	DOUBLE	*distinct-type-name*	DOUBLE

Table 146. CAST functions on distinct types (continued)

Source Type Name	Function Name	Parameter	Return-type
DOUBLE	*distinct-type-name*	DOUBLE	*distinct-type-name*
DOUBLE	DOUBLE	*distinct-type-name*	DOUBLE
DOUBLE PRECISION	*distinct-type-name*	DOUBLE	*distinct-type-name*
DOUBLE PRECISION	DOUBLE	*distinct-type-name*	DOUBLE
DECFLOAT	*distinct-type-name*	DECFLOAT(*n*)	*distinct-type-name*
DECFLOAT	DECFLOAT	*distinct-type-name*	DECFLOAT(*n*)
CHAR	*distinct-type-name*	CHAR (*n*)	*distinct-type-name*
CHAR	CHAR	*distinct-type-name*	CHAR (*n*)
CHAR	*distinct-type-name*	VARCHAR (*n*)	*distinct-type-name*
VARCHAR	*distinct-type-name*	VARCHAR (*n*)	*distinct-type-name*
VARCHAR	VARCHAR	*distinct-type-name*	VARCHAR (*n*)
CLOB	*distinct-type-name*	CLOB (*n*)	*distinct-type-name*
CLOB	CLOB	*distinct-type-name*	CLOB (*n*)
GRAPHIC	*distinct-type-name*	GRAPHIC (*n*)	*distinct-type-name*
GRAPHIC	GRAPHIC	*distinct-type-name*	GRAPHIC (*n*)
GRAPHIC	*distinct-type-name*	VARGRAPHIC (*n*)	*distinct-type-name*
VARGRAPHIC	*distinct-type-name*	VARGRAPHIC (*n*)	*distinct-type-name*
VARGRAPHIC	VARGRAPHIC	*distinct-type-name*	VARGRAPHIC (*n*)
DBCLOB	*distinct-type-name*	DBCLOB (*n*)	*distinct-type-name*
DBCLOB	DBCLOB	*distinct-type-name*	DBCLOB (*n*)
BINARY	*distinct-type-name*	BINARY (*n*)	*distinct-type-name*
BINARY	BINARY	*distinct-type-name*	BINARY (*n*)
BINARY	*distinct-type-name*	VARBINARY (*n*)	*distinct-type-name*
VARBINARY	*distinct-type-name*	VARBINARY (*n*)	*distinct-type-name*
VARBINARY	VARBINARY	*distinct-type-name*	VARBINARY (*n*)
BLOB	*distinct-type-name*	BLOB (*n*)	*distinct-type-name*
BLOB	BLOB	*distinct-type-name*	BLOB (*n*)
DATE	*distinct-type-name*	DATE	*distinct-type-name*
DATE	DATE	*distinct-type-name*	DATE
TIME	*distinct-type-name*	TIME	*distinct-type-name*
TIME	TIME	*distinct-type-name*	TIME
TIMESTAMP	*distinct-type-name*	TIMESTAMP(*p*	*distinct-type-name*
TIMESTAMP	TIMESTAMP	*distinct-type-name*	TIMESTAMP(*p*)
BOOLEAN	*distinct-type-name*	BOOLEAN	*distinct-type-name*

Table 146. CAST functions on distinct types (continued)

Source Type Name	Function Name	Parameter	Return-type
BOOLEAN	BOOLEAN	*distinct-type-name*	BOOLEAN

Note: NUMERIC and FLOAT are not recommended when creating a user-defined type for a portable application. DECIMAL and DOUBLE should be used instead.

The functions described in the preceding table and the comparison operator functions are the only functions that are generated automatically when distinct types are defined. Consequently, none of the built-in functions (AVG, MAX, LENGTH, and so on) are supported for strongly typed distinct types until the CREATE FUNCTION statement is used to register user-defined functions for the strongly typed distinct type, and those user-defined functions are sourced on the appropriate built-in functions. In particular, note that it is possible to register user-defined functions that are sourced on the built-in column functions.

When a strongly typed distinct type is created, system-generated comparison operators are created when the source type supports comparisons . Creation of these comparison operators will generate entries in the SYSCAT.ROUTINES catalog view for the new functions.

The schema name of the distinct type must be included in the SQL path or the FUNCPATH BIND option for successful use of these operators and cast functions in SQL statements.

- When a weakly typed distinct type is created, no additional functions need to be generated or created because the weak type rules allow a weakly typed distinct type to be used in the same context where the source type can be used.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - CREATE DISTINCT TYPE can be specified in place of CREATE TYPE
  - The LONG VARCHAR and LONG VARGRAPHIC data types and cast functions are supported but are deprecated and might be removed in a future release. The WITH COMPARISONS clause continues to not support the LONG VARCHAR and LONG VARGRAPHIC data types.
  - The WITH COMPARISONS clause, which specifies that system-generated comparison operators are to be created for comparing two instances of the distinct type, can be specified as the last clause of the statement if WITH WEAK TYPE RULES is not specified. Use WITH COMPARISONS only if it is required for compatibility with earlier versions of products in the Db2 family. If the source data type is either BLOB, CLOB, or DBCLOB and WITH COMPARISONS is specified, a warning occurs as in previous releases.
  - ALLOW NULL, or just NULL, can be specified as the opposite of NOT NULL. This is the default nullability characteristic of the distinct type if neither the ALLOW NULL clause nor the NOT NULL clause are specified. Specification of ALLOW NULL is not considered to define a data type constraint for the distinct type.

Examples

- *Example 1:* Create a strongly typed distinct type named SHOESIZE that is based on an INTEGER data type.

```
CREATE TYPE SHOESIZE AS INTEGER
```

This will also result in the creation of comparison operators (=, <>, <, <=, >, >=) and cast functions INTEGER(SHOESIZE) returning INTEGER and SHOESIZE(INTEGER) returning SHOESIZE.

- *Example 2:* Create a strongly typed distinct type named MILES that is based on a DOUBLE data type.

```
CREATE TYPE MILES AS DOUBLE
```

This will also result in the creation of comparison operators (=, <>, <, =, >, >=) and cast functions DOUBLE(MILES) returning DOUBLE and MILES(DOUBLE) returning MILES.

- *Example 3:* Create a weakly typed distinct type named BONUS that is based on an INTEGER data type and represents a percentage which cannot exceed 100.

```
CREATE TYPE BONUS AS INTEGER WITH WEAK TYPE RULES
      CHECK(VALUE >= 0 AND VALUE <= 100)
```

Because it is defined with weak type rules, comparison and cast functions are not generated for the weakly typed distinct type called BONUS.

- *Example 4:* Create a weakly typed distinct type named SALARY that is based on a DOUBLE data type which cannot be NULL and where the upper range is limited to less than one hundred thousand.

```
CREATE TYPE SALARY AS DOUBLE WITH WEAK TYPE RULES
      NOT NULL CHECK(VALUE < 100000)
```

CREATE TYPE (row)

The CREATE TYPE (row) statement defines a row type. A row type includes one or more fields with associated data types that make up a row of data.

Invocation

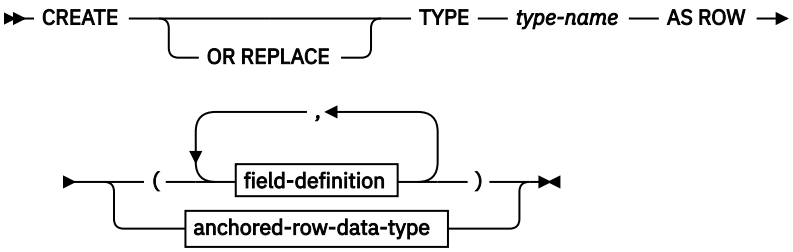
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

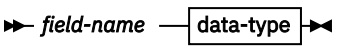
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- IMPLICIT_SCHEMA authority on the database, if the schema name of the row type does not refer to an existing schema
- CREATEIN privilege on the schema, if the schema name of the row type refers to an existing schema
- SCHEMAADM authority on the schema, if the schema name of the row type refers to an existing schema
- DBADM authority

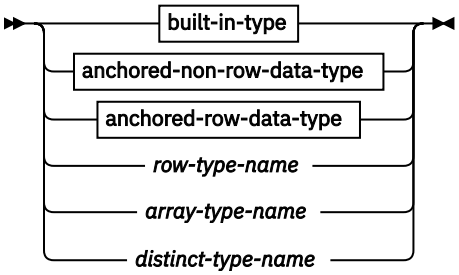
Syntax



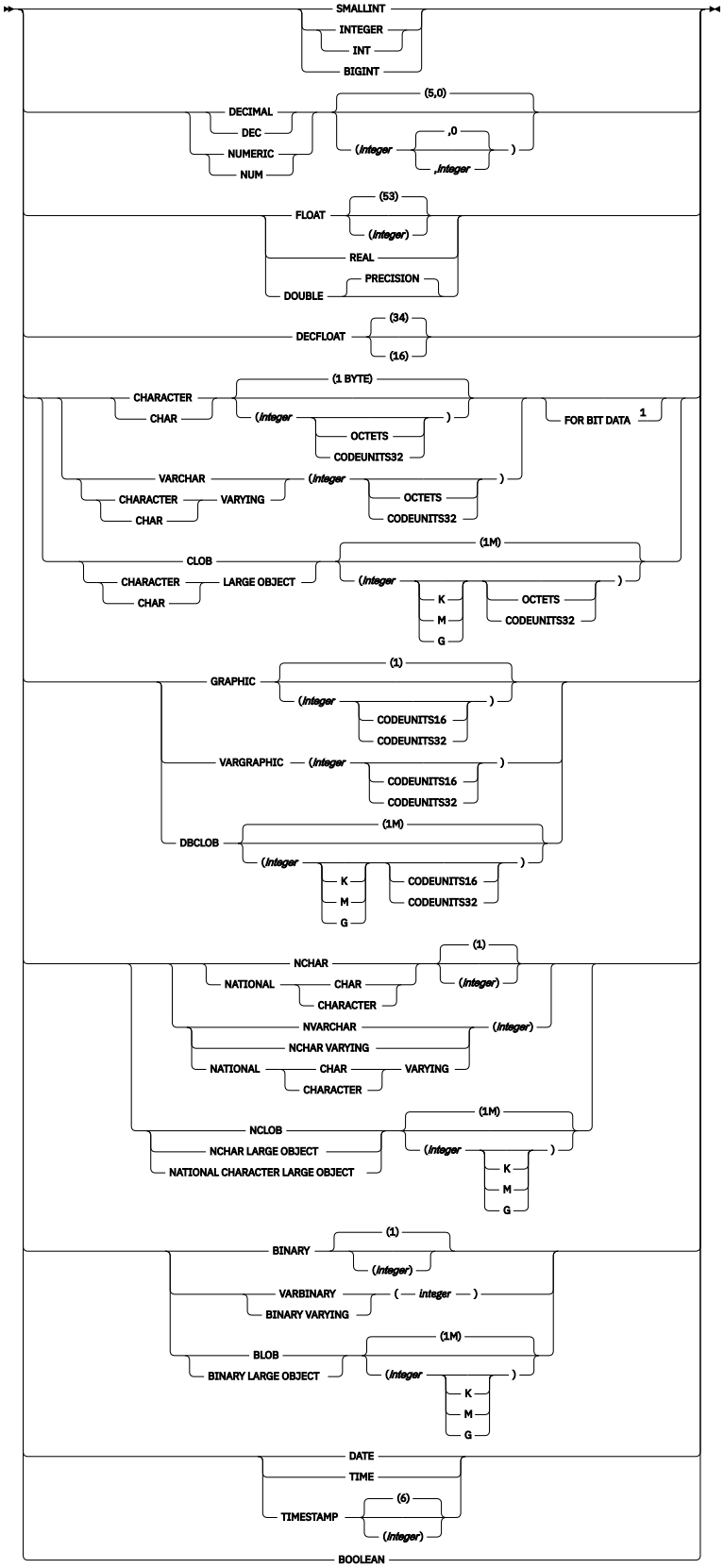
field-definition



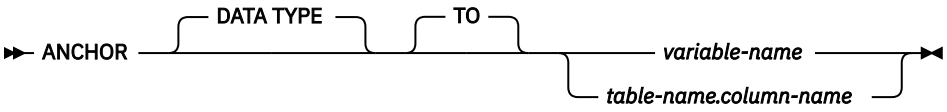
data-type



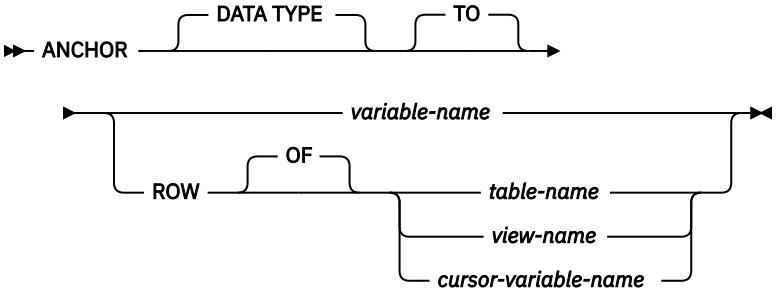
built-in type



anchored-non-row-data-type



anchored-row-data-type



Notes:

[1] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

Description

OR REPLACE
Specifies to replace the definition for the data type if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog, with the exception that functions and methods are invalidated instead of dropped when they have parameters or a return value defined with the data type being replaced. The existing definition must not be a structured type (SQLSTATE 42809). This option is ignored if a definition for the data type does not exist at the current server.

type-name
Names the type. The name, including the implicit or explicit qualifier, cannot identify any other type (built-in, structured, array, row, or distinct) already described in the catalog. The unqualified name cannot be the same as the name of a built-in data type (SQLSTATE 42918).

Several names used as keywords in predicates are reserved for system use, and cannot be used as a type name (SQLSTATE 42939). These names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators.

If a two-part name is specified, the schema name cannot begin with the characters SYS (SQLSTATE 42939).

field-definition
Defines the fields of the row type.

field-name
Specifies the name of a field within the row type. The name cannot be the same as any other field of this row type (SQLSTATE 42711).

data-type
Specifies the data type of the field.

built-in-type
Specifies a built-in data type. See "CREATE TABLE" for the description of built-in data types. Built-in types include the data types described in "CREATE TABLE", other than reference, SYSPROC.DB2SECURITYLABEL, XML, or user-defined types (SQLSTATE 429C2).

row-type-name
Specifies the name of a user-defined row type. If a *row-type-name* is specified without a schema name, the *row-type-name* is resolved by searching the schemas in the SQL path. Row types can be nested as field types of a row type with a maximum nesting level of sixteen.

array-type-name
Specifies an array type. If an *array-type-name* is specified without a schema name, the *array-type-name* is resolved by searching the schemas in the SQL path. Array types can be nested as field types of a row type with a maximum nesting level of sixteen.

distinct-type-name
Specifies a user-defined distinct data type. The specified distinct type cannot have any data type constraints (SQLSTATE 429C5).

anchored-non-row-data-type
Identifies another object used to determine the data type. The data type of the anchor object is has the same limitations that apply when specifying the data type directly.

ANCHOR DATA TYPE TO
Indicates that an anchored data type is used to specify the data type.

variable-name
Identifies a global variable with a data type that is a supported row field data type. The data type of the global variable is used as the data type for the field.

table-name.column-name
Identifies a column name of an existing table or view with a data type that is a built-in-type or a distinct type. The data type of the column is used as the data type for the field.

anchored-row-data-type
Identifies row information from another object to use as the fields of the row.

ANCHOR DATA TYPE TO
Indicates an anchored data type is used to specify the data type.

variable-name
Identifies a global variable. The data type of the referenced variable must be a row type.

ROW OF *table-name* or *view-name*
Specifies a row of fields with names and data types that are based on the column names and column data types of the table identified by *table-name* or the view identified by *view-name*. The data types of the anchor object columns have the same limitations that apply to field data types.

ROW OF *cursor-variable-name*
Specifies a row of fields with names and data types that are based on the field names and field data types of the cursor variable identified by *cursor-variable-name*. The specified cursor variable must be one of the following objects (SQLSTATE 428HS):

- A global variable with a strongly typed cursor data type
- A global variable with a weakly typed cursor data type that was created or declared with a CONSTANT clause specifying a *select-statement* where all the result columns are named.

Rules

- ***Use of anchored data types:*** An anchored data type cannot refer to the following objects (SQLSTATE 428HS): a nickname, typed table, typed view, statistical view that is associated with an expression-based index, declared temporary table, row definition that is associated with a weakly typed cursor, object with a code page or collation that is different from the database code page or database collation.

Notes

- ***Row type usage:*** A row type can only be used as the data type of:
  - A local variable in a compound SQL (compiled) statement
  - A parameter of an SQL routine
  - The returns type of an SQL function
  - The element of an array type

  - A user-defined cursor type
  - A global variable
- A variable or parameter defined with a row type can only be used in compound SQL (compiled) statements

Example

- Create a row type based on the columns of the DEPARTMENT table.

```
CREATE TYPE DEPTROW AS ROW (DEPTNO   VARCHAR(3),
                            DEPTNAME VARCHAR(29),
                            MGRNO    CHAR(6),
                            ADMRDEPT CHAR(3),
                            LOCATION CHAR(16))
```

CREATE TYPE (structured)

The CREATE TYPE statement defines a user-defined structured type.

A user-defined structured type can include zero or more attributes. A structured type can be a subtype allowing attributes to be inherited from a supertype. Successful execution of the statement generates methods, for retrieving and updating values of attributes. Successful execution of the statement also generates functions, for constructing instances of a structured type used in a column, for casting between the reference type and its representation type, and for supporting the comparison operators (=, <>, <, <=, >, and >=) on the reference type.

The CREATE TYPE statement also defines any method specifications for user-defined methods to be used with the user-defined structured type.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

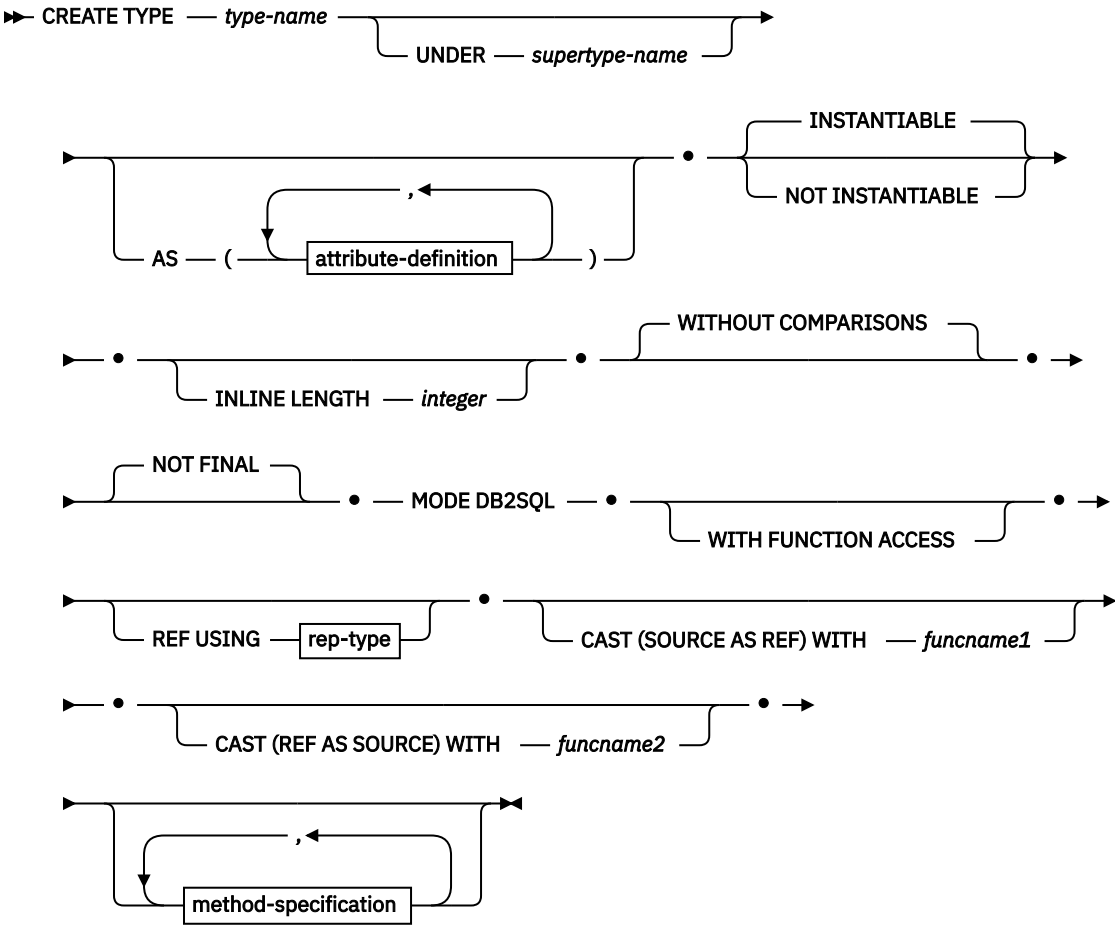
Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

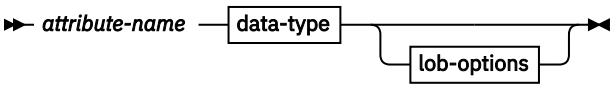
- IMPLICIT_SCHEMA authority on the database, if the schema name of the type does not refer to an existing schema
- CREATEIN privilege on the schema, if the schema name of the type refers to an existing schema
- SCHEMAADM authority on the schema, if the schema name of the type refers to an existing schema
- DBADM authority

If UNDER is specified, and the authorization ID of the statement is not the same as the owner of the root type of the type hierarchy, SCHEMAADM authority on the schema containing the root type is required or DBADM authority is required.

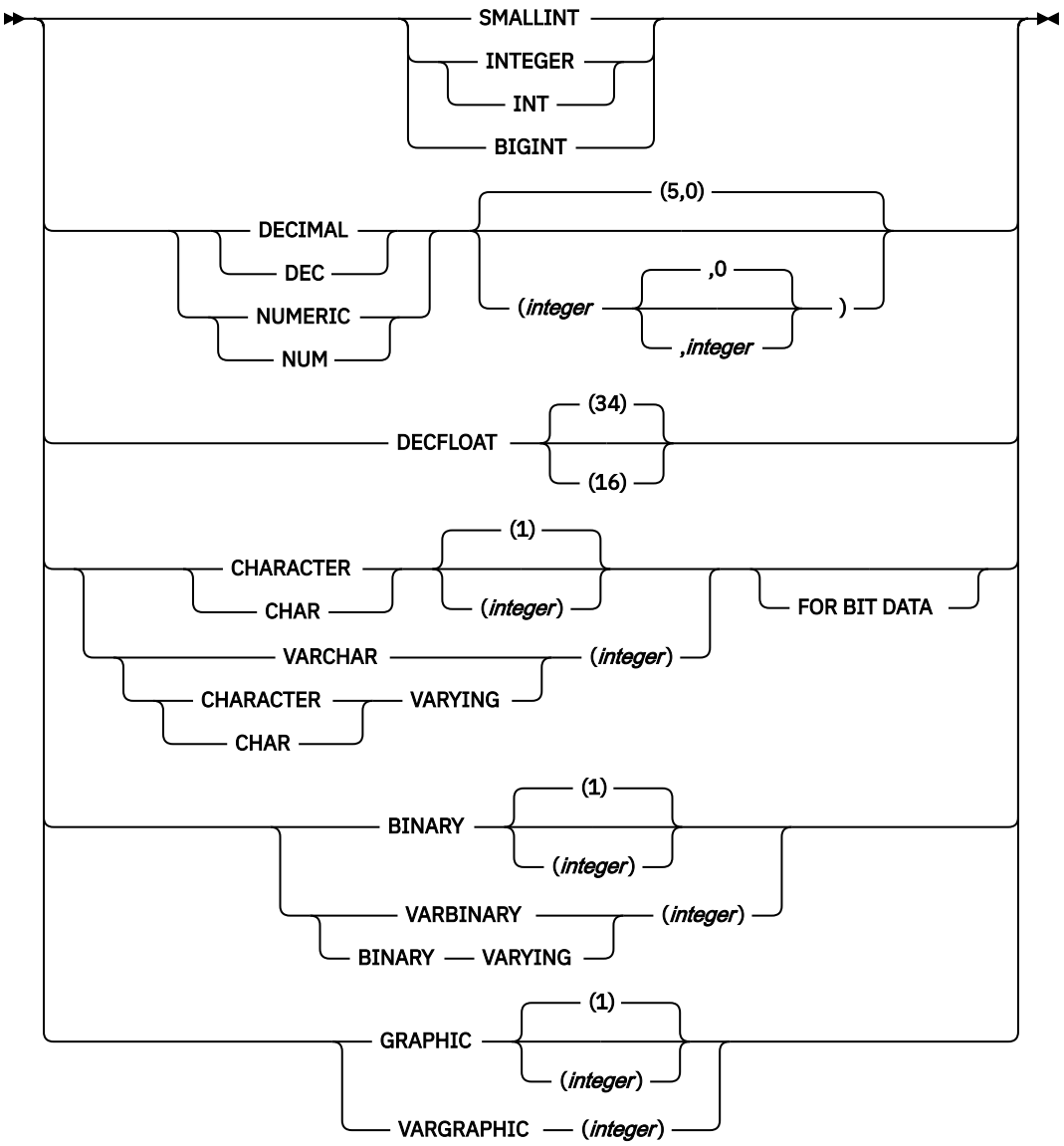
Syntax



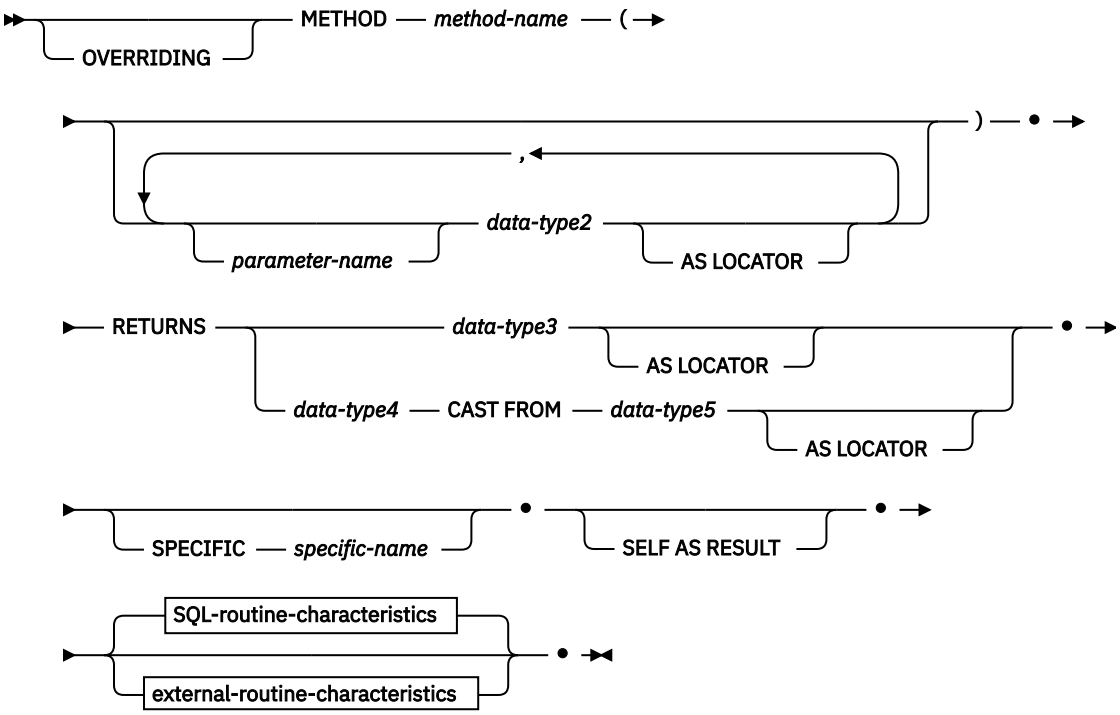
attribute-definition



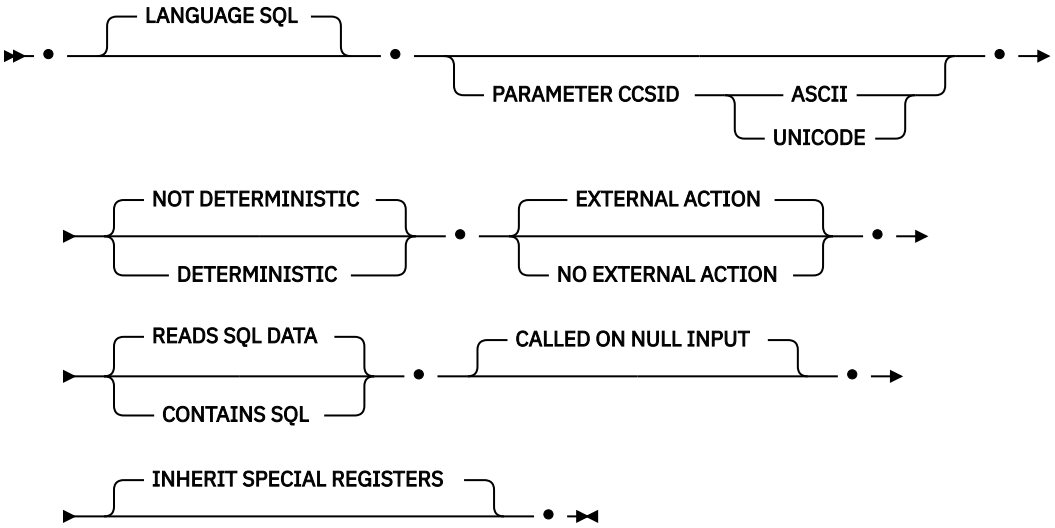
rep-type



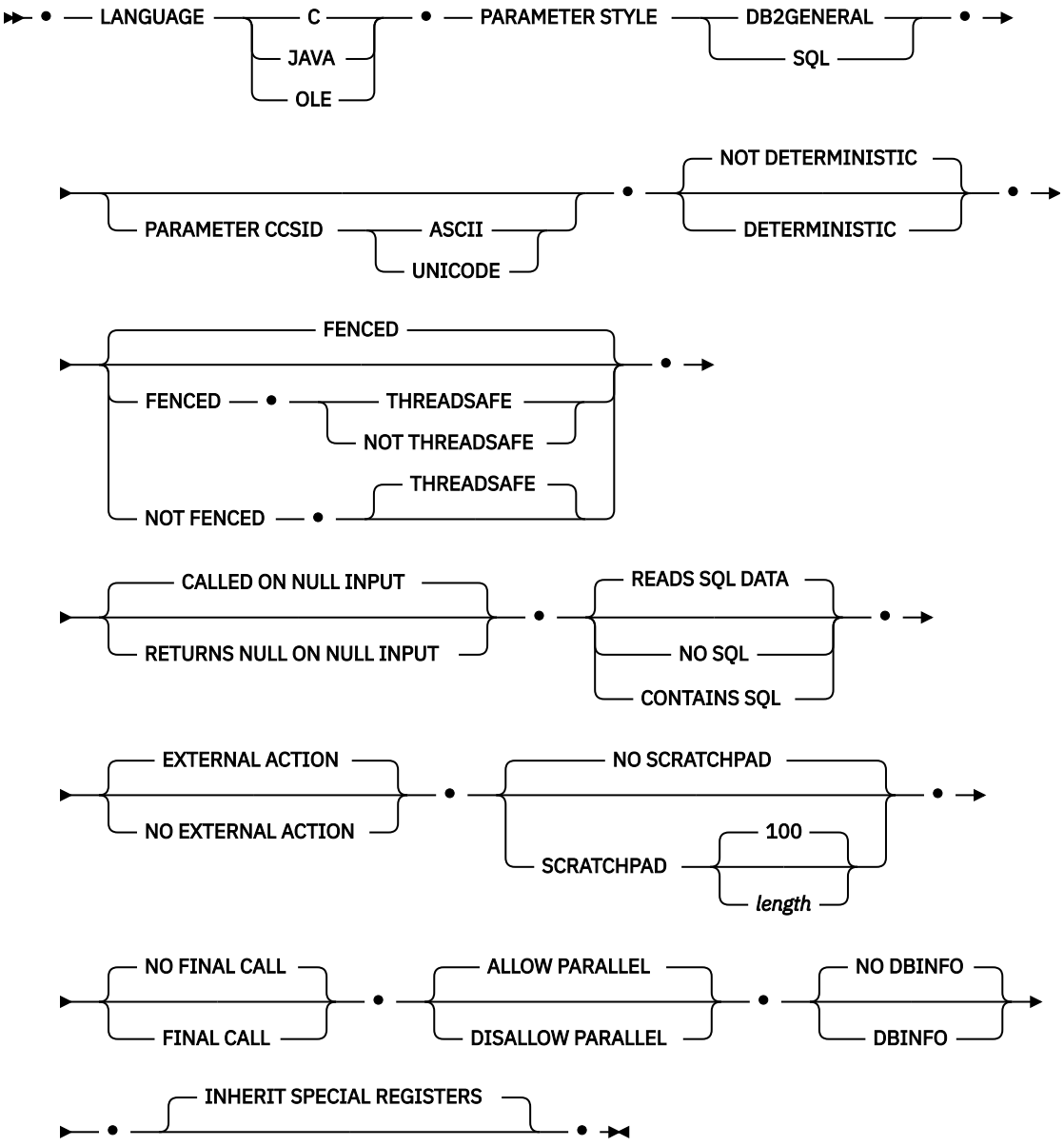
method-specification



SQL-routine-characteristics



external-routine-characteristics



Description

type-name

Names the type. The name, including the implicit or explicit qualifier, must not identify any other type (built-in, structured, or distinct) that already exists at the current server. The unqualified name must not be the same as the name of a built-in data type or BOOLEAN (SQLSTATE 42918). The unqualified name should also not be ARRAY, INTERVAL, or ROWID. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile or bind option implicitly specifies the qualifier for unqualified object names.

A number of names used as keywords in predicates are reserved for system use, and cannot be used as a *type-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators.

If a two-part *type-name* is specified, the schema name must not begin with the characters 'SYS' (SQLSTATE 42939).

UNDER *supertype-name*
Specifies that this structured type is a subtype under the specified *supertype-name*. The *supertype-name* must identify an existing structured type (SQLSTATE 42704). If *supertype-name* is specified without a schema name, the type is resolved by searching the schemas on the SQL path. The structured type includes all the attributes of the supertype followed by the additional attributes given in the *attribute-definition*.

attribute-definition
Defines the attributes of the structured type.

attribute-name
The name of an attribute. The *attribute-name* cannot be the same as any other attribute of this structured type or any supertype of this structured type (SQLSTATE 42711).

A number of names used as keywords in predicates are reserved for system use, and cannot be used as an *attribute-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators.

data-type
The data type of the attribute. It is one of the data types listed under "CREATE TABLE", other than XML or a weakly typed distinct type (SQLSTATE 42601). The data type must identify an existing data type (SQLSTATE 42704). If *data-type* is specified without a schema name, the type is resolved by searching the schemas on the SQL path. The description of various data types is given in "CREATE TABLE". If the attribute data type is a reference type, the target type of the reference must be a structured type that exists, or is created by this statement (SQLSTATE 42704).

To prevent type definitions that would, at run time, permit an instance of the type to directly or indirectly contain another instance of the same type or one of its subtypes, a type cannot be defined such that one of its attribute types directly or indirectly uses itself (SQLSTATE 428EP).

Character and graphic string data types cannot specify string units of CODEUNITS32.

lob-options
Specifies the options associated with LOB types (or distinct types based on LOB types). For a detailed description of *lob-options*, see "CREATE TABLE".

INSTANTIABLE or NOT INSTANTIABLE
Determines whether an instance of the structured type can be created. Implications of not instantiable structured types are:

- no constructor function is generated for a non-instantiable type
- a non-instantiable type cannot be used as the type of a table or view (SQLSTATE 428DP)
- a non-instantiable type can be used as the type of a column (only null values or instances of instantiable subtypes can be inserted into the column.

To create instances of a non-instantiable type, instantiable subtypes must be created. If NOT INSTANTIABLE is specified, no instance of the new type can be created.

INLINE LENGTH *integer*
This option indicates the maximum size (in bytes) of a structured type column instance to store inline with the rest of the values in the row of a table. Instances of a structured type or its subtypes, that are larger than the specified inline length, are stored separately from the base table row, similar to the way that LOB values are handled.

If the specified INLINE LENGTH is smaller than the size of the result of the constructor function for the newly-created type (32 bytes plus 10 bytes per attribute) and smaller than 292 bytes, an error results (SQLSTATE 429B2). Note that the number of attributes includes all attributes inherited from the supertype of the type.

The INLINE LENGTH for the type, whether specified or a default value, is the default inline length for columns that use the structured type. This default can be overridden at CREATE TABLE time.

INLINE LENGTH has no meaning when the structured type is used as the type of a typed table.

The default INLINE LENGTH for a structured type is calculated by the system. In the formulae that follow, the following terms are used:

short attribute
refers to an attribute with any of the following data types: SMALLINT, INTEGER, BIGINT, REAL, DOUBLE, FLOAT, DATE, or TIME. Also included are distinct types or reference types based on these types.

non-short attribute
refers to an attribute of any of the remaining data types, or distinct types based on those data types.

The system calculates the default inline length as follows:

1. Determine the added space requirements for non-short attributes using the following formula:

   *space_for_non_short_attribute*s = SUM(*attributelength* + n)

   n is defined as:

   - 0 bytes for nested structured type attributes
   - 2 bytes for non-LOB attributes
   - 9 bytes for LOB attributes

   attributelength is based on the data type specified for the attribute as shown in Table 147 on page 1506.
2. Calculate the total default inline length using the following formula:

   default_length(*structured_type*) = (*number_of_attributes* * 10) + 32 + *space_for_non-short_attributes*

   number_of_attributes is the total number of attributes for the structured type, including attributes that are inherited from its supertype. However, *number_of_attributes* does not include any attributes defined for any subtype of *structured_type*.

Table 147. Byte Counts for Attribute Data Types

Attribute Data Type	**Byte Count**
DECIMAL	The integral part of (*p* / 2) + 1, where *p* is the precision
DECFLOAT(*n*)	If *n* is 16, the byte count is 8; if *n* is 34, the byte count is 16
CHAR(*n*)	*n*
VARCHAR(*n*)	*n*
GRAPHIC(*n*)	*n* * 2
VARGRAPHIC(*n*)	*n* * 2
TIMESTAMP	10
LOB type	Each LOB attribute has a LOB descriptor in the structured type instance that points to the location of the actual value. The size of the descriptor varies according to the maximum length defined for the LOB attribute (see Table 148 on page 1507.
Distinct type	Length of the source type of the distinct type
Reference type	Length of the built-in data type on which the reference type is based
Structured type	inline_length(*attribute_type*)

Table 148. LOB Descriptor Size as a Function of the Maximum LOB Length	
Maximum LOB Length	**LOB Descriptor Size**
1024	68
8192	92
65 536	116
524 000	140
4 190 000	164
134 000 000	196
536 000 000	220
1 070 000 000	252
1 470 000 000	276
2 147 483 647	312

WITHOUT COMPARISONS
Indicates that there are no comparison functions supported for instances of the structured type.

NOT FINAL
Indicates that the structured type may be used as a supertype.

MODE DB2SQL
This clause is required and allows for direct invocation of the constructor function on this type.

WITH FUNCTION ACCESS
Indicates that all methods of this type and its subtypes, including methods created in the future, can be accessed using functional notation. This clause can be specified only for the root type of a structured type hierarchy (the UNDER clause is not specified) (SQLSTATE 42613). This clause is provided to allow the use of functional notation for those applications that prefer this form of notation over method invocation notation.

REF USING *rep-type*
Defines the built-in data type used as the representation (underlying data type) for the reference type of this structured type and all its subtypes. This clause can only be specified for the root type of a structured type hierarchy (UNDER clause is not specified) (SQLSTATE 42613). The *rep-type* cannot be a REAL, FLOAT, DECFLOAT, BLOB, CLOB, DBCLOB, array type, or structured type, and must have a length less than or equal to 32 672 bytes (SQLSTATE 42613).

If this clause is not specified for the root type of a structured type hierarchy, then REF USING VARCHAR(16) FOR BIT DATA is assumed.

CAST (SOURCE AS REF) WITH *funcname1*
Defines the name of the system-generated function that casts a value with the data type *rep-type* to the reference type of this structured type. A schema name must not be specified as part of *funcname1* (SQLSTATE 42601). The cast function is created in the same schema as the structured type. If the clause is not specified, the default value for *funcname1* is *type-name* (the name of the structured type). A function signature matching *funcname1(rep-type)* must not already exist in the same schema (SQLSTATE 42710).

CAST (REF AS SOURCE) WITH *funcname2*
Defines the name of the system-generated function that casts a reference type value for this structured type to the data type *rep-type*. A schema name must not be specified as part of *funcname2* (SQLSTATE 42601). The cast function is created in the same schema as the structured type. If the clause is not specified, the default value for *funcname2* is *rep-type* (the name of the representation type).

method-specification

Defines the methods for this type. A method cannot actually be used until it is given a body with a CREATE METHOD statement (SQLSTATE 42884).

OVERRIDING

Specifies that the method being defined overrides a method of a supertype of the type being defined. Overriding enables one to re-implement methods in subtypes, thereby providing more specific functionality. Overriding is not supported for the following types of methods:

- Table and row methods
- External methods declared with PARAMETER STYLE JAVA
- Methods that can be used as predicates in an index extension
- System-generated mutator or observer methods

Attempting to override such a method will result in an error (SQLSTATE 42745).

If a method is to be a valid overriding method, there must already exist one original method for one of the proper supertypes of the type being defined, and the following relationships must exist between the overriding method and the original method:

- The method name of the method being defined and the original method are equivalent.
- The method being defined and the original method have the same number of parameters.
- The data type of each parameter of the method being defined and the data type of the corresponding parameters of the original method are identical. This requirement excludes the implicit SELF parameter.

If such an original method does not exist, an error is returned (SQLSTATE 428FV).

The overriding method inherits the following attributes from the original method:

- Language
- Determinism indication
- External action indication
- An indication whether this method should be called if any of its arguments is the null value
- Result cast (if specified in the original method)
- SELF AS RESULT indication
- The SQL-data access or CONTAINS SQL indication
- For external methods:
  - Parameter style
  - Locator indication of the parameters and of the result (if specified in the original method)
  - FENCED, SCRATCHPAD, FINAL CALL, ALLOW PARALLEL, and DBINFO indication
  - INHERIT SPECIAL REGISTER and THREADSAFE indication

method-name

Names the method being defined. It must be an unqualified SQL identifier (SQLSTATE 42601). The method name is implicitly qualified with the schema used for CREATE TYPE.

A number of names used as keywords in predicates are reserved for system use, and cannot be used as a *method-name* (SQLSTATE 42939). The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH, and the comparison operators.

In general, the same name can be used for more than one method if there is some difference in their signatures.

parameter-name

Identifies the parameter name. It cannot be SELF, which is the name for the implicit subject parameter of a method (SQLSTATE 42734). If the method is an SQL method, all its parameters

must have names (SQLSTATE 42629). If the method being declared overrides another method, the parameter name must be exactly the same as the name of the corresponding parameter of the overridden method; otherwise, an error is returned (SQLSTATE 428FV).

data-type2
Specifies the data type of each parameter. One entry in the list must be specified for each parameter that the method will expect to receive. No more than 90 parameters are allowed, including the implicit SELF parameter. If this limit is exceeded, an error is raised (SQLSTATE 54023).

You can specify SQL data types and abbreviations that can be specified as a column type in the CREATE TABLE statement, and that have equivalents in the language that is being used to write the method. For details on the mapping between SQL data types and host language data types, see the topic that pertains to your language from the following list of related topics.

Note: If the SQL data type in question is a structured type, there is no default mapping to a host language data type. A user-defined transform function must be used to create a mapping between the structured type and the host language data type.

DECIMAL (or NUMERIC) and decimal floating-point are invalid with LANGUAGE C and OLE (SQLSTATE 42815).

XML data types cannot be used (SQLSTATE 42815).

REF may be specified, but it does not have a defined scope. Inside the body of the method, a reference-type can be used in a path-expression only by first casting it to have a scope. Similarly, a reference returned by a method can be used in a path-expression only by first casting it to have a scope.

AS LOCATOR
For LOB types or distinct types which are based on a LOB type, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed to the method instead of the actual value. This saves greatly in the number of bytes passed to the method, and may save as well in performance, particularly in the case where only a few bytes of the value are actually of interest to the method.

An error is raised (SQLSTATE 42601) if AS LOCATOR is specified for a type other than a LOB or a distinct type based on a LOB.

If the method is FENCED, or if LANGUAGE is SQL, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

If the method being declared overrides another method, the AS LOCATOR indication of the parameter must match exactly the AS LOCATOR indication of the corresponding parameter of the overridden method (SQLSTATE 428FV).

If the method being declared overrides another method, the FOR BIT DATA indication of each parameter must match exactly the FOR BIT DATA indication of the corresponding parameter of the overridden method. (SQLSTATE 428FV).

RETURNS
This mandatory clause identifies the method's result.

data-type3
Specifies the data type of the method's result. In this case, exactly the same considerations apply as for the parameters of methods specified in the description for data-type2.

AS LOCATOR
For LOB types or distinct types which are based on LOB types, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed from the method instead of the actual value.

An error is raised (SQLSTATE 42601) if AS LOCATOR is specified for a type other than a LOB or a distinct type based on a LOB.

If the method is FENCED, or if LANGUAGE is SQL, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

If the method being defined overrides another method, this clause cannot be specified (SQLSTATE 428FV).

If the method overrides another method, *data-type3* must be a subtype of the data type of the result of the overridden method if this data type is a structured type; otherwise both data types must be identical (SQLSTATE 428FV).

data-type4* CAST FROM *data-type5

Specifies the data type of the method's result.

This clause is used to return a different data type to the invoking statement from the data type returned by the method code. The *data-type5* must be castable to the *data-type4* parameter. If it is not castable, an error is returned (SQLSTATE 42880).

Because the length, precision, or scale for *data-type4* can be inferred from *data-type5*, it is not necessary (but still permitted) to specify the length, precision, or scale for parameterized types specified for *data-type4*. Instead, empty parentheses can be used, such as VARCHAR(), for example. FLOAT() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (REAL or DOUBLE).

A distinct type is not valid as the type specified in *data-type5* (SQLSTATE 42815). XML is not valid as the type specified in *data-type4* or *data-type5* (SQLSTATE 42815).

The cast operation is also subject to runtime checks that might result in conversion errors being returned.

AS LOCATOR

For LOB types or distinct types which are based on LOB types, the AS LOCATOR clause can be added. This indicates that a LOB locator is to be passed from the method instead of the actual value.

An error is raised (SQLSTATE 42601) if AS LOCATOR is specified for a type other than a LOB or a distinct type based on a LOB.

If the method is FENCED, or if LANGUAGE is SQL, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

If the method being defined overrides another method, this clause cannot be specified (SQLSTATE 428FV).

If the method being defined overrides another method, the FOR BIT DATA clause cannot be specified (SQLSTATE 428FV).

SPECIFIC *specific-name*

Provides a unique name for the instance of the method that is being defined. This specific name can be used when creating the method body or dropping the method. It can never be used to invoke the method. The unqualified form of *specific-name* is an SQL identifier (with a maximum length of 18). The qualified form is a schema-name followed by a period and an SQL identifier. The name, including the implicit or explicit qualifier, must not identify another specific method name that exists at the application server; otherwise an error is raised (SQLSTATE 42710).

The *specific-name* may be the same as an existing *method-name*.

If no qualifier is specified, the qualifier that was used for *type-name* is used. If a qualifier is specified, it must be the same as the explicit or implicit qualifier of *type-name* or an error is raised (SQLSTATE 42882).

If *specific-name* is not specified, a unique name is generated by the database manager. The unique name is SQL followed by a character timestamp, SQLyymmddhhmmssxxx.

SELF AS RESULT

Identifies this method as a type-preserving method, which is defined as follows:

- The declared return type must be the same as the declared subject-type (SQLSTATE 428EQ).

- When an SQL statement is compiled and resolves to a type preserving method, the static type of the result of the method is the same as the static type of the subject argument.
- The method must be implemented in such a way that the dynamic type of the result is the same as the dynamic type of the subject argument (SQLSTATE 2200G), and the result cannot be NULL (SQLSTATE 22004).

If the method being defined overrides another method, this clause cannot be specified (SQLSTATE 428FV).

SQL-routine-characteristics
Specifies the characteristics of the method body that will be defined for this type using CREATE METHOD.

LANGUAGE SQL
This clause is used to indicate that the method is written in SQL with a single RETURN statement. The method body is specified using the CREATE METHOD statement.

PARAMETER CCSID
Specifies the encoding scheme to use for all string data passed into and out of the SQL method. If the PARAMETER CCSID clause is not specified, the default is PARAMETER CCSID UNICODE for Unicode databases, and PARAMETER CCSID ASCII for all other databases.

ASCII
Specifies that string data is encoded in the database code page. If the database is a Unicode database, PARAMETER CCSID ASCII cannot be specified (SQLSTATE 56031).

UNICODE
Specifies that character data is in UTF-8, and that graphic data is in UCS-2. If the database is not a Unicode database, PARAMETER CCSID UNICODE cannot be specified (SQLSTATE 56031).

NOT DETERMINISTIC or DETERMINISTIC
This optional clause specifies whether the method always returns the same results for given argument values (DETERMINISTIC) or whether the method depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC method must always return the same result from successive invocations with identical inputs. Optimizations taking advantage of the fact that identical inputs always produce the same results are prevented by specifying NOT DETERMINISTIC. NOT DETERMINISTIC must be explicitly or implicitly specified if the body of the method accesses a special register, or calls another non-deterministic routine (SQLSTATE 428C2).

EXTERNAL ACTION or NO EXTERNAL ACTION
This optional clause specifies whether or not the method takes some action that changes the state of an object not managed by the database manager. Optimizations that assume methods have no external impacts are prevented by specifying EXTERNAL ACTION. For example: sending a message, ringing a bell, or writing a record to a file.

READS SQL DATA or CONTAINS SQL
Specifies the classification of SQL statements that the method can run. The database manager verifies that the SQL statements that the method issues are consistent with this specification.

For the classification of each statement, see "SQL statements that can be executed in routines and triggers" in Developing User-defined Routines (SQL and External).

Because the SQL statement supported is the RETURN statement, the distinction has to do with whether the expression is a subquery.

The default is READS SQL DATA.

READS SQL DATA
Specifies that the method can run statements with a data access classification of READS SQL DATA or CONTAINS SQL. The method cannot run SQL statements that modify data (SQLSTATE 42985). Nicknames cannot be referenced in the SQL statement (SQLSTATE 42997).

CONTAINS SQL
Specifies that the method can run only SQL statements with a data access classification of CONTAINS SQL. The method cannot run any SQL statements that read or modify data (SQLSTATE 42985).

CALLED ON NULL INPUT
This optional clause indicates that regardless of whether any arguments are null, the user-defined method is called. It can return a null value or a normal (non-null) value. However, responsibility for testing for null argument values lies with the method.

If the method being defined overrides another method, this clause cannot be specified (SQLSTATE 428FV).

NULL CALL can be used as a synonym for CALLED ON NULL INPUT.

INHERIT SPECIAL REGISTERS
This optional clause specifies that updatable special registers in the method will inherit their initial values from the environment of the invoking statement. For a method invoked in the select-statement of a cursor, the initial values are inherited from the environment in which the cursor is opened. For a routine invoked in a nested object (for example a trigger or view), the initial values are inherited from the runtime environment (not inherited from the object definition).

No changes to the special registers are passed back to the invoker of the function.

Non-updatable special registers, such as the datetime special registers, reflect a property of the statement currently executing, and are therefore set to their default values.

external-routine-characteristics

LANGUAGE
This mandatory clause is used to specify the language interface convention to which the user-defined method body is written.

C
This means the database manager will call the user-defined method as if it were a C function. The user-defined method must conform to the C language calling and linkage convention as defined by the standard ANSI C prototype.

JAVA
This means the database manager will call the user-defined method as a method in a Java class.

OLE
This means the database manager will call the user-defined method as if it were a method exposed by an OLE automation object. The method must conform with the OLE automation data types and invocation mechanism as described in the *OLE Automation Programmer's Reference*.

LANGUAGE OLE is only supported for user-defined methods stored in Windows 32-bit operating systems. THREADSAFE may not be specified for methods defined with LANGUAGE OLE (SQLSTATE 42613).

PARAMETER STYLE
This clause is used to specify the conventions used for passing parameters to and returning the value from methods.

DB2GENERAL
Used to specify the conventions for passing parameters to and returning the value from external methods that are defined as a method in a Java class. This can only be specified when LANGUAGE JAVA is used.

The value DB2GENRL may be used as a synonym for DB2GENERAL.

SQL
Used to specify the conventions for passing parameters to and returning the value from external methods that conform to C language calling and linkage conventions or methods

exposed by OLE automation objects. This must be specified when either LANGUAGE C or LANGUAGE OLE is used.

PARAMETER CCSID
Specifies the encoding scheme to use for all string data passed into and out of the external method. If the PARAMETER CCSID clause is not specified, the default is PARAMETER CCSID UNICODE for Unicode databases, and PARAMETER CCSID ASCII for all other databases.

ASCII
Specifies that string data is encoded in the database code page. If the database is a Unicode database, PARAMETER CCSID ASCII cannot be specified (SQLSTATE 56031).

UNICODE
Specifies that character data is in UTF-8, and that graphic data is in UCS-2. If the database is not a Unicode database, PARAMETER CCSID UNICODE cannot be specified (SQLSTATE 56031).

This clause cannot be specified with LANGUAGE OLE (SQLSTATE 42613).

DETERMINISTIC or NOT DETERMINISTIC
This optional clause specifies whether the method always returns the same results for given argument values (DETERMINISTIC) or whether the method depends on some state values that affect the results (NOT DETERMINISTIC). That is, a DETERMINISTIC method must always return the same result from successive invocations with identical inputs. Optimizations taking advantage of the fact that identical inputs always produce the same results are prevented by specifying NOT DETERMINISTIC. An example of a type that is non-deterministic is one that references special registers, global variables, or non-deterministic functions in a way that affects the result type.

FENCED or NOT FENCED
This clause specifies whether the method is considered "safe" to run in the database manager operating environment's process or address space (NOT FENCED), or not (FENCED).

If a method is registered as FENCED, the database manager protects its internal resources (data buffers, for example) from access by the method. Most methods will have the option of running as FENCED or NOT FENCED. In general, a method running as FENCED will not perform as well as a similar one running as NOT FENCED.

CAUTION: Use of NOT FENCED for methods that were not adequately coded, reviewed, and tested can compromise the integrity of your database. The database engine takes some precautions against many of the common types of inadvertent failures that might occur, but cannot guarantee complete integrity when NOT FENCED methods are used.

Only FENCED can be specified for a method with LANGUAGE OLE or NOT THREADSAFE (SQLSTATE 42613).

If the method is FENCED and has the NO SQL option, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

Either SYSADM authority, DBADM authority, or a special authority (CREATE_NOT_FENCED_ROUTINE) is required to register a method as NOT FENCED.

THREADSAFE or NOT THREADSAFE
Specifies whether the method is considered "safe" to run in the same process as other routines (THREADSAFE), or not (NOT THREADSAFE).

If the method is defined with LANGUAGE other than OLE:

- If the method is defined as THREADSAFE, the database manager can invoke the method in the same process as other routines. In general, to be threadsafe, a method should not use any global or static data areas. Most programming references include a discussion of writing threadsafe routines. Both FENCED and NOT FENCED methods can be THREADSAFE.
- If the method is defined as NOT THREADSAFE, the database manager will never invoke the method in the same process as another routine.

For FENCED methods, THREADSAFE is the default if the LANGUAGE is JAVA. For all other languages, NOT THREADSAFE is the default. If the method is defined with LANGUAGE OLE, THREADSAFE may not be specified (SQLSTATE 42613).

For NOT FENCED methods, THREADSAFE is the default. NOT THREADSAFE cannot be specified (SQLSTATE 42613).

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT
This optional clause may be used to avoid a call to the external method if any of the non-subject arguments is null.

If RETURNS NULL ON NULL INPUT is specified, and if at execution time any one of the method's arguments is null, the method is not called and the result is the null value.

If CALLED ON NULL INPUT is specified, then regardless of the number of null arguments, the method is called. It can return a null value or a normal (non-null) value. However, responsibility for testing for null argument values lies with the method.

The value NULL CALL may be used as a synonym for CALLED ON NULL INPUT for backwards and family compatibility. Similarly, NOT NULL CALL may be used as a synonym for RETURNS NULL ON NULL INPUT.

There are two cases in which this specification is ignored:

- If the subject argument is null, in which case the method is not executed and the result is null
- If the method is defined to have no parameters, in which case this null argument condition cannot occur.

READS SQL DATA, NO SQL, CONTAINS SQL
Specifies the classification of SQL statements that the method can run. The database manager verifies that the SQL statements that the method issues are consistent with this specification.

For the classification of each statement, see "SQL statements that can be executed in routines and triggers" in Developing User-defined Routines (SQL and External).

The default is READS SQL DATA.

READS SQL DATA
Specifies that the method can run statements with a data access classification of READS SQL DATA or CONTAINS SQL (SQLSTATE 38002 or 42985). The method cannot run SQL statements that modify data (SQLSTATE 38003 or 42985).

NO SQL
Specifies that the method can run only SQL statements with a data access classification of NO SQL (SQLSTATE 38001).

CONTAINS SQL
Specifies that the method can run only SQL statements with a data access classification of CONTAINS SQL (SQLSTATE 38004 or 42985). The method cannot run any SQL statements that read or modify data (SQLSTATE 38003 or 42985).

EXTERNAL ACTION or NO EXTERNAL ACTION
This optional clause specifies whether or not the method takes some action that changes the state of an object not managed by the database manager. Optimizations that assume methods have no external impacts are prevented by specifying EXTERNAL ACTION.

NO SCRATCHPAD or SCRATCHPAD *length*
This optional clause may be used to specify whether a scratchpad is to be provided for an external method. It is strongly recommended that methods be re-entrant, so a scratchpad provides a means for the method to "save state" from one call to the next.

If SCRATCHPAD is specified, then at the first invocation of the user-defined method, memory is allocated for a scratchpad to be used by the external method. This scratchpad has the following characteristics:

- *length*, if specified, sets the size in bytes of the scratchpad and must be between 1 and 32 767 (SQLSTATE 42820). The default value is 100.
- It is initialized to all X'00''s.
- Its scope is the SQL statement. There is one scratchpad per reference to the external method in the SQL statement.

So, if method X in the following statement is defined with the SCRATCHPAD keyword, three scratchpads would be assigned.

```
SELECT A, X..(A) FROM TABLEB
   WHERE X..(A) > 103 OR X..(A) < 19
```

If ALLOW PARALLEL is specified or defaulted to, then the scope is different from the one shown previously. If the method is executed on multiple database partitions, a scratchpad would be assigned on each database partition where the method is processed, for each reference to the method in the SQL statement. Similarly, if the query is executed with intrapartition parallelism enabled, more than three scratchpads may be assigned.

The scratchpad is persistent. Its content is preserved from one external method call to the next. Any changes made to the scratchpad by the external method on one call will be present on the next call. The database manager initializes scratchpads at the beginning of execution of each SQL statement. The database manager may reset scratchpads at the beginning of execution of each subquery. The system issues a final call before resetting a scratchpad if the FINAL CALL option is specified.

The scratchpad can be used as a central point for system resources (memory, for example) which the external method might acquire. The method could acquire the memory on the first call, keep its address in the scratchpad, and refer to it in subsequent calls.

In such a case where system resource is acquired, the FINAL CALL keyword should also be specified; this causes a special call to be made at end-of-statement to allow the external method to free any system resources acquired.

If SCRATCHPAD is specified, then on each invocation of the user-defined method, an additional argument is passed to the external method which addresses the scratchpad.

If NO SCRATCHPAD is specified, then no scratchpad is allocated or passed to the external method.

NO FINAL CALL or FINAL CALL

This optional clause specifies whether a final call is to be made to an external method. The purpose of such a final call is to enable the external method to free any system resources it has acquired. It can be useful in conjunction with the SCRATCHPAD keyword in situations where the external method acquires system resources such as memory and anchors them in the scratchpad.

If FINAL CALL is specified, then at execution time, an additional argument is passed to the external method which specifies the type of call. The types of calls are:

- Normal call: SQL arguments are passed and a result is expected to be returned.
- First call: the first call to the external method for this specific reference to the method in this specific SQL statement. The first call is a normal call.
- Final call: a final call to the external method to enable the method to free up resources. The final call is not a normal call. This final call occurs at the following times:
  - End-of-statement: this case occurs when the cursor is closed for cursor-oriented statements, or when the statement is through executing otherwise.
  - End-of-transaction: This case occurs when the normal end-of-statement does not occur. For example, the logic of an application may for some reason bypass the close of the cursor.

  If a commit operation occurs while a cursor defined as WITH HOLD is open, a final call is made at the subsequent close of the cursor or at the end of the application.

If NO FINAL CALL is specified, then no "call type" argument is passed to the external method, and no final call is made.

ALLOW PARALLEL or DISALLOW PARALLEL

This optional clause specifies whether, for a single reference to the method, the invocation of the method can be parallelized. In general, the invocations of most scalar methods should be parallelizable, but there may be methods (such as those depending on a single copy of a scratchpad) that cannot. If either ALLOW PARALLEL or DISALLOW PARALLEL are specified for a method, then this specification will be accepted.

The following questions should be considered in determining which keyword is appropriate for the method:

- Are all the method invocations completely independent of each other? If YES, then specify ALLOW PARALLEL.
- Does each method invocation update the scratchpad, providing value(s) that are of interest to the next invocation (the incrementing of a counter, for example)? If YES, then specify DISALLOW PARALLEL or accept the default.
- Is there some external action performed by the method which should happen only on one database partition? If YES, then specify DISALLOW PARALLEL or accept the default.
- Is the scratchpad used, but only so that some expensive initialization processing can be performed a minimal number of times? If YES, then specify ALLOW PARALLEL.

In any case, the body of every external method should be in a directory that is available on every database partition.

The syntax diagram indicates that the default value is ALLOW PARALLEL. However, the default is DISALLOW PARALLEL if one or more of the following options is specified in the statement:

- NOT DETERMINISTIC
- EXTERNAL ACTION
- SCRATCHPAD
- FINAL CALL

NO DBINFO or DBINFO

This optional clause specifies whether certain specific information known by the database manager will be passed to the method as an additional invocation-time argument (DBINFO), or not (NO DBINFO).NO DBINFO is the default. DBINFO is not supported for LANGUAGE OLE (SQLSTATE 42613). If the method being defined overrides another method, this clause cannot be specified (SQLSTATE 428FV).

If DBINFO is specified, a structure that contains the following information is passed to the method:

- Database name - the name of the currently connected database.
- Application ID - unique application ID which is established for each connection to the database.
- Application Authorization ID - the application runtime authorization ID, regardless of the nested methods in between this method and the application.
- Code page - identifies the database code page.
- Schema name - under the exact same conditions as for Table name, contains the name of the schema; otherwise blank.
- Table name - if and only if the method reference is either the right side of a SET clause in an UPDATE statement, or an item in the VALUES list of an INSERT statement, contains the unqualified name of the table being updated or inserted; otherwise blank.
- Column name - under the exact same conditions as for Table name, contains the name of the column being updated or inserted; otherwise blank.
- Database version/release - identifies the version, release and modification level of the database server invoking the method.
- Platform - contains the server's platform type.
- Table method result column numbers - not applicable to methods.

INHERIT SPECIAL REGISTERS

This optional clause specifies that special registers in the method will inherit their initial values from the calling statement. For cursors, the initial values are inherited from the time that the cursor is opened.

No changes to the special registers are passed back to the caller of the method.

Some special registers, such as the datetime special registers, reflect a property of the statement currently executing, and are therefore never inherited from the caller.

Notes

- Creating a structured type with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- A structured subtype defined with no attributes defines a subtype that inherits all its attributes from the supertype. If neither an UNDER clause nor any other attribute is specified, then the type is a root type of a type hierarchy without any attributes.
- The addition of a new subtype to a type hierarchy may cause packages to be invalidated. A package may be invalidated if it depends on a supertype of the new type. Such a dependency is the result of the use of a TYPE predicate or a TREAT specification.
- A structured type may have no more than 4082 attributes (SQLSTATE 54050).
- A method specification is not allowed to have the same signature as a function (comparing the first parameter-type of the function with the subject-type of the method).
- No original method may override another method, or be overridden by an original method (SQLSTATE 42745). Furthermore, a function and a method cannot be in an overriding relationship. This means that if the function were considered to be a method with its first parameter as subject S, it must not override another method in any supertype of S, and it must not be overridden by another method in any subtype of S (SQLSTATE 42745).
- Creation of a structured type automatically generates a set of functions and methods for use with the type. All the functions and methods are generated in the same schema as the structured type. If the signature of the generated function or method conflicts with or overrides the signature of an existing function in this schema, the statement fails (SQLSTATE 42710). The generated functions or methods cannot be dropped without dropping the structured type (SQLSTATE 42917). The following functions and methods are generated:
  - Functions
    - Reference Comparisons

      Six comparison functions with names =, <>, <, <=, >, >= are generated for the reference type REF(*type-name*). Each of these functions takes two parameters of type REF(*type-name*) and returns true, false, or unknown. The comparison operators for REF(*type-name*) are defined to have the same behavior as the comparison operators for the underlying data type of REF(*type-name*). (All references in a type hierarchy have the same reference representation type. This enables REF(S) and REF(T) to be compared, provided that S and T have a common supertype. Because uniqueness of the OID column is enforced only within a table hierarchy, it is possible that a value of REF(T) in one table hierarchy may be "equal" to a value of REF(T) in another table hierarchy, even though they reference different rows.)

      The scope of the reference type is not considered in the comparison.

    - Cast functions

      Two cast functions are generated to cast between the generated reference type REF(*type-name*) and the underlying data type of this reference type.

      - The name of the function to cast from the underlying type to the reference type is the implicit or explicit *funcname1*.

        The format of this function is:

```
CREATE FUNCTION funcname1 (rep-type)
  RETURNS REF(type-name) ...
```

- The name of the function to cast from the reference type to the underlying type of the reference type is the implicit or explicit *funcname2*.

  The format of this function is:

```
CREATE FUNCTION funcname2 ( REF(type-name) )
  RETURNS rep-type ...
```

For some rep-types, there are additional cast functions generated with *funcname1* to handle casting from constants.

- If *rep-type* is SMALLINT, the additional generated cast function has the format:

```
CREATE FUNCTION funcname1 (INTEGER)
  RETURNS REF(type-name)
```

- If *rep-type* is CHAR(n), the additional generated cast function has the format:

```
CREATE FUNCTION funcname1 ( VARCHAR(n))
  RETURNS REF(type-name)
```

- If *rep-type* is GRAPHIC(n), the additional generated cast function has the format:

```
CREATE FUNCTION funcname1 (VARGRAPHIC(n))
  RETURNS REF(type-name)
```

The schema name of the structured type must be included in the SQL path for successful use of these operators and cast functions in SQL statements.

- Constructor function

  The constructor function is generated to allow a new instance of the type to be constructed. This new instance will have null for all attributes of the type, including attributes that are inherited from a supertype.

  The format of the generated constructor function is:

```
CREATE FUNCTION type-name ( )
  RETURNS type-name
 ...
```

If NOT INSTANTIABLE is specified, no constructor function is generated.

– Methods

- Observer methods

  An observer method is defined for each attribute of the structured type. For each attribute, the observer method returns the type of the attribute. If the subject is null, the observer method returns a null value of the attribute type.

  For example, the attributes of an instance of the structured type ADDRESS can be observed using `C1..STREET`, `C1..CITY`, `C1..COUNTRY`, and `C1..CODE`.

  The method signature of the generated observer method is as if the following statement had been executed:

```
CREATE TYPE  type-name
   ...
  METHOD attribute-name()
    RETURNS attribute-type
```

  where *type-name* is the structured type name.

- Mutator methods

A type-preserving mutator method is defined for each attribute of the structured type. Use mutator methods to change attributes within an instance of a structured type. For each attribute, the mutator method returns a copy of the subject modified by assigning the argument to the named attribute of the copy.

For example, an instance of the structured type ADDRESS can be mutated using `C1..CODE('M3C1H7')`. If the subject is null, the mutator method raises an error (SQLSTATE 2202D).

The method signature of the generated mutator method is as if the following statement had been executed:

```
CREATE TYPE  type-name
        ...
     METHOD attribute-name (attribute-type)
        RETURNS type-name
```

If the attribute data type is SMALLINT, REAL, CHAR, or GRAPHIC, an additional mutator method is generated in order to support mutation using constants:

- If *attribute-type* is SMALLINT, the additional mutator supports an argument of type INTEGER.
- If *attribute-type* is REAL, the additional mutator supports an argument of type DOUBLE.
- If *attribute-type* is CHAR, the additional mutator supports an argument of type VARCHAR.
- If *attribute-type* is GRAPHIC, the additional mutator supports an argument of type VARGRAPHIC.

- If the structured type is used as a column type, the length of an instance of the type can be no more than 1 GB in length at runtime (SQLSTATE 54049).

- When creating a new subtype for an existing structured type (for use as a column type), any transform functions already written in support of existing related structured types should be re-examined and updated as necessary. Whether the new type is in the same hierarchy as a given type, or in the hierarchy of a nested type, it is likely that the existing transform function associated with this type will need to be modified to include some or all of the new attributes introduced by the new subtype. Generally speaking, because it is the set of transform functions associated with a given type (or type hierarchy) that enables UDF and client application access to the structured type, the transform functions should be written to support *all* of the attributes in a given composite hierarchy (that is, including the transitive closure of all subtypes and their nested structured types).

  When a new subtype of an existing type is created, all packages dependent on methods that are defined in supertypes of the type being created, and that are eligible for overriding, are invalidated.
- ***Table access restrictions***: If a method is defined as READS SQL DATA, no statement in the method can access a table that is being modified by the statement which invoked the method (SQLSTATE 57053). For example, suppose the method BONUS() is defined as READS SQL DATA. If the statement UPDATE DEPTINFO SET SALARY = SALARY + EMP..BONUS() is invoked, no SQL statement in the BONUS method can read from the EMPLOYEE table.
- ***Privileges***: The definer of the user-defined type always receives the EXECUTE privilege WITH GRANT OPTION on all methods and functions automatically generated for the structured type. The EXECUTE privilege is not granted on any methods explicitly specified in the CREATE TYPE statement until a method body is defined using the CREATE METHOD statement. The definer of the user-defined type does have the right to drop the method specification using the ALTER TYPE statement. EXECUTE privilege on all methods and functions automatically generated during the CREATE TYPE (structured) statement is granted to PUBLIC.

  When an external method is used in an SQL statement, the method definer must have the EXECUTE privilege on any packages used by the method or EXECUTEIN privilege on the schema containing the packages used by the method.
- In a partitioned database environment, the use of SQL in external user-defined functions or methods is not supported (SQLSTATE 42997).
- Only routines defined as NO SQL can be used to define an index extension (SQLSTATE 428F8).

- A Java routine defined as NOT FENCED will be invoked as if it had been defined as FENCED THREADSAFE.
- ***EXTERNAL ACTION methods***: If an EXTERNAL ACTION method is invoked in other than the outermost select list, the results are unpredictable since the number of times the method is invoked will vary depending on the access plan used.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - NOT VARIANT can be specified in place of DETERMINISTIC
  - VARIANT can be specified in place of NOT DETERMINISTIC
  - NULL CALL can be specified in place of CALLED ON NULL INPUT
  - NOT NULL CALL can be specified in place of RETURNS NULL ON NULL INPUT
  - PARAMETER STYLE DB2SQL can be specified in place of PARAMETER STYLE SQL

  The following syntax is accepted as the default behavior for external methods:
  - ASUTIME NO LIMIT
  - NO COLLID
  - PROGRAM TYPE SUB
  - STAY RESIDENT NO
  - CCSID UNICODE in a Unicode database
  - CCSID ASCII in a non-Unicode database if PARAMETER CCSID UNICODE is not specified

  The following syntax is accepted as the default behavior for SQL methods:
  - CCSID UNICODE in a Unicode database
  - CCSID ASCII in a non-Unicode database

Examples

- *Example 1:* Create a type for department.

```
CREATE TYPE DEPT AS
   (DEPT_NAME     VARCHAR(20),
      MAX_EMPS INT)
      REF USING INT
   MODE DB2SQL
```

- *Example 2:* Create a type hierarchy consisting of a type for employees and a subtype for managers.

```
CREATE TYPE EMP AS
  (NAME      VARCHAR(32),
  SERIALNUM INT,
  DEPT      REF(DEPT),
  SALARY    DECIMAL(10,2))
  MODE DB2SQL

CREATE TYPE MGR UNDER EMP AS
  (BONUS     DECIMAL(10,2))
  MODE DB2SQL
```

- *Example 3:* Create a type hierarchy for addresses. Addresses are intended to be used as types of columns. The inline length is not specified, so a default length is calculated. Encapsulate within the address type definition an external method that calculates how close this address is to a given input address. Create the method body using the CREATE METHOD statement.

```
CREATE TYPE address_t AS
  (STREET     VARCHAR(30),
  NUMBER     CHAR(15),
  CITY       VARCHAR(30),
  STATE      VARCHAR(10))
  NOT FINAL
```

```
    MODE DB2SQL
      METHOD SAMEZIP (addr address_t)
      RETURNS INTEGER
      LANGUAGE SQL
      DETERMINISTIC
      CONTAINS SQL
      NO EXTERNAL ACTION,

      METHOD DISTANCE (address_t)
      RETURNS FLOAT
      LANGUAGE C
      DETERMINISTIC
      PARAMETER STYLE SQL
      NO SQL
      NO EXTERNAL ACTION

  CREATE TYPE germany_addr_t UNDER address_t AS
    (FAMILY_NAME VARCHAR(30))
    NOT FINAL
    MODE DB2SQL

  CREATE TYPE us_addr_t UNDER address_t AS
    (ZIP VARCHAR(10))
    NOT FINAL
    MODE DB2SQL
```

- *Example 4:* Create a type that has nested structured type attributes.

```
  CREATE TYPE PROJECT AS
    (PROJ_NAME  VARCHAR(20),
     PROJ_ID    INTEGER,
     PROJ_MGR   MGR,
     PROJ_LEAD  EMP,
     LOCATION   ADDR_T,
     AVAIL_DATE DATE)
     MODE DB2SQL
```

CREATE TYPE MAPPING

The CREATE TYPE MAPPING statement defines a mapping between data types.

The mapping can be defined between the following data types:

- The data type of a column in a data source table or view that is going to be defined to a federated database
- A corresponding data type that is already defined to the federated database

The mapping can associate the federated database data type with a data type at:

- A specified data source
- A range of data sources; for example, all data sources of a particular type and version

A data type mapping must be created only if an existing one is not adequate.

If multiple type mappings are applicable when creating a nickname or creating a table (transparent DDL), the most recent one is applied.

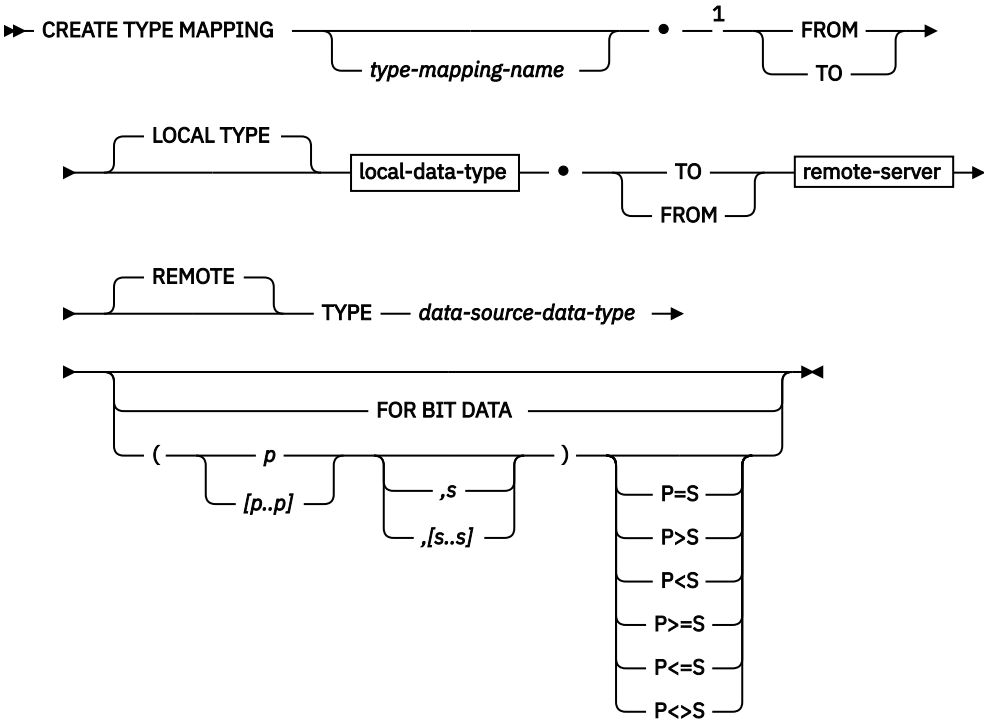
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

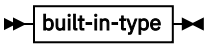
Authorization

The privileges held by the authorization ID of the statement must include DBADM authority.

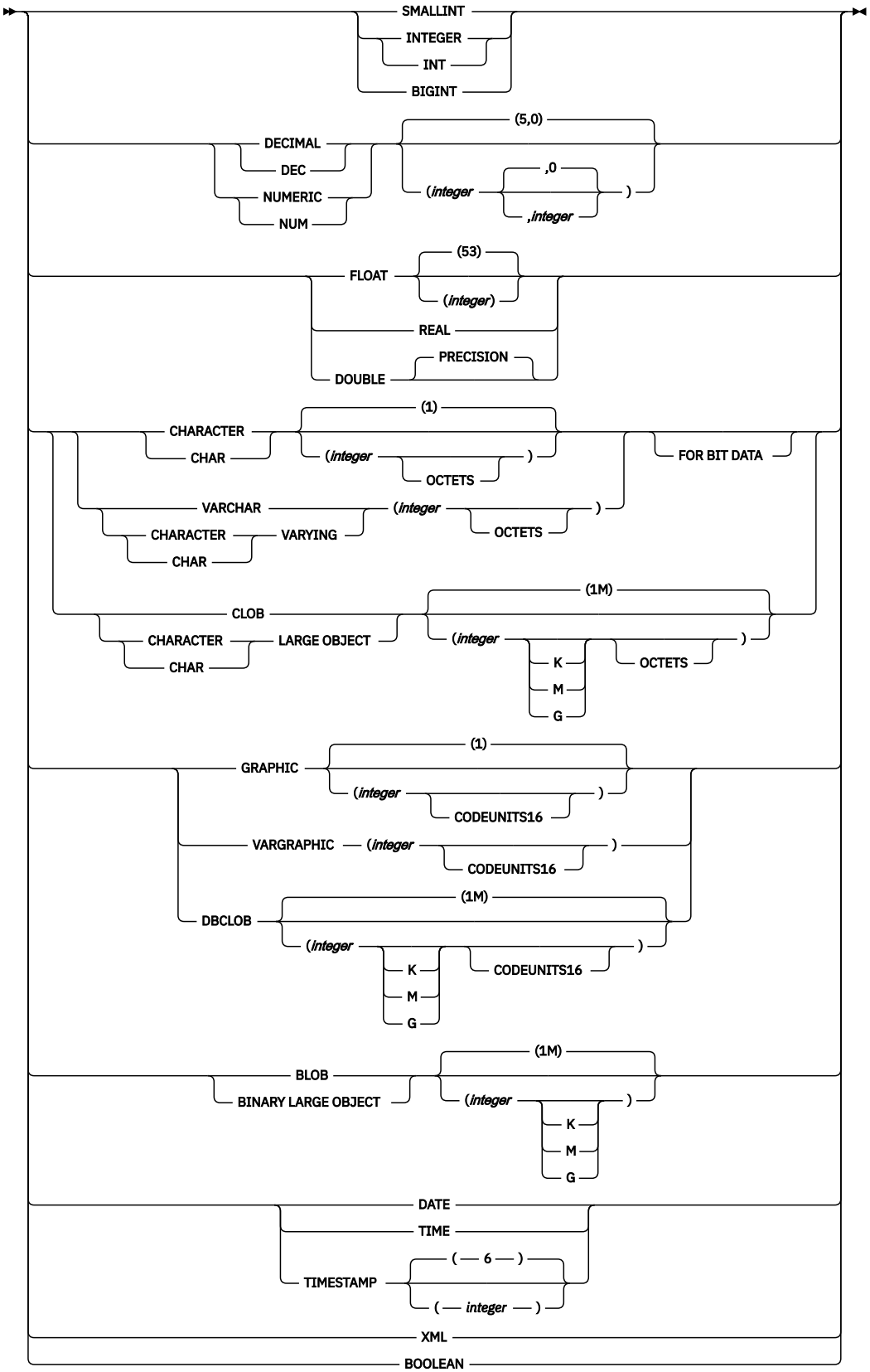
Syntax



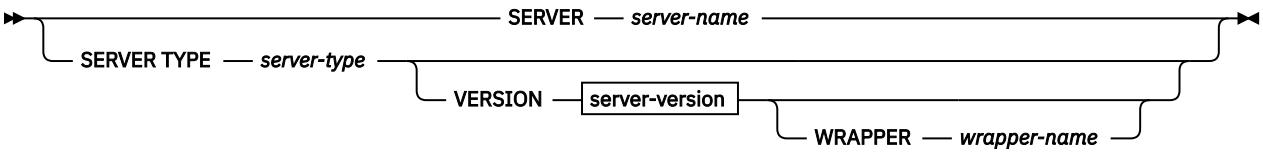
local-data-type



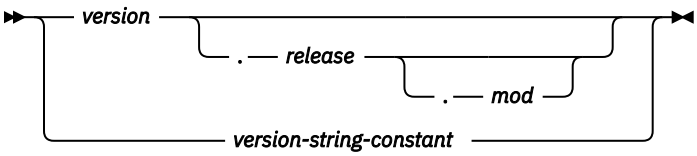
built-in-type



remote-server



server-version



Notes:

[1] Both a TO and a FROM keyword must be present in the CREATE TYPE MAPPING statement.

Description

type-mapping-name
Names the data type mapping. The name must not identify a data type mapping that is already described in the catalog. A unique name is generated if *type-mapping-name* is not specified.

FROM or TO
Specifies a reverse or forward type mapping.

FROM
Specifies a forward type mapping when followed by *local-data-type* or a reverse type mapping when followed by *remote-server*.

TO
Specifies a forward type mapping when followed by *remote-server* or a reverse type mapping when followed by *local-data-type*.

local-data-type
Identifies a data type that is defined to a federated database. If *local-data-type* is specified without a schema name, the type name is resolved by searching the schemas in the SQL path.

Empty parentheses can be used for the parameterized data types. A parameterized data type is any one of the data types that can be defined with a specific length, scale, or precision. If empty parentheses are specified in a forward type mapping, such as, for example, CHAR(), the length is determined from the column length on the remote table. If empty parentheses are specified in a reverse type mapping, the type mapping is applied to the data type with any length. If you omit parentheses altogether, the default length for the data type is used.

FLOAT() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (REAL or DOUBLE). NUMBER() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (DECFLOAT or DECIMAL).

DECFLOAT can be accepted only as the *local-data-type* by Oracle wrapper, Db2 wrapper for IBM Db2 Version 9.5 or later.

The *local-data-type* cannot be a user-defined type (SQLSTATE 42611).

built-in-type
See "CREATE TABLE" for the description of built-in data types.

SERVER *server-name*
Names the data source to which *data-source-data-type* is defined.

SERVER TYPE *server-type*
Identifies the type of data source to which *data-source-data-type* is defined.

VERSION
Identifies the version of the data source to which *data-source-data-type* is defined.

version
Specifies the version number. The value must be an integer.

release
Specifies the number of the release of the version denoted by *version*. The value must be an integer.

mod
Specifies the number of the modification of the release denoted by *release*. The value must be an integer.

version-string-constant
Specifies the complete designation of the version. The *version-string-constant* can be a single value (for example, '8i'); or it can be the concatenated values of *version*, *release* and, if applicable, *mod* (for example, '8.0.3').

WRAPPER *wrapper-name*
Specifies the name of the wrapper that the federated server uses to interact with data sources of the type and version denoted by *server-type* and *server-version*.

TYPE *data-source-data-type*
Specifies the data source data type that is being mapped to or from the local data type.

Empty parentheses can be used for the parameterized data types. If empty parentheses are specified in a forward type mapping, such as, for example, CHAR(), the type mapping is applied to the data type with any length. If empty parentheses are specified in a reverse type mapping, the length is determined from the column length specified in the transparent DDL. If you omit parentheses altogether, the default length for the data type is used.

The *data-source-data-type* must be a built-in data type. User-defined types are not allowed.

If *server-name* is specified with a type mapping, or existing servers are affected by the type mapping, *data-source-data-type*, *p*, and *s* are verified when creating the type mapping (SQLSTATE 42611).

p
If *p* is specified, only the data type whose length or precision equals *p* is affected by the type mapping.

[p1..p2]
For forward type mapping only. For a decimal data type, *p1* and *p2* specify the minimum and maximum number of digits that a value can have. For string data types, *p1* and *p2* specify the minimum and maximum number of characters that a value can have. In all cases, the maximum must equal or exceed the minimum; and both numbers must be valid with respect to the data type.

s
If *s* is specified, only the data type whose scale equals *s* is affected by the type mapping.

[s1..s2]
For forward type mapping only. For a decimal data type, *s1* and *s2* specify the minimum and maximum number of digits allowed to the right of the decimal point. The maximum must equal or exceed the minimum, and both numbers must be valid with respect to the data type.

P [operand] S
For a decimal data type, P *[operand]* S specifies a comparison between the precision and the number of digits allowed to the right of the decimal point. For example, the operand = indicates that the type mapping is applied if the precision and the number of digits allowed in the decimal fraction are the same.

FOR BIT DATA
Indicates whether *data-source-data-type* is for bit data. These keywords are required if the data source type column contains binary values. The database manager will determine this attribute if it is not specified for a character data type.

Notes

- A CREATE TYPE MAPPING statement within a given unit of work (UOW) cannot be processed (SQLSTATE 55007) under either of the following conditions:
  - The statement references a single data source, and the UOW already includes one of the following:
    - A SELECT statement that references a nickname for a table or view within this data source
    - An open cursor on a nickname for a table or view within this data source
    - Either an INSERT, DELETE, or UPDATE statement issued against a nickname for a table or view within this data source
  - The statement references a category of data sources (for example, all data sources of a specific type and version), and the UOW already includes one of the following:
    - A SELECT statement that references a nickname for a table or view within one of these data sources
    - An open cursor on a nickname for a table or view within one of these data sources
    - Either an INSERT, DELETE, or UPDATE statement issued against a nickname for a table or view within one of these data sources
- When multiple type mappings are applicable, the most recent one will be used. You can retrieve the creation time for a type mapping by querying the CREATE_TIME column of the SYSCAT.TYPEMAPPINGS catalog view.
- BINARY and VARBINARY types are not supported in a Federated system.

Examples

1. Create a forward type mapping between the Oracle data type DATE and the data type SYSIBM.DATE. For all of the nicknames that are created after this mapping is defined, Oracle columns of data type DATE will map to Db2 columns of data type DATE.

```
CREATE TYPE MAPPING MY_ORACLE_DATE
  FROM LOCAL TYPE SYSIBM.DATE
  TO SERVER TYPE ORACLE
  REMOTE TYPE DATE
```

2. Create a forward type mapping between data type SYSIBM.DECIMAL(10,2) and the Oracle data type NUMBER([10..38],2) at data source ORACLE1. If there is a column in the Oracle table of data type NUMBER(11,2), it will be mapped to a column of data type DECIMAL(10,2), because 11 is between 10 and 38.

```
CREATE TYPE MAPPING MY_ORACLE_DEC
  FROM LOCAL TYPE SYSIBM.DECIMAL(10,2)
  TO SERVER ORACLE1
  REMOTE TYPE NUMBER([10..38],2)
```

3. Create a forward type mapping between data type SYSIBM.VARCHAR(*p*) and the Oracle data type CHAR(*p*) at data source ORACLE1 (*p* is any length). If there is a column in the Oracle table of data type CHAR(10), it will be mapped to a column of data type VARCHAR(10).

```
CREATE TYPE MAPPING MY_ORACLE_CHAR
  FROM LOCAL TYPE SYSIBM.VARCHAR()
  TO SERVER ORACLE1
  REMOTE TYPE CHAR()
```

4. Create a reverse type mapping between the Oracle data type NUMBER(10,2) at data source ORACLE2 and data type SYSIBM.DECIMAL(10,2). If you use transparent DDL to create an Oracle table and specify a column of data type DECIMAL(10,2), the Oracle table will be created with a column of data type NUMBER(10,2).

```
CREATE TYPE MAPPING MY_ORACLE_DEC
  TO LOCAL TYPE SYSIBM.DECIMAL(10,2)
  FROM SERVER ORACLE2
  REMOTE TYPE NUMBER(10,2)
```

CREATE USAGE LIST

The CREATE USAGE LIST statement defines a usage list. A usage list is a database object for monitoring all unique sections (DML statements) that have referenced a particular table or index during their execution.

Invocation

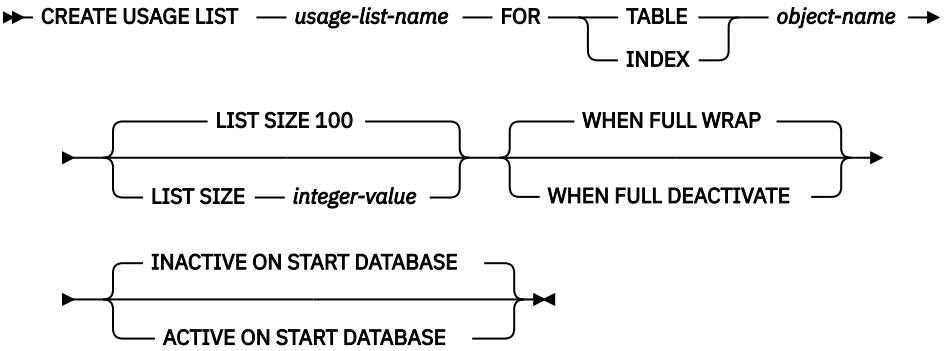
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include one of the following privileges:

- DBADM authority
- SQLADM authority

Syntax



Description

usage-list-name
Names the usage list. The *usage-list-name*, including the implicit or explicit qualifier, must not identify a usage list that is described in the catalog (SQLSTATE 42710). If the usage list is explicitly qualified with a schema name, the schema name must not begin with the characters 'SYS' (SQLSTATE 42939).

TABLE *object-name*
Designates the table for which the usage list is defined. The *object-name*, including the implicit or explicit qualifier, must specify a table defined in the catalog (SQLSTATE 42704). The name must not specify an alias, catalog table, created temporary table, hierarchy table, detached table, nickname, typed table, or view (SQLSTATE 42809). If the table is explicitly qualified with a schema name, the schema name must not begin with the characters 'SYS' (SQLSTATE 42939).

INDEX *object-name*
Designates the index for which the usage list is defined. The *object-name*, including the implicit or explicit qualifier, must specify an index defined in the catalog (SQLSTATE 42704). Indexes defined on tables other than untyped tables or materialized query tables are not supported (SQLSTATE 42809). The name must specify a physical index; Block Indexes (BLOK), Clustering indexes (CLUS), Dimension block indexes (DIM), Regular indexes (REG), and Physical indexes over XML column (XVIP). All other index types are not supported (SQLSTATE 42809). If the index is explicitly qualified with a schema name, the schema name must not begin with the characters 'SYS' (SQLSTATE 42939).

LIST SIZE *integer-value*
Specifies that the size of this list is *integer-value* entries. The minimum size that can be specified is 10 and the maximum is 5000 (SQLSTATE 428B7). The default size is 100 entries.

WHEN FULL
Specifies what action is performed when an active usage list becomes full. The default is to wrap when the list becomes full.

WRAP
Specifies that the usage list wraps and replaces the oldest entries.

DEACTIVATE
Specifies that the usage list deactivates.

INACTIVE ON START DATABASE
Specifies that the usage list is not activated for monitoring whenever the database is activated. Collection must be explicitly started using the SET USAGE LIST statement. This clause is the default.

ACTIVE ON START DATABASE
Specifies that the usage list is automatically activated for monitoring whenever the database is activated.

Notes

- ***Tracking sections with unique keys:*** A usage list keep tracks of all unique sections (DML statements only) that have referenced a particular object. References are aggregated within the list with the unique key of executable ID, representing the section doing the reference, and the monitor interval ID at the time of the reference. Each list entry keeps a count of section executions related to that entry and a set of statistics outlining the affect that the section had on the object across those executions.
- ***Usage list release time:*** A usage list is set to released when the CREATE USAGE LIST statement is committed.
- ***Memory allocation:*** Memory is allocated the first time that the object for which the usage list is defined is referenced by a section.
- ***Memory allocation in a partitioned database environment or Db2 pureScale environment***: If the state of a usage list for a partitioned table or index is set to active, memory is allocated for each data partition when the data partition is first referenced by the section. Similarly, in a partitioned database environment or Db2 pureScale environment, memory is allocated at each active member. If a member is unavailable at the time of activation, then the memory is allocated when the member is next activated (if the state of the usage list is still set to active). This also applies when a member is added to the cluster.
- ***State of the usage list when specifying WHEN FULL DEACTIVATE:*** If the usage list was created with the clause WHEN FULL DEACTIVATE, then the state of the usage list at each member is set to inactive independently. Similarly, for partitioned tables and indexes, the state of the usage list for each data partition is set to inactive independently.
- ***Implicit reactivation of an active usage list:*** If the state of an INACTIVE ON START DATABASE usage list is set to active in a partitioned database environment or Db2 pureScale environment, then its behavior is similar to the ACTIVE ON START DATABASE clause until the state of the usage list is explicitly set to inactive or the instance is recycled. That is, if the state of a usage list is active when a database member is deactivated or offline, and that database member is subsequently reactivated, the usage list for this member is also implicitly reactivated.
- ***Inactive usage lists remain inactive upon database member reactivation:*** If the state of an ACTIVE ON START DATABASE usage list is set to inactive in a partitioned database environment or Db2 pureScale environment, then its behavior is similar to the INACTIVE ON START DATABASE clause until the state of the usage list is explicitly set to active or the instance is recycled. That is, if the state of a usage list is inactive when a database member is deactivated or offline, and that database member is subsequently reactivated, the state of the usage list for this member will remain inactive.
- ***Multiple usage lists:*** Multiple usage lists can be created for the same table or index, however, it is recommended that only one of them be activated. Activating all of them affects database performance and memory usage.
- ***Activating and deactivating usage lists:*** See the Notes section for the SET USAGE LIST STATE statement regarding activation and deactivation of the usage list.

- ***Usage list size considerations:*** When the state of a usage list is set to active, the memory for the usage list is allocated from the monitor heap. At the maximum list size setting, the usage list is approximately 2MB. For partitioned tables or indexes, memory is allocated for each data partition. For example, if a partitioned table has three data partitions defined, approximately 6MB of memory is allocated. Therefore, activating multiple usage lists imposes more memory requirements on the monitor heap. It is therefore suggested that a reasonable list size is selected, or that you set the **mon_heap_sz** configuration parameter to AUTOMATIC so that the database manager manages the monitor heap size.
- ***Performance considerations:*** To maintain high performance, create usage lists such that they are limited to the amount required to gather the information you need. Each usage list requires system memory; system performance can degrade as additional usage lists are activated.

Examples

- *Example 1*: Create a usage list USL_ACC for table SAYYID.ACCOUNTS with a default list size of 100 entries.

```
CREATE USAGE LIST USL_ACC FOR TABLE SAYYID.ACCOUNTS
```

- *Example 2*: Create a usage list USL_SHOPPING_IND for index BIRD.SHOPPINGIND with a list of 50 entries that wraps when the list becomes full.

```
CREATE USAGE LIST USL_SHOPPING_IND FOR INDEX BIRD.SHOPPINGIND
   LISTSIZE 50
   WHEN FULL WRAP
```

- *Example 3*: Create a usage list USL_PAYROLL for table MIKE.PAYROLL with a list size of 200 entries which will deactivate when the list becomes full and will automatically start collecting whenever the database is activated.

```
CREATE USAGE LIST USL_PAYROLL FOR TABLE MIKE.PAYROLL
   LISTSIZE 200
   WHEN FULL DEACTIVATE
   ACTIVE ON START DATABASE
```

- *Example 4*: Create a usage list USL_EMP for partitioned table JACOBO.EMPLOYEES with a list size of 500 entries which will deactivate when the list becomes full.

```
CREATE USAGE LIST USL_EMP FOR TABLE JACOBO.EMPLOYEES
   LIST SIZE 500
   WHEN FULL DEACTIVATE
```

  When the usage list is activated for monitoring, then a list of 500 entries will be allocated for each data partition.

- *Example 5*: Create a usage list USL_PARTS for table SHAKTI.PARTS with a list size of 20 entries that will be activated manually on database activation and will wrap when it becomes full.

```
CREATE USAGE LIST USL_PARTS FOR TABLE SHAKTI.PARTS
   LIST SIZE 20
   INACTIVE ON START DATABASE
   WHEN FULL WRAP
```

CREATE USER MAPPING

The CREATE USER MAPPING statement defines a mapping between an authorization ID that uses a federated database and the authorization ID and password to use at a specified data source.

Invocation

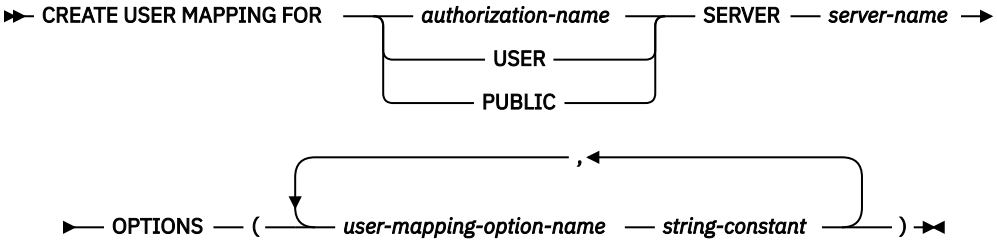
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

If the authorization ID of the statement is different from the authorization name that is being mapped to the data source, the privileges held by the authorization ID of the statement must include DBADM authority. Otherwise, if the authorization ID and the authorization name match, no authorities or privileges are required.

When creating a public user mapping, the privileges held by the authorization ID of the statement must include DBADM authority.

Syntax



Description

authorization-name
Specifies the authorization name under which a user or application connects to a federated database. The *authorization_name* is mapped to the REMOTE_AUTHID user mapping option.

USER
The value in the USER special register. When USER is specified, the authorization ID issuing the CREATE USER MAPPING statement is mapped to the REMOTE_AUTHID user mapping option.

PUBLIC
Specifies that any valid authorization ID for the local federated database will be mapped to the data source authorization ID that is specified in the REMOTE_AUTHID user option.

SERVER *server-name*
Names the server object for the data source that the *authorization-name* can access. The *server-name* is the local name for the remote server that is registered with the federated database.

OPTIONS
Specify configuration options for the user mapping to be created. Which options you can specify depends on the data source of the object for which a user mapping is being created. For a list of data sources and the user mapping options that apply to each, see Data source options. Each option value is a character string constant that must be enclosed in single quotation marks.

Notes

- User mappings are required only for the following data sources: the Db2 family of products, Documentum, Informix, Microsoft SQL Server, ODBC, Oracle, Sybase, and Teradata.
- The REMOTE_PASSWORD option is always required for a user mapping.
- Public user mappings and non-public user mappings cannot coexist on the same federated server. This means that if you have created public user mappings, you will not be able to create non-public user mappings on the same federated server. The reverse is also true, if you have created non-public user mappings, you will not be able to create public user mappings on the same federated server.
- **Syntax alternatives:** The following syntax is supported for compatibility with previous versions of Db2:
  - ADD can be specified before *user-mapping-option-name string-constant*.

Example

```
CREATE USER MAPPING FOR <db2inst1>
   SERVER <server_name>
   OPTIONS (
      REMOTE_AUTHID '<admin>',
      REMOTE_PASSWORD '<password>);
```

where

- *db2inst1* specifies the local authorization ID in the Db2 instance. You should use the keyword USER or PUBLIC, or the Db2 instance name. USER is for current Db2 user, PUBLIC is for all Db2 users.
- *server_name* specifies the server definition name that you defined in the **CREATE SERVER** statement for the JDBC data source. The user mapping is paired with the server statement.
- *admin* specifies the remote user ID for the remote data source (for example, MySQL). The value is case-sensitive unless you set the **FOLD_ID** server parameter to "U" or "L" in the **CREATE SERVER** statement.
- *password* specifies the remote password for the remote data source (for example, MySQL). The value is case-sensitive unless you set the **FOLD_PW** server option to "U" or "L" in the CREATE SERVER statement.

CREATE VARIABLE

The CREATE VARIABLE statement defines a session global variable.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the variable does not exist
- CREATEIN privilege on the schema, if the schema name of the variable refers to an existing schema
- SCHEMAADM authority on the schema, if the schema name of the variable refers to an existing schema
- DBADM authority

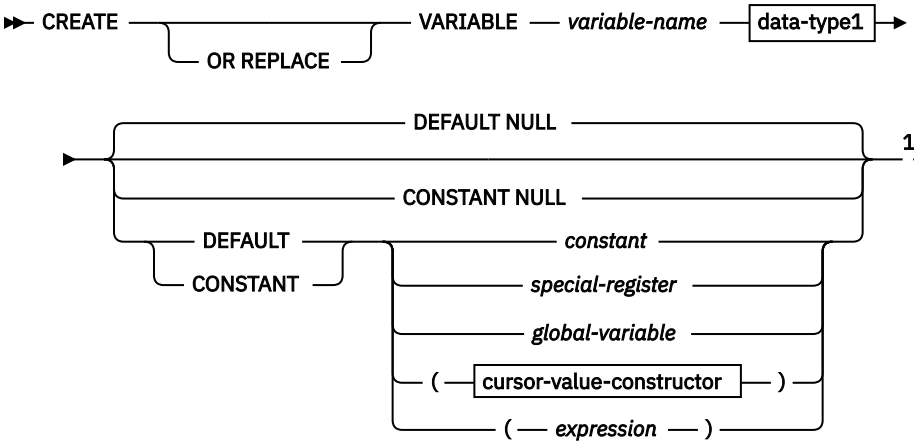
and any privileges that are necessary to execute the default expression.

To execute this statement with a *cursor-value-constructor* that uses a *select-statement*, the privileges held by the authorization ID of the statement must include the privileges necessary to execute the *select-statement*. See the Authorization section in "SQL queries".

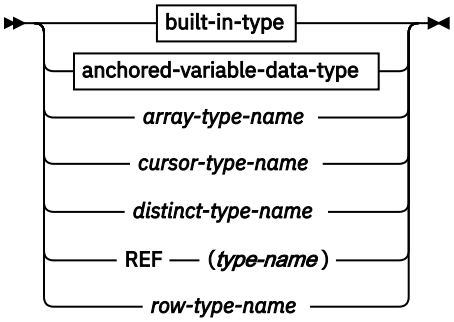
Group privileges are not considered when checking authorization for objects referenced in the statement

To replace an existing variable, the authorization ID of the statement must be the owner of the existing variable (SQLSTATE 42501).

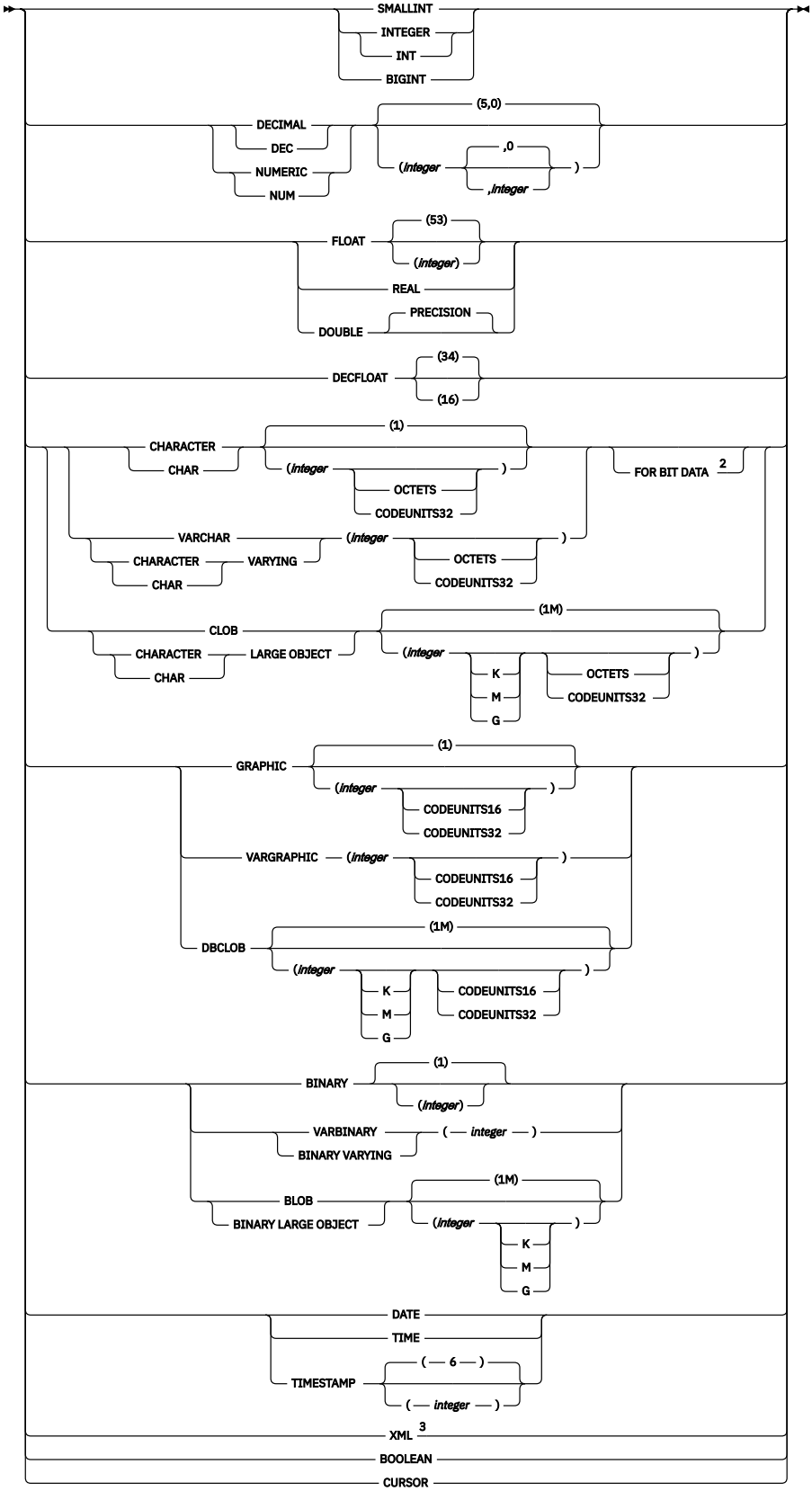
Syntax



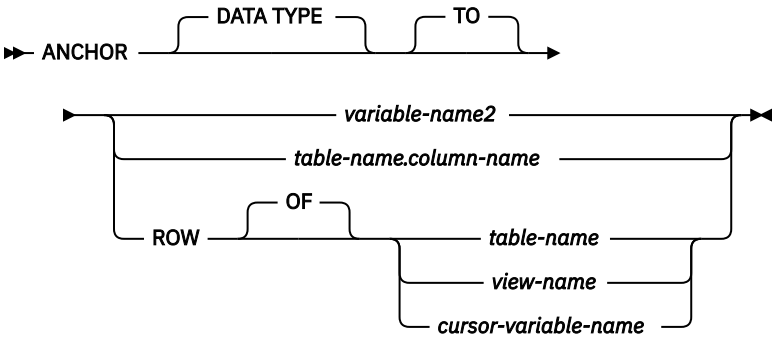
data-type1



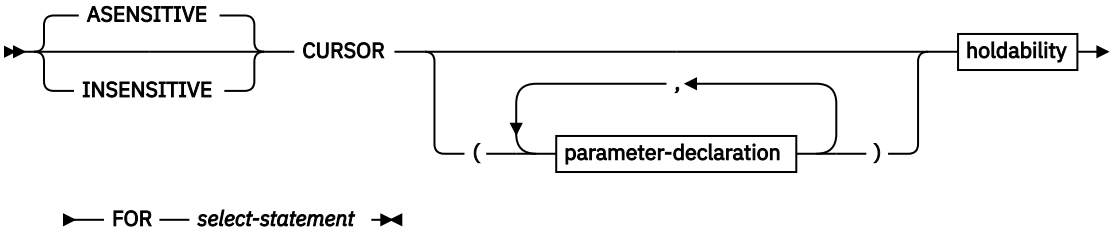
built-in-type



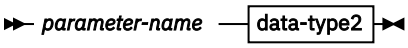
anchored-variable-data-type



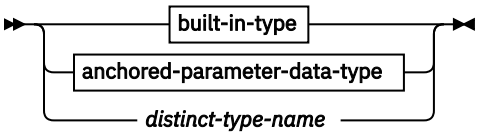
cursor-value-constructor



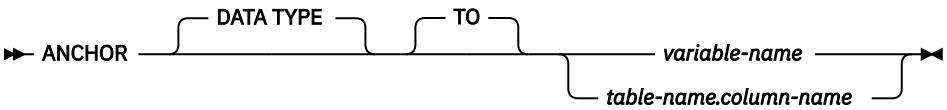
parameter-declaration



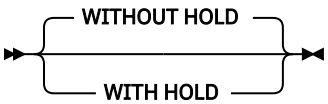
data-type2



anchored-parameter-data-type



holdability



Notes:

[1] If *data-type1* specifies a CURSOR built-in type or *cursor-type-name*, only NULL or *cursor-value-constructor* can be specified. Only DEFAULT NULL can be explicitly specified for *array-type-name* or *row-type-name*.

[2] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

[3] For version 10.1, you can use the XML data type only as a parameter data type in a cursor value constructor. For version 10.1 Fix Pack 1 or later fix pack releases, you can also use the XML data type to create global variables.

Description

OR REPLACE

Specifies to replace the definition for the variable if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog, with the exception that privileges that were granted on the variable are not affected. This option is ignored

if a definition for the variable does not exist at the current server. This option can be specified only by the owner of the object.

variable-name
Names the global variable. The name, including an implicit or explicit qualifier, must not identify a global variable that already exists at the current server (SQLSTATE 42710). If a qualifier is not specified, the current schema is implicitly assigned. If the global variable name is explicitly qualified with a schema name, the schema name must not begin with the characters "SYS" (SQLSTATE 42939).

data-type1
Specifies the data type of the global variable. A structured type cannot be specified (SQLSTATE 42611).

built-in-type
Specifies a built-in data type. BOOLEAN and CURSOR cannot be specified for a table. For version 10.1, an XML data type cannot be specified (SQLSTATE 42611). The XML data type support starts in version 10.1 Fix Pack 1. For a more complete description of each built-in data type, see "CREATE TABLE".

FOR BIT DATA can be specified as part of character string data types.

BOOLEAN
For a Boolean.

CURSOR
For a reference to an underlying cursor.

anchored-variable-data-type
Identifies another object used to determine the data type of the global variable. The data type of the anchor object has the same limitations that apply to specifying the data type directly, or in the case of a row, to creating a row type.

ANCHOR DATA TYPE TO
Indicates an anchored data type is used to specify the data type.

variable-name2
Identifies a global variable. The data type of the referenced variable is used as the data type for the global variable.

table-name.column-name
Identifies a column name of an existing table or view. The data type of the column is used as the data type for the global variable.

ROW OF *table-name* or *view-name*
Specifies that the global variable is a row of fields with names and data types that are based on the column names and column data types of the table identified by *table-name* or the view identified by *view-name*. The data type of the global variable is an unnamed row type.

ROW OF *cursor-variable-name*
Specifies a row of fields with names and data types that are based on the field names and field data types of the cursor variable identified by *cursor-variable-name*. The specified cursor variable must be one of the following elements (SQLSTATE 428HS):

- A global variable with a strongly typed cursor data type
- A global variable with a weakly typed cursor data type that was created or declared with a CONSTANT clause specifying a select-statement where all the result columns are named.

If the cursor type of the cursor variable is not strongly-typed using a named row type, the data type of the global variable is an unnamed row type.

array-type-name
Specifies the name of a user-defined array type. If *array-type-name* is specified without a schema name, the array type is resolved by searching the schemas in the SQL path.

cursor-type-name
Specifies the name of a cursor type. If *cursor-type-name* is specified without a schema name, the cursor type is resolved by searching the schemas in the SQL path.

distinct-type-name
Specifies the name of a distinct type. The length, precision, and scale of the declared variable are, respectively, the length, precision, and scale of the source type of the distinct type. If *distinct-type-name* is specified without a schema name, the distinct type is resolved by searching the schemas in the SQL path.

REF (*type-name*)
Specifies a reference type. If a type name is specified without a schema name, the *type-name* is resolved by searching the schemas in the SQL path.

row-type-name
Specifies the name of a user-defined row type. The fields of the variable are the fields of the row type. If *row-type-name* is specified without a schema name, the row type is resolved by searching the schemas in the SQL path.

DEFAULT or CONSTANT
Specifies a value for the global variable when it is first referenced. The DEFAULT or CONSTANT clause value is determined on this first reference. If neither is specified, the default for the global variable is the null value. Only DEFAULT NULL can be explicitly specified if *array-type-name* or *row-type-name* is specified.

DEFAULT
Defines the default for the global variable. The default value must be assignment-compatible with the data type of the variable.

CONSTANT
Specifies that the global variable has a fixed value that cannot be changed. A global variable that is defined using CONSTANT cannot be used as the target of any assignment operation. The fixed value must be assignment-compatible with the data type of the variable.

NULL
Specifies NULL as the default for the global variable. If *row-type-name* is specified, the value for the global variable is a row where each field has the null value.

constant
Specifies the value of a constant as the default for the global variable. If *data-type1* specifies a CURSOR built-in type or *cursor-type-name*, *constant* cannot be specified (SQLSTATE 42601).

special-register
Specifies the value of a special register as the default for the global variable. If *data-type1* specifies a CURSOR built-in type or *cursor-type-name*, *special-register* cannot be specified (SQLSTATE 42601).

global-variable
Specifies the value of a global variable as the default for the global variable. If *data-type1* specifies a CURSOR built-in type or *cursor-type-name*, *global-variable* cannot be specified (SQLSTATE 42601).

cursor-value-constructor
A *cursor-value-constructor* specifies the *select-statement* that is associated with the global variable. The assignment of a *cursor-value-constructor* to a cursor variable defines the underlying cursor of that cursor variable.

ASENSITIVE or INSENSITIVE
Specifies whether the cursor is asensitive or insensitive to changes. See "DECLARE CURSOR" for more information. The default is ASENSITIVE.

ASENSITIVE
Specifies that the cursor should be as sensitive as possible to insert, update, or delete operations made to the rows underlying the result table, depending on how the *select-statement* is optimized. This option is the default.

INSENSITIVE
Specifies that the cursor does not have sensitivity to insert, update, or delete operations that are made to the rows underlying the result table. If INSENSITIVE is specified, the cursor is read-only and the result table is materialized when the cursor is opened. As a result, the size of the result table, the order of the rows, and the values for each row do not change after the cursor is opened. The SELECT statement cannot contain a FOR UPDATE clause, and the cursor cannot be used for positioned updates or deletes.

(parameter-declaration, ...)
Specifies the input parameters of the cursor, including the name and the data type of each parameter.

parameter-name
Names the parameter for use as an SQL variable within *select-statement*. The name cannot be the same as any other parameter name for the cursor. Names should also be chosen to avoid any column names that could be used in *select-statement*, since column names are resolved before parameter names.

data-type2
Specifies the data type of the cursor parameter used within *select-statement*.

built-in-type
Specifies a built-in data type. For a more complete description of each built-in data type, see "CREATE TABLE". The BOOLEAN and CURSOR built-in types cannot be specified (SQLSTATE 429BB).

anchored-parameter-data-type
Identifies another object used to determine the data type of the cursor parameter. The data type of the anchor object is bound by the same limitations that apply when specifying the data type directly.

ANCHOR DATA TYPE TO
Indicates an anchored data type is used to specify the data type.

variable-name
Identifies a global variable. The data type of the referenced variable is used as the data type for the cursor parameter.

table-name.column-name
Identifies a column name of an existing table or view. The data type of the column is used as the data type for the cursor parameter.

distinct-type-name
Specifies the name of a distinct type. If *distinct-type-name* is specified without a schema name, the distinct type is resolved by searching the schemas in the SQL path.

holdability
Specifies whether the cursor is prevented from being closed as a consequence of a commit operation. See "DECLARE CURSOR" for more information. The default is WITHOUT HOLD.

WITHOUT HOLD
Does not prevent the cursor from being closed as a consequence of a commit operation.

WITH HOLD
Maintains resources across multiple units of work. Prevents the cursor from being closed as a consequence of a commit operation.

select-statement
Specifies the SELECT statement of the cursor. See "select-statement" for more information.

statement-name
Specifies the prepared *select-statement* of the cursor. See "PREPARE" for an explanation of prepared statements. The target cursor variable must not have a data type that is a strongly-typed user-defined cursor type (SQLSTATE 428HU).

expression
Specifies the value of an expression as the default for the global variable. The expression can be any expression of the type described in "Expressions". The expression must be assignment-compatible with the data type of the variable. The maximum size of the expression is 64K. The default expression must not modify SQL data (SQLSTATE 428FL) or perform external action (SQLSTATE 42845). If *data-type1* specifies a CURSOR built-in type or *cursor-type-name*, *expression* cannot be specified (SQLSTATE 42601).

Rules

- ***Use of anchored data types:*** An anchored data type cannot refer to the following objects (SQLSTATE 428HS): a nickname, typed table, typed view, statistical view that is associated with an expression-based index, declared temporary table, row definition that is associated with a weakly typed cursor, object with a code page or collation that is different from the database code page or database collation.

Notes

- Session global variables have a session scope. This means that, although they are available to all sessions that are active on the database, their value is private for each session.
- ***Contexts for array, Boolean, cursor, and row global variables:*** Global variables that are array variables, Boolean variables, or row variables can only be used in compound SQL (compiled) statements or SET variable statements. Global variables that are cursor variables can only be used in compound SQL (compiled) statements.
- ***Create with errors:*** If an object referenced in the default expression does not exist or is marked invalid, or the definer temporarily doesn't have privileges to access the object, and if the database configuration parameter **auto_reval** is not set to DISABLED, then the variable will still be created successfully. The variable will be marked invalid and will be revalidated the next time it is invoked.
- ***Scope of global variable values:*** The values for session global variables persist until they are updated in the current session, the global variable is dropped or altered, or the application session ends. The value is unaffected by COMMIT or ROLLBACK statements. The default value for a global variable can be not deterministic and dependent on when the default value is calculated for the global variable (for example, a reference to the time of day, or a reference to some data stored in a table).

  A technique commonly used, especially for performance, is for an application or product to manage a set of connections and route transactions to an arbitrary connection. In these situations, the non-default value of a global variable or the not deterministic initial default value for a global variable should only be relied on until the end of the transaction. Examples of where this type of situation can occur include applications that: use XA protocols, use connection pooling, use the connection concentrator, and use HADR to achieve failover.
- ***Privileges to use a global variable:*** An attempt to read from or to write to a global variable created by this statement requires that the authorization ID attempting this action hold the appropriate privilege on the global variable. The definer of the variable is implicitly granted all privileges on the variable.
- ***Setting of the default value:*** A created global variable is instantiated to its default value when it is first referenced within its given scope. Note that if a global variable is referenced in a statement, it is instantiated independently of the control flow for that statement.
- ***Using a newly created session global variable:*** If a global variable is created within a session, it cannot be used by other sessions until the unit of work has committed. However, the new global variable can be used within the session that created the variable before the unit of work commits.

Examples

- *Example 1:* Create a session global variable to indicate what printer to use for the session.

```
CREATE VARIABLE MYSCHEMA.MYJOB_PRINTER VARCHAR(30)
  DEFAULT 'Default printer'
```

- *Example 2:* Create a session global variable to indicate the department where an employee works.

```
CREATE VARIABLE SCHEMA1.GV_DEPTNO INTEGER
  DEFAULT ((SELECT DEPTNO FROM HR.EMPLOYEES
    WHERE EMPUSER = SESSION_USER))
```

- *Example 3:* Create a session global variable to indicate the security level of the current user.

```
CREATE VARIABLE SCHEMA2.GV_SECURITY_LEVEL INTEGER
  DEFAULT (GET_SECURITY_LEVEL (SESSION_USER))
```

- *Example 4:* Create a session global variable as a cursor on the STAFF table that returns the names of each employee for the specified job type. Order the results by the department number.

```
CREATE VARIABLE STAFFJOBS CURSOR
  CONSTANT (CURSOR (WHICHJOB CHAR(5))
  FOR SELECT NAME, DEPT FROM STAFF WHERE JOB = WHICHJOB
    ORDER BY DEPT)
```

- *Example 5:* Create a global variable of the XML data type:

```
CREATE VARIABLE MYSCHEMA.CUSTOMER_HISTORY_VAR XML
```

CREATE VIEW

The CREATE VIEW statement defines a view on one or more tables, views or nicknames.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the view does not exist
- CREATEIN privilege on the schema, if the schema name of the view refers to an existing schema
- SCHEMAADM authority on the schema, if the schema name of the view refers to an existing schema
- DBADM authority

and at least one of the following authorities for each table, view, or nickname identified in any fullselect:

- CONTROL privilege on that table, view, or nickname
- SELECT privilege on that table, view, or nickname
- SELECTIN privilege on the schema containing the table, view, or nickname
- DATAACCESS authority on the schema containing the table, view, or nickname
- DATAACCESS authority

If creating a subview:

- The authorization ID of the statement must be the same as the definer of the root table of the table hierarchy, or
- The privileges held by the authorization ID must include SCHEMAADM authority on the schema containing the root table of the table hierarchy
- The privileges held by the authorization ID must include DBADM authority

and

- The authorization ID of the statement must have SELECT WITH GRANT privilege on the underlying table of the subview, or the superview must not have SELECT privilege granted to any user other than the view definer, or
- ACCESSCTRL authority on the database or ACCESSCTRL authority on the schema containing the underlying table of the subview, and one of the following authorities:
  - SELECT privilege on the underlying table of the subview
  - SELECTIN privilege on the schema containing the underlying table of the subview
  - DATAACCESS authority on the schema containing the underlying table of the subview
  - DATAACCESS authority

If WITH ROW MOVEMENT is specified, the privileges held by the authorization ID of the statement must include at least one of the following authorities:

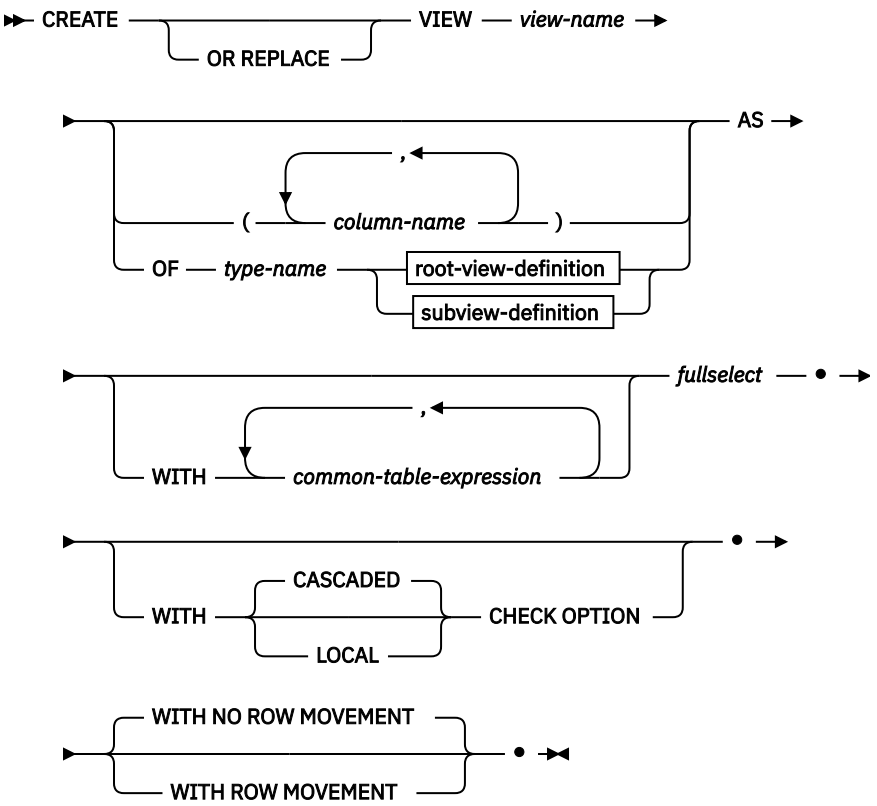
- UPDATE privilege on that table or view
- UPDATEIN privilege on the schema containing that table or view
- DATAACCESS authority on the schema containing that table or view
- DATAACCESS authority

Group privileges are not considered for any table or view specified in the CREATE VIEW statement.

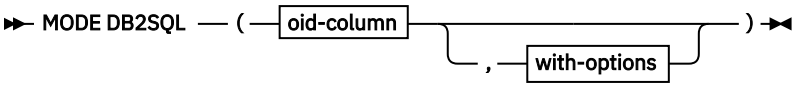
Privileges are not considered when defining a view on a federated database nickname. Authorization requirements of the data source for the table or view referenced by the nickname are applied when the query is processed. The authorization ID of the statement can be mapped to a different remote authorization ID.

To replace an existing view, the authorization ID of the statement must be the owner of the existing view (SQLSTATE 42501).

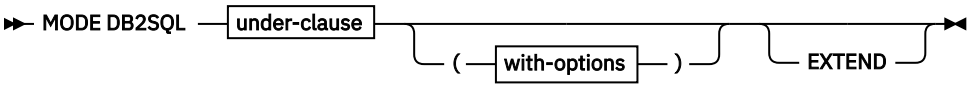
Syntax



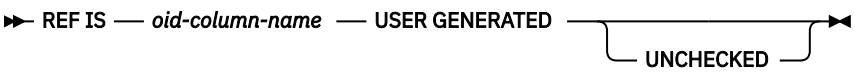
root-view-definition



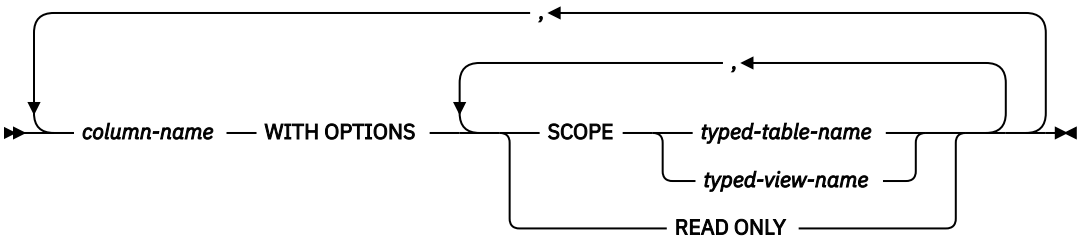
subview-definition



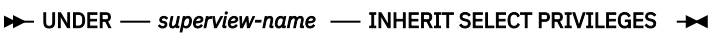
oid-column



with-options



under-clause



Description

OR REPLACE
Specifies to replace the definition for the view if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog, with the exception that privileges that were granted on the view are not affected. This option is ignored if a definition for the view does not exist at the current server. This option can be specified only by the owner of the object.

view-name
Names the view. The name, including the implicit or explicit qualifier, must not identify a table, view, nickname or alias described in the catalog. The qualifier must not be SYSIBM, SYSCAT, SYSFUN, or SYSSTAT (SQLSTATE 42939).

The name can be the same as the name of an inoperative view (see Inoperative views). In this case the new view specified in the CREATE VIEW statement will replace the inoperative view. The user will get a warning (SQLSTATE 01595) when an inoperative view is replaced. No warning is returned if the application was bound with the bind option SQLWARN set to NO.

column-name
Names the columns in the view. If a list of column names is specified, it must consist of as many names as there are columns in the result table of the fullselect. Each *column-name* must be unique and unqualified. If a list of column names is not specified, the columns of the view inherit the names of the columns of the result table of the fullselect.

A list of column names must be specified if the result table of the fullselect has duplicate column names or an unnamed column (SQLSTATE 42908). An unnamed column is a column derived from a constant, function, expression, or set operation that is not named using the AS clause of the select list.

OF *type-name*
Specifies that the columns of the view are based on the attributes of the structured type identified by *type-name*. If *type-name* is specified without a schema name, the type name is resolved by searching the schemas on the SQL path (defined by the FUNCPATH preprocessing option for static SQL and by the CURRENT PATH register for dynamic SQL). The type name must be the name of an existing

user-defined type (SQLSTATE 42704) and it must be a structured type that is instantiable (SQLSTATE 428DP).

MODE DB2SQL
This clause is used to specify the mode of the typed view. This is the only valid mode currently supported.

UNDER *superview-name*
Indicates that the view is a subview of *superview-name*. The superview must be an existing view (SQLSTATE 42704) and the view must be defined using a structured type that is the immediate supertype of *type-name* (SQLSTATE 428DB). The schema name of *view-name* and *superview-name* must be the same (SQLSTATE 428DQ). The view identified by *superview-name* must not have any existing subview already defined using *type-name* (SQLSTATE 42742).

The columns of the view include the object identifier column of the superview with its type modified to be REF(*type-name*), followed by columns based on the attributes of *type-name* (remember that the type includes the attributes of its supertype).

INHERIT SELECT PRIVILEGES
Any user or group holding a SELECT privilege on the superview will be granted an equivalent privilege on the newly created subview. The subview definer is considered to be the grantor of this privilege.

OID-column
Defines the object identifier column for the typed view.

REF IS *OID-column-name* USER GENERATED
Specifies that an object identifier (OID) column is defined in the view as the first column. An OID is required for the root view of a view hierarchy (SQLSTATE 428DX). The view must be a typed view (the OF clause must be present) that is not a subview (SQLSTATE 42613). The name for the column is defined as *OID-column-name* and cannot be the same as the name of any attribute of the structured type *type-name* (SQLSTATE 42711). The first column specified in *fullselect* must be of type REF(*type-name*) (you may need to cast it so that it has the appropriate type). If UNCHECKED is not specified, it must be based on a not nullable column on which uniqueness is enforced through an index (primary key, unique constraint, unique index, or OID-column). This column will be referred to as the *object identifier column* or *OID column*. The keywords USER GENERATED indicate that the initial value for the OID column must be provided by the user when inserting a row. Once a row is inserted, the OID column cannot be updated (SQLSTATE 42808).

UNCHECKED
Defines the object identifier column of the typed view definition to assume uniqueness even though the system can not prove this uniqueness. This is intended for use with tables or views that are being defined into a typed view hierarchy where the user knows that the data conforms to this uniqueness rule but it does not comply with the rules that allow the system to prove uniqueness. UNCHECKED option is mandatory for view hierarchies that range over multiple hierarchies or legacy tables or views By specifying UNCHECKED, the user takes responsibility for ensuring that each row of the view has a unique OID. If the user fails to ensure this property, and a view contains duplicate OID values, then a path-expression or DEREF operator involving one of the non-unique OID values may result in an error (SQLSTATE 21000).

with-options
Defines additional options that apply to columns of a typed view.

***column-name* WITH OPTIONS**
Specifies the name of the column for which additional options are specified. The *column-name* must correspond to the name of an attribute defined in (not inherited by) the *type-name* of the view. The column must be a reference type (SQLSTATE 42842). It cannot correspond to a column that also exists in the superview (SQLSTATE 428DJ). A column name can only appear in one WITH OPTIONS SCOPE clause in the statement (SQLSTATE 42613).

SCOPE
Identifies the scope of the reference type column. A scope must be specified for any column that is intended to be used as the left operand of a dereference operator or as the argument of the DEREF function.

Specifying the scope for a reference type column may be deferred to a subsequent ALTER VIEW statement (if the scope is not inherited) to allow the target table or view to be defined, usually in the case of mutually referencing views and tables. If no scope is specified for a reference type column of the view and the underlying table or view column was scoped, then the underlying column's scope is inherited by the reference type column. The column remains unscoped if the underlying table or view column did not have a scope. See "Notes" on page 1546 for more information about scope and reference type columns.

typed-table-name
The name of a typed table. The table must already exist or be the same as the name of the table being created (SQLSTATE 42704). The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-table-name* (SQLSTATE 428DM). No checking is done of any existing values in *column-name* to ensure that the values actually reference existing rows in *typed-table-name*.

typed-view-name
The name of a typed view. The view must already exist or be the same as the name of the view being created (SQLSTATE 42704). The data type of *column-name* must be REF(*S*), where *S* is the type of *typed-view-name* (SQLSTATE 428DM). No checking is done of any existing values in *column-name* to ensure that the values actually reference existing rows in *typed-view-name*.

READ ONLY
Identifies the column as a read-only column. This option is used to force a column to be read-only so that subview definitions can specify an expression for the same column that is implicitly read-only.

AS
Identifies the view definition.

WITH *common-table-expression*
Defines a common table expression for use with the fullselect that follows. A common table expression cannot be specified when defining a typed view.

fullselect
Defines the view. At any time, the view consists of the rows that would result if the SELECT statement were executed. The data type of the columns of the view cannot be a distinct type with data type constraints, array type, cursor type, or row type. The fullselect must not reference host variables, parameter markers, or declared temporary tables. However, a parameterized view can be created as an SQL table function.

The fullselect cannot include an SQL data change statement in the FROM clause (SQLSTATE 428FL).

If a view is created by using a 'SELECT *' statement, the view is not updated when a new column is added to the base table.

For Typed Views and Subviews: The *fullselect* must conform to the following rules otherwise an error is returned (SQLSTATE 428EA unless otherwise specified).

- The fullselect must not include references to the DBPARTITIONNUM or HASHEDVALUE functions, non-deterministic functions, or functions defined to have external action.
- The body of the view must consist of a single subselect, or a UNION ALL of two or more subselects. Let each of the subselects participating directly in the view body be called a *branch* of the view. A view may have one or more branches.
- The FROM-clause of each branch must consist of a single table or view (not necessarily typed), called the *underlying* table or view of that branch.
- The underlying table or view of each branch must be in a separate hierarchy (that is, a view cannot have multiple branches with their underlying tables or views in the same hierarchy).
- None of the branches of a typed view definition may specify GROUP BY or HAVING.
- If the view body contains UNION ALL, the root view in the hierarchy must specify the UNCHECKED option for its OID column.

For a hierarchy of views and subviews: Let BR1 and BR2 be any branches that appear in the definitions of views in the hierarchy. Let T1 be the underlying table or view of BR1, and let T2 be the underlying table or view of BR2. Then:

- If T1 and T2 are not in the same hierarchy, then the root view in the view hierarchy must specify the UNCHECKED option for its OID column.
- If T1 and T2 are in the same hierarchy, then BR1 and BR2 must contain predicates or ONLY-clauses that are sufficient to guarantee that their row-sets are disjoint.

For typed subviews defined using EXTEND AS: For every branch in the body of the subview:

- The underlying table of each branch must be a (not necessarily proper) subtable of some underlying table of the immediate superview.
- The expressions in the SELECT list must be assignable to the non-inherited columns of the subview (SQLSTATE 42854).

For typed subviews defined using AS without EXTEND:

- For every branch in the body of the subview, the expressions in the SELECT-list must be assignable to the declared types of the inherited and non-inherited columns of the subview (SQLSTATE 42854).
- The OID-expression of each branch over a given hierarchy in the subview must be equivalent (except for casting) to the OID-expression in the branch over the same hierarchy in the root view.
- The expression for a column not defined (implicitly or explicitly) as READ ONLY in a superview must be equivalent in all branches over the same underlying hierarchy in its subviews.

WITH CHECK OPTION

Specifies the constraint that every row that is inserted or updated through the view must conform to the definition of the view. A row that does not conform to the definition of the view is a row that does not satisfy the search conditions of the view.

WITH CHECK OPTION must not be specified if any of the following conditions is true:

- The view is read-only (SQLSTATE 42813). If WITH CHECK OPTION is specified for an updatable view that does not allow inserts, the constraint applies to updates only.
- The view references the DBPARTITIONNUM or HASHEDVALUE function, a non-deterministic function, or a function with external action (SQLSTATE 42997).
- A nickname is the update target of the view.
- A view that has an INSTEAD OF trigger defined on it is the update target of the view (SQLSTATE 428FQ).

If WITH CHECK OPTION is omitted, the definition of the view is not used in the checking of any insert or update operations that use the view. Some checking might still occur during insert or update operations if the view is directly or indirectly dependent on another view that includes WITH CHECK OPTION. Because the definition of the view is not used, rows might be inserted or updated through the view that do not conform to the definition of the view.

CASCADED

The WITH CASCADED CHECK OPTION constraint on a view *V* means that *V* inherits the search conditions as constraints from any updatable view on which *V* is dependent. Furthermore, every updatable view that is dependent on *V* is also subject to these constraints. Thus, the search conditions of *V* and each view on which *V* is dependent are ANDed together to form a constraint that is applied for an insert or update of *V* or of any view dependent on *V*.

LOCAL

The WITH LOCAL CHECK OPTION constraint on a view *V* means the search condition of *V* is applied as a constraint for an insert or update of *V* or of any view that is dependent on *V*.

The difference between CASCADED and LOCAL is shown in the following example. Consider the following updatable views (substituting for Y from column headings of the table that follows):

```
V1 defined on table T
V2 defined on V1 WITH Y CHECK OPTION
```

```
V3 defined on V2
V4 defined on V3 WITH Y CHECK OPTION
V5 defined on V4
```

The following table shows the search conditions against which inserted or updated rows are checked:

	Y is LOCAL	**Y is CASCADED**
V1 checked against:	no view	no view
V2 checked against:	V2	V2, V1
V3 checked against:	V2	V2, V1
V4 checked against:	V2, V4	V4, V3, V2, V1
V5 checked against:	V2, V4	V4, V3, V2, V1

Consider the following updatable view which shows the impact of the WITH CHECK OPTION using the default CASCADED option:

```
CREATE VIEW V1 AS SELECT COL1 FROM T1 WHERE COL1 > 10

CREATE VIEW V2 AS SELECT COL1 FROM V1 WITH CHECK OPTION

CREATE VIEW V3 AS SELECT COL1 FROM V2 WHERE COL1 < 100
```

The following INSERT statement using *V1* will succeed because *V1* does not have a WITH CHECK OPTION and *V1* is not dependent on any other view that has a WITH CHECK OPTION.

```
INSERT INTO V1 VALUES(5)
```

The following INSERT statement using *V2* will result in an error because *V2* has a WITH CHECK OPTION and the insert would produce a row that did not conform to the definition of *V2*.

```
INSERT INTO V2 VALUES(5)
```

The following INSERT statement using *V3* will result in an error even though it does not have WITH CHECK OPTION because *V3* is dependent on *V2* which does have a WITH CHECK OPTION (SQLSTATE 44000).

```
INSERT INTO V3 VALUES(5)
```

The following INSERT statement using *V3* will succeed even though it does not conform to the definition of *V3* (*V3* does not have a WITH CHECK OPTION); it does conform to the definition of *V2* which does have a WITH CHECK OPTION.

```
INSERT INTO V3 VALUES(200)
```

WITH NO ROW MOVEMENT or WITH ROW MOVEMENT

Specifies the action to take for an updatable UNION ALL view when a row is updated in a way that violates a check constraint on the underlying table. The default is WITH NO ROW MOVEMENT.

WITH NO ROW MOVEMENT

Specifies that an error (SQLSTATE 23513) is to be returned if a row is updated in a way that violates a check constraint on the underlying table.

WITH ROW MOVEMENT

Specifies that an updated row is to be moved to the appropriate underlying table, even if it violates a check constraint on that table.

Row movement involves deletion of the rows that violate the check constraint, and insertion of those rows back into the view. The WITH ROW MOVEMENT clause can only be specified for UNION ALL views whose columns are all updatable (SQLSTATE 429BJ). If a row is inserted (perhaps after trigger activation) into the same underlying table from which it was deleted, an error is returned (SQLSTATE 23524). A view defined using the WITH ROW MOVEMENT clause must not

contain nested UNION ALL operations, except in the outermost fullselect (SQLSTATE 429BJ). A view defined using the WITH ROW MOVEMENT clause, cannot contain any references to a system-period temporal table, application-period temporal table, or bitemporal table.

Notes

- Creating a view with a schema name that does not already exist will result in the implicit creation of that schema provided the authorization ID of the statement has IMPLICIT_SCHEMA authority. The schema owner is SYSIBM. The CREATEIN privilege on the schema is granted to PUBLIC.
- View columns inherit the NOT NULL WITH DEFAULT attribute from the base table or view except when columns are derived from an expression. When a row is inserted or updated into an updatable view, it is checked against the constraints (primary key, referential integrity, and check) if any are defined on the base table.
- A new view cannot be created if it uses an inoperative view in its definition. (SQLSTATE 51024).
- If an object referenced in the view body does not exist or is marked invalid, or the definer temporarily doesn't have privileges to access the object, and if the database configuration parameter **auto_reval** is set to DEFERRED_FORCE, then the view will still be created successfully. The view will be marked invalid and will be revalidated the next time it is referenced.
- This statement does not support declared temporary tables (SQLSTATE 42995).
- ***Views based on column-organized tables:***
  - Creating a typed view on column-organized tables is not supported.
  - The WITH CHECK OPTION clause cannot be specified if a column-organized table is part of the view definition.
- ***Deletable views:*** A view is *deletable* if an INSTEAD OF trigger for the delete operation has been defined for the view, or if all of the following conditions are true:
  - Each FROM clause of the outer fullselect identifies only one base table (with no OUTER clause), deletable view (with no OUTER clause), deletable nested table expression, or deletable common table expression (cannot identify a nickname). Also, any period-specification specified for the base table or deletable view does not reference the SYSTEM_TIME period.
  - The outer fullselect does not include a VALUES clause
  - The outer fullselect does not include a GROUP BY clause or HAVING clause
  - The outer fullselect does not include aggregate functions in the select list
  - The outer fullselect does not include SET operations (UNION, EXCEPT or INTERSECT) with the exception of UNION ALL
  - The base tables in the operands of a UNION ALL must not be the same table and each operand must be deletable
  - The select list of the outer fullselect does not include DISTINCT
- ***Updatable views:*** A column of a view is *updatable* if an INSTEAD OF trigger for the update operation has been defined for the view, or if all of the following conditions are true:
  - The view is deletable (independent of an INSTEAD OF trigger for delete), the column resolves to a column of a base table (not using a dereference operation), and the READ ONLY option is not specified
  - All the corresponding columns of the operands of a UNION ALL have exactly matching data types (including length or precision and scale) and matching default values if the fullselect of the view includes a UNION ALL

  A view is updatable if *any* column of the view is updatable.
- ***Insertable views***: A view is insertable if an INSTEAD OF trigger for the insert operation has been defined for the view, or at least one column of the view is updatable (independent of an INSTEAD OF trigger for update), and the fullselect of the view does not include UNION ALL.

A given row can be inserted into a view (including a UNION ALL) if, and only if, it fulfills the check constraints of exactly one of the underlying base tables.

To insert into a view that includes non-updatable columns, those columns must be omitted from the column list.

- ***Read-only views***: A view is *read-only* if it is *not* deletable, updatable, or insertable.

  The READONLY column in the SYSCAT.VIEWS catalog view indicates if a view is read-only without considering period specifications or INSTEAD OF triggers.
- Common table expressions and nested table expressions follow the same set of rules for determining whether they are deletable, updatable, insertable, or read-only.
- ***Special registers for temporal support:*** The values of the CURRENT TEMPORAL SYSTEM_TIME and CURRENT TEMPORAL BUSINESS_TIME special registers have no impact on the query expression that defines a view while it is being defined. When a view is used in an SQL statement, the values of the CURRENT TEMPORAL SYSTEM_TIME and CURRENT TEMPORAL BUSINESS_TIME special registers for the session processing the SQL statement are applied to the view.
- ***Inoperative views:*** An *inoperative view* is a view that is no longer available for SQL statements. A view becomes inoperative if:
  - A privilege, upon which the view definition is dependent, is revoked.
  - An object such as a table, nickname, alias or function, upon which the view definition is dependent, is dropped.
  - A view, upon which the view definition is dependent, becomes inoperative.
  - A view that is the superview of the view definition (the subview) becomes inoperative.

  In practical terms, an inoperative view is one in which the view definition has been unintentionally dropped. For example, when an alias is dropped, any view defined using that alias is made inoperative. All dependent views also become inoperative and packages dependent on the view are no longer valid.

  Until the inoperative view is explicitly re-created or dropped, a statement using that inoperative view cannot be compiled (SQLSTATE 51024) with the exception of the CREATE ALIAS, CREATE VIEW, DROP VIEW, and COMMENT ON TABLE statements. Until the inoperative view has been explicitly dropped, its qualified name cannot be used to create another table or alias (SQLSTATE 42710).

  An inoperative view may be re-created by issuing a CREATE VIEW statement using the definition text of the inoperative view. This view definition text is stored in the TEXT column of the SYSCAT.VIEWS catalog. When recreating an inoperative view, it is necessary to explicitly grant any privileges required on that view by others, due to the fact that all authorization records on a view are deleted if the view is marked inoperative. Note that there is no need to explicitly drop the inoperative view in order to re-create it. Issuing a CREATE VIEW statement with the same *view-name* as an inoperative view will cause that inoperative view to be replaced, and the CREATE VIEW statement will return a warning (SQLSTATE 01595).

  Inoperative views are indicated by an X in the VALID column of the SYSCAT.VIEWS catalog view and an X in the STATUS column of the SYSCAT.TABLES catalog view.
- ***Privileges***: The definer of a view always receives the SELECT privilege on the view as well as the right to drop the view. The definer of a view will get CONTROL privilege on the view only if the definer has CONTROL privilege on every base table, view, or nickname identified in the fullselect, or if the definer has each of the following authorities:
  - ACCESSCTRL or SECADM on the database or ACCESSCTRL on the schema containing every base table, view, or nickname identified in the fullselect
  - DATAACCESS on the database or DATAACCESS on the schema containing every base table, view, or nickname identified in the fullselect
  - DBADM or SCHEMAADM on the schema containing every base table, view, or nickname identified in the fullselect

The definer of the view is granted INSERT, UPDATE, column level UPDATE or DELETE privileges on the view if the view is not read-only and the definer has the corresponding privileges on the underlying objects.

For a view defined WITH ROW MOVEMENT, the definer acquires the UPDATE privilege on the view only if the definer has the UPDATE privilege on all columns of the view, as well as INSERT and DELETE privileges on all underlying tables or views.

The definer of a view only acquires privileges if the privileges from which they are derived exist at the time the view is created. The definer must have these privileges either directly or because PUBLIC has these privilege. Privileges are not considered when defining a view on a federated server nickname. However, when using a view on a nickname, the user's authorization ID must have valid select privileges on the table or view that the nickname references at the data source. Otherwise, an error is returned. Privileges held by groups of which the view definer is a member, are not considered.

When a subview is created, the SELECT privileges held on the immediate superview are automatically granted on the subview.

- ***Scope and REF columns***: When selecting a reference type column in the fullselect of a view definition, consider the target type and scope that is required.
  - If the required target type and scope is the same as the underlying table or view, the column can simply be selected.
  - If the scope needs to be changed, use the WITH OPTIONS SCOPE clause to define the required scope table or view.
  - If the target type of the reference needs to be changed, the column must be cast first to the representation type of the reference and then to the new reference type. The scope in this case can be specified in the cast to the reference type or using the WITH OPTIONS SCOPE clause. For example, assume you select column Y defined as REF(TYP1) SCOPE TAB1. You want this to be defined as REF(VTYP1) SCOPE VIEW1. The select list item would be as follows:

    ```
    CAST(CAST(Y AS VARCHAR(16) FOR BIT DATA) AS REF(VTYP1) SCOPE VIEW1)
    ```

- ***Identity columns:*** A column of a view is considered an identity column, if the element of the corresponding column in the fullselect of the view definition is the name of an identity column of a table, or the name of a column of a view which directly or indirectly maps to the name of an identity column of a base table.

  In all other cases, the columns of a view will not get the identity property. For example:
  - the select-list of the view definition includes multiple instances of the name of an identity column (that is, selecting the same column more than once)
  - the view definition involves a join
  - a column in the view definition includes an expression that refers to an identity column
  - the view definition includes a UNION

  When inserting into a view for which the select list of the view definition directly or indirectly includes the name of an identity column of a base table, the same rules apply as if the INSERT statement directly referenced the identity column of the base table.
- ***Federated views:*** A federated view is a view that includes a reference to a nickname somewhere in the fullselect. The presence of such a nickname changes the authorization model used for the view when the view is subsequently referenced in a query.

  When the view is created, no privilege checking is done to determine whether the view definer has access to the underlying data source table or view of a nickname. Privilege checking of references to tables or views at the federated database are handled as usual, requiring the view definer to have at least SELECT privilege on such objects.

  When a federated view is subsequently referenced in a query, the nicknames result in queries against the data source, and the authorization ID that issued the query (or the remote authorization ID to which it maps) must have the necessary privileges to access the data source table or view. The authorization

ID that issues the query referencing the federated view is not required to have any additional privileges on tables or views (non-federated) that exist at the federated server.

- ***ROW MOVEMENT, triggers and constraints:*** When a view that is defined using the WITH ROW MOVEMENT clause is updated, the sequence of trigger and constraints operations is as follows:
  1. BEFORE UPDATE triggers are activated for all rows being updated, including rows that will eventually be moved.
  2. The update operation is processed.
  3. Constraints are processed for all updated rows.
  4. AFTER UPDATE triggers (both row-level and statement-level) are activated in creation order, for all rows that satisfy the constraints after the update operation. Because this is an UPDATE statement, all UPDATE statement-level triggers are activated for all underlying tables.
  5. BEFORE DELETE triggers are activated for all rows that did not satisfy the constraints after the update operation (these are the rows that are to be moved).
  6. The delete operation is processed.
  7. Constraints are processed for all deleted rows.
  8. AFTER DELETE triggers (both row-level and statement-level) are activated in creation order, for all deleted rows. Statement-level triggers are activated for only those tables that are involved in the delete operation.
  9. BEFORE INSERT triggers are activated for all rows being inserted (that is, the rows being moved). The new transition tables for the BEFORE INSERT triggers contain the input data provided by the user. Such triggers cannot contain an UPDATE, a DELETE, or an INSERT operation, or invoke any routine containing such operations (SQLSTATE 42987).
  10. The insert operation is processed.
  11. Constraints are processed for all inserted rows.
  12. AFTER INSERT triggers (both row-level and statement-level) are activated in creation order, for all inserted rows. Statement-level triggers are activated for only those tables that are involved in the insert operation.
- ***Nested UNION ALL views:*** A view defined with UNION ALL and based, either directly or indirectly, on a view that is also defined with UNION ALL cannot be updated if either view is defined using the WITH ROW MOVEMENT clause (SQLSTATE 429BK).
- ***Considerations for implicitly hidden columns:*** It is possible that the result table of the fullselect will include a column of the base table that is defined as implicitly hidden. This can occur when the implicitly hidden column is explicitly referenced in the fullselect of the view definition. However, the corresponding column of the view does not inherit the implicitly hidden attribute. Columns of a view cannot be defined as hidden.
- ***Subselect***: The *isolation-clause* cannot be specified in the *fullselect* (SQLSTATE 42601).
- ***Obfuscation***: The CREATE VIEW statement can be submitted in obfuscated form. In an obfuscated statement, only the view name is readable. The rest of the statement is encoded in such a way that is not readable but can be decoded by the database server. Obfuscated statements can be produced by calling the DBMS_DDL.WRAP function.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products.
  - The FEDERATED keyword can be specified between the keywords CREATE and VIEW. The FEDERATED keyword is ignored, however, because a warning is no longer returned if federated objects are used in the view definition.

Examples

- *Example 1:* Create a view named MA_PROJ upon the PROJECT table that contains only those rows with a project number (PROJNO) starting with the letters 'MA'.

```
CREATE VIEW MA_PROJ  AS SELECT *
  FROM PROJECT
   WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

- *Example 2:* Create a view as in example 1, but select only the columns for project number (PROJNO), project name (PROJNAME) and employee in charge of the project (RESPEMP).

```
CREATE VIEW MA_PROJ
  AS SELECTPROJNO, PROJNAME, RESPEMP
  FROM PROJECT
  WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

- *Example 3:* Create a view as in example 2, but, in the view, call the column for the employee in charge of the project IN_CHARGE.

```
CREATE VIEW MA_PROJ
  (PROJNO, PROJNAME, IN_CHARGE)
  AS SELECTPROJNO, PROJNAME, RESPEMP
  FROM PROJECT
  WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

  Note: Even though only one of the column names is being changed, the names of all three columns in the view must be listed in the parentheses that follow MA_PROJ.

- *Example 4:* Create a view named PRJ_LEADER that contains the first four columns (PROJNO, PROJNAME, DEPTNO, RESPEMP) from the PROJECT table together with the last name (LASTNAME) of the person who is responsible for the project (RESPEMP). Obtain the name from the EMPLOYEE table by matching EMPNO in EMPLOYEE to RESPEMP in PROJECT.

```
CREATE VIEW PRJ_LEADER
  AS SELECT PROJNO, PROJNAME, DEPTNO, RESPEMP, LASTNAME
  FROM PROJECT, EMPLOYEE
  WHERE RESPEMP = EMPNO
```

- *Example 5:* Create a view as in example 4, but in addition to the columns PROJNO, PROJNAME, DEPTNO, RESPEMP, and LASTNAME, show the total pay (SALARY + BONUS + COMM) of the employee who is responsible. Also select only those projects with mean staffing (PRSTAFF) greater than one.

```
CREATE VIEW PRJ_LEADER
(PROJNO, PROJNAME, DEPTNO, RESPEMP, LASTNAME, TOTAL_PAY )
AS SELECT PROJNO, PROJNAME, DEPTNO, RESPEMP, LASTNAME, SALARY+BONUS+COMM
  FROM PROJECT, EMPLOYEE
  WHERE RESPEMP = EMPNO
  AND PRSTAFF > 1
```

  Specifying the column name list could be avoided by naming the expression SALARY+BONUS+COMM as TOTAL_PAY in the fullselect.

```
CREATE VIEW PRJ_LEADER
  AS SELECT PROJNO, PROJNAME, DEPTNO, RESPEMP,
              LASTNAME, SALARY+BONUS+COMM AS TOTAL_PAY
    FROM PROJECT, EMPLOYEE
    WHERE RESPEMP = EMPNO AND PRSTAFF > 1
```

- *Example 6:* Given the set of tables and views shown in the following figure:

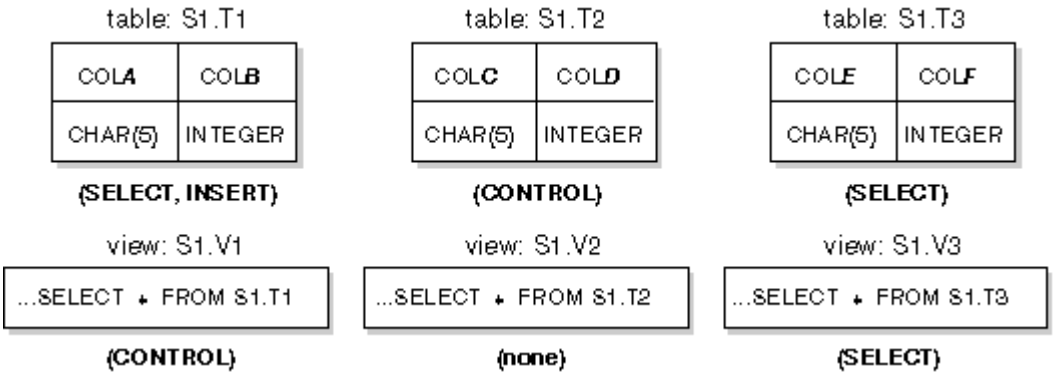


Figure 5. Tables and Views for Example 6

User ZORPIE (who does not have ACCESSCTRL, DATAACCESS, or DBADM authority) has the privileges shown in parentheses for each object:

1. ZORPIE will get CONTROL privilege on the view that she creates with:

   ```
   CREATE VIEW VA AS SELECT * FROM S1.V1
   ```

   because she has CONTROL on S1.V1. (CONTROL on S1.V1 must have been granted to ZORPIE by someone with ACCESSCTRL or SECADM authority.) It does not matter which, if any, privileges she has on the underlying base table.
2. ZORPIE will not be allowed to create the view:

   ```
   CREATE VIEW VB AS SELECT * FROM S1.V2
   ```

   because she has neither CONTROL nor SELECT on S1.V2. It does not matter that she has CONTROL on the underlying base table (S1.T2).
3. ZORPIE will get CONTROL privilege on the view that she creates with:

   ```
   CREATE VIEW VC (COLA, COLB, COLC, COLD)
     AS SELECT * FROM S1.V1, S1.T2
     WHERE COLA = COLC
   ```

   because the fullselect of ZORPIE.VC references view S1.V1 and table S1.T2 and she has CONTROL on both of these. Note that the view VC is read-only, so ZORPIE does not get INSERT, UPDATE or DELETE privileges.
4. ZORPIE will get SELECT privilege on the view that she creates with:

   ```
   CREATE VIEW VD (COLA,COLB, COLE, COLF)
     AS SELECT * FROM S1.V1, S1.V3
     WHERE COLA = COLE
   ```

   because the fullselect of ZORPIE.VD references the two views S1.V1 and S1.V3, one on which she has only SELECT privilege, and one on which she has CONTROL privilege. She is given the lesser of the two privileges, SELECT, on ZORPIE.VD.
5. ZORPIE will get INSERT, UPDATE and DELETE privilege WITH GRANT OPTION and SELECT privilege on the view VE in the following view definition.

   ```
   CREATE VIEW VE
      AS SELECT * FROM S1.V1
     WHERE COLA > ANY
            (SELECT COLE FROM S1.V3)
   ```

   ZORPIE's privileges on VE are determined primarily by her privileges on S1.V1. Since S1.V3 is only referenced in a subquery, she only needs SELECT privilege on S1.V3 to create the view VE. The definer of a view only gets CONTROL on the view if they have CONTROL on all objects referenced in the view definition. ZORPIE does not have CONTROL on S1.V3, consequently she does not get CONTROL on VE.

CREATE WORK ACTION SET

The CREATE WORK ACTION SET statement defines a work action set and work actions within the work action set.

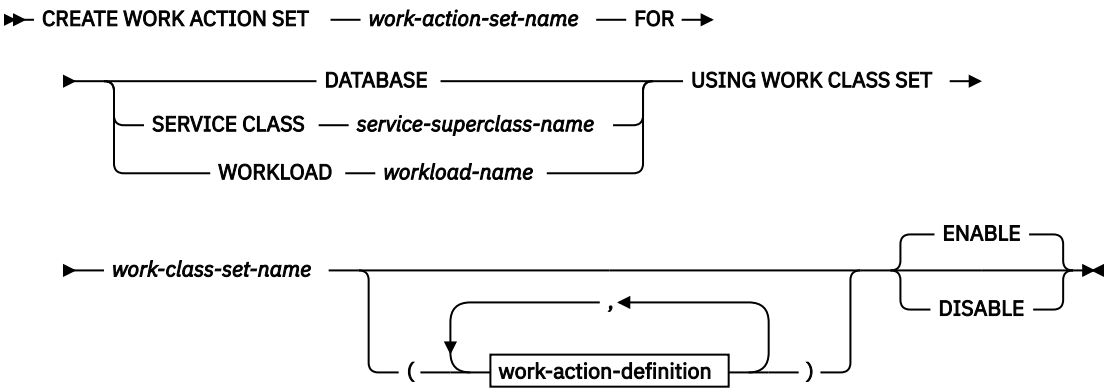
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

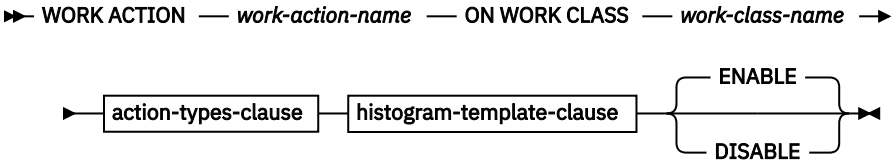
Authorization

The privileges held by the authorization ID of the statement must include WLMADM or DBADM authority.

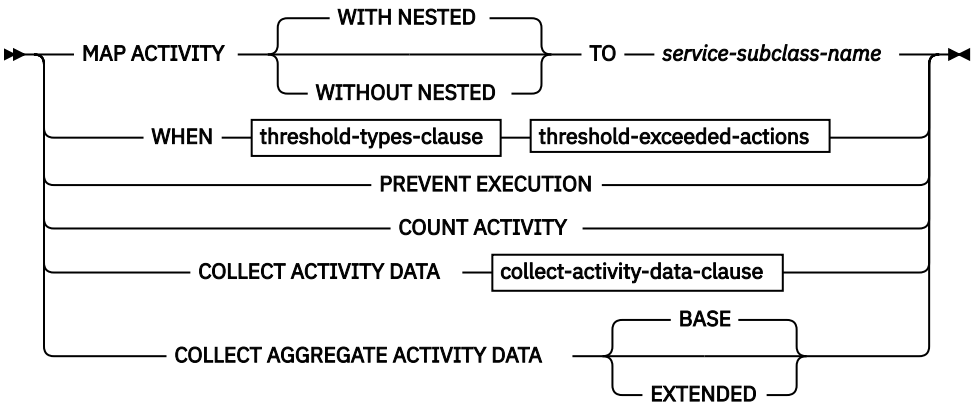
Syntax



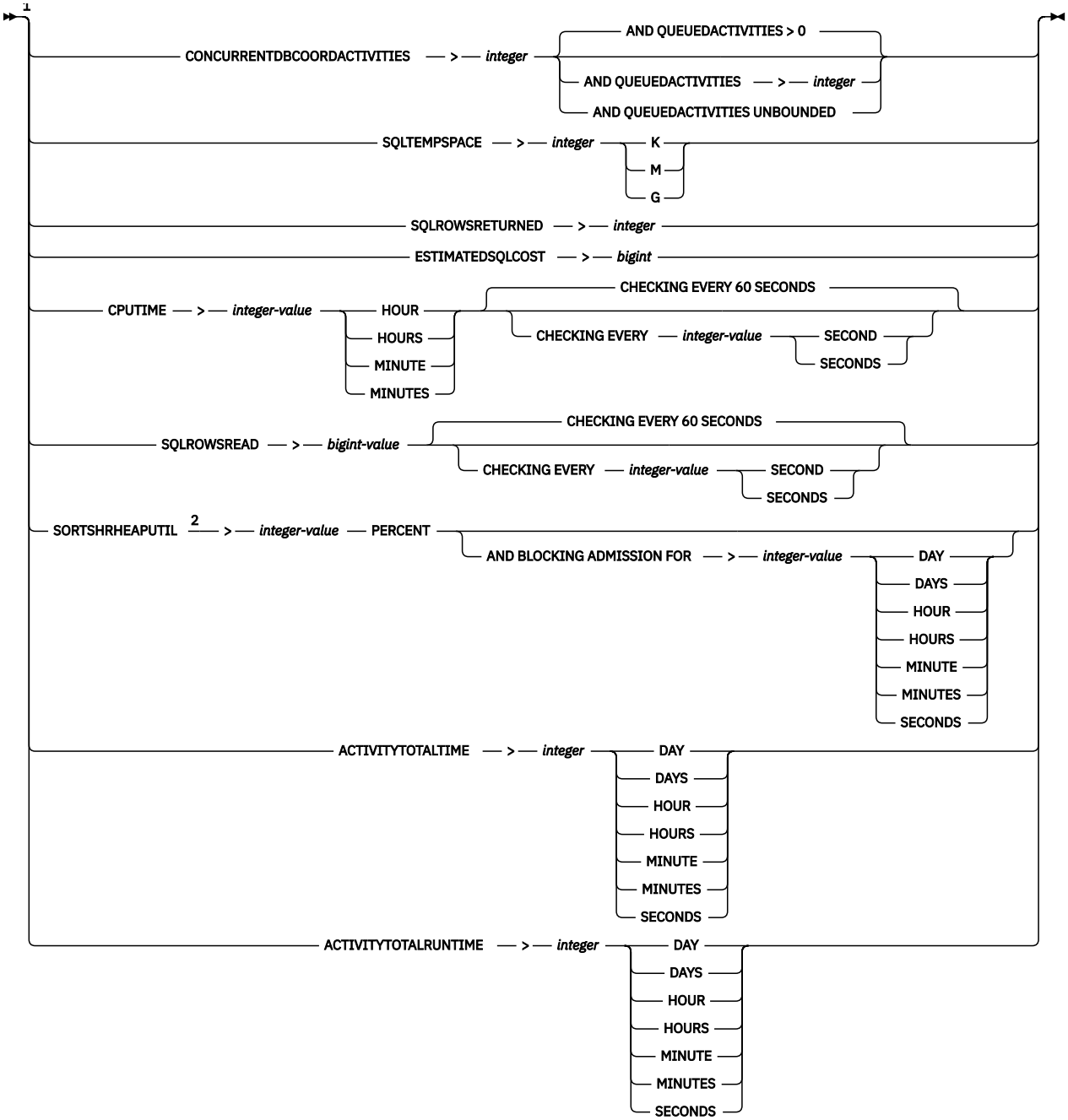
work-action-definition



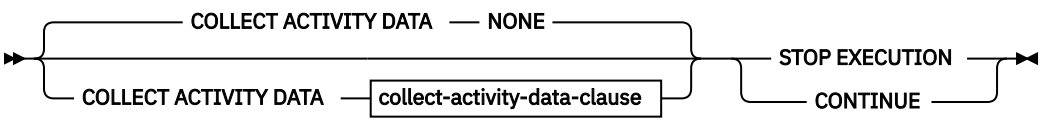
action-types-clause



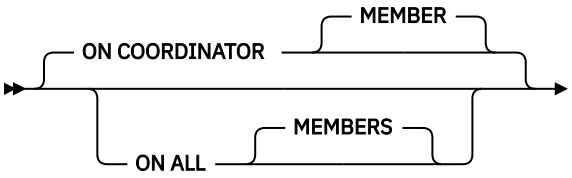
threshold-types-clause

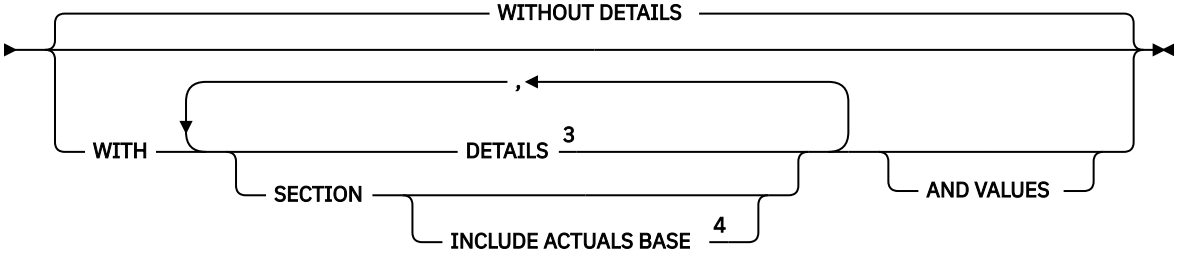


threshold-exceeded-actions

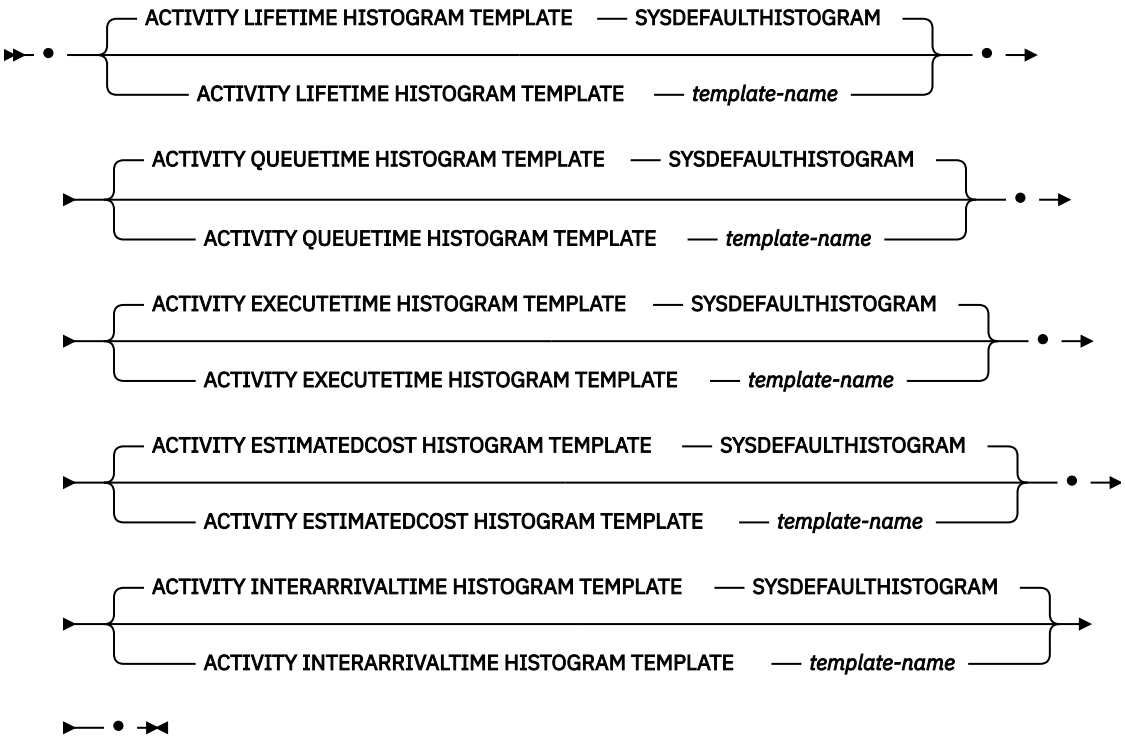


collect-activity-data-clause





histogram-template-clause



Notes:

[1] Only one work action of the same threshold type can be applied to a single work class at a time.
[2] This feature is available in Db2 Version 11.5 Mod Pack 2 and later versions.
[3] The DETAILS keyword is the minimum to be specified, followed by the option separated by a comma.
[4] This clause does not apply to thresholds.

Description

work-action-set-name
Names the work action set. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *work-action-set-name* must not identify a work action set that already exists at the current server (SQLSTATE 42710). The name must not begin with the characters 'SYS' (SQLSTATE 42939).

FOR
Specifies the database manager object to which the actions in this work action set will apply. Each database manager object can have only one work action set defined for it (SQLSTATE 5U017).

DATABASE
The actions in this work action set are to apply to the database. If DATABASE is specified, the MAP ACTIVITY action cannot be specified (SQLSTATE 5U034).

SERVICE CLASS *service-superclass-name*
The actions in this work action set are to apply to *service-superclass-name*. If SERVICE CLASS is specified, threshold actions cannot be specified (SQLSTATE 5U034). The *service-superclass-name* must exist at the current server (SQLSTATE 42704). The *service-superclass-name* must not be a service subclass and cannot be any of the following classes (SQLSTATE 5U032):

- The system service class (SYSDEFAULTSYSTEMCLASS)
- The maintenance service class (SYSDEFAULTMAINTENANCECLASS)
- The default user service class (SYSDEFAULTUSERCLASS)

WORKLOAD *workload-name*
The actions in this work action set are to apply to workload *workload-name*. If WORKLOAD is specified, the MAP ACTIVITY action cannot be specified (SQLSTATE 5U034). The *workload-name* must exist at the current server (SQLSTATE 42704). The *workload-name* cannot be the SYSDEFAULTADMWORKLOAD (SQLSTATE 5U032).

USING WORK CLASS SET *work-class-set-name*
Specifies the work class set containing the work classes that will classify database activities on which to perform actions. The *work-class-set-name* must exist at the current server (SQLSTATE 42704).

work-action-definition
Specifies the definition of the work action.

WORK ACTION *work-action-name*
Names the work action. The *work-action-name* must not identify a work action that already exists at the current server under this work action set (SQLSTATE 42710). The *work-action-name* cannot begin with 'SYS' (SQLSTATE 42939).

ON WORK CLASS *work-class-name*
Specifies the work class that identifies the database activities to which this work action will apply. The *work-class-name* must exist in the *work-class-set-name* at the current server (SQLSTATE 42704).

MAP ACTIVITY
Specifies a work action of mapping the activity. This action can only be specified if the object for which this work action set is defined is a service superclass (SQLSTATE 5U034).

WITH NESTED or WITHOUT NESTED
Specifies whether or not activities that are nested under this activity are mapped to the service subclass. The default is WITH NESTED.

WITH NESTED
All database activities that have a nesting level of zero that are classified under the work class, and all database activities nested under this activity, are mapped to the service subclass; that is, activities with a nesting level greater than zero are run under the same service class as activities with a nesting level of zero.

WITHOUT NESTED
Only database activities that have a nesting level of zero that are classified under the work class are mapped to the service subclass. Database activities that are nested under this activity are handled according to their activity type.

TO *service-subclass-name*
Specifies the service subclass to which activities are to be mapped. The *service-subclass-name* must already exist in the *service-superclass-name* at the current server (SQLSTATE 42704). The *service-subclass-name* cannot be the default service subclass, SYSDEFAULTSUBCLASS (SQLSTATE 5U018).

WHEN
Specifies the threshold that will be applied to the database activity that is associated with the work class for which this work action is defined. A threshold can only be specified if the database

manager object for which this work action set is defined is a database or a workload (SQLSTATE 5U034). None of these thresholds apply to internal database activities initiated by the database manager or to database activities generated by administrative SQL routines.

threshold-types-clause
For a description of valid threshold types, see "CREATE THRESHOLD" statement.

threshold-exceeded-actions
For a description of valid threshold-exceeded actions, see "CREATE THRESHOLD" statement.

PREVENT EXECUTION
Specifies that none of the database activities associated with the work class for which this work action is defined will be allowed to run (SQLSTATE 5U033).

COUNT ACTIVITY
Specifies that all of the database activities associated with the work class for which this work action is defined are to be run and that each time one is run, the counter for the work class will be incremented.

COLLECT ACTIVITY DATA
Specifies that data about each activity associated with the work class for which this work action is defined is to be sent to any active activities event monitor when the activity completes. The default is COLLECT ACTIVITY DATA WITHOUT DETAILS.

collect-activity-data-clause

ON COORDINATOR MEMBER
Specifies that the activity data is to be collected only at the coordinator member of the activity.

ON ALL MEMBERS
Specifies that activity data is to be collected at all members where the activity is processed. On remote members, a record for the activity may be captured multiple times as the activity comes and goes on those members. If the AND VALUES clause is specified, activity input values will be collected only for the members of the coordinator.

WITHOUT DETAILS
Specifies that data about each activity that executes in the service class should be sent to any active activities event monitor, when the activity completes execution. Details about statement, compilation environment, and section environment data are not sent.

WITH

DETAILS
Specifies that statement and compilation environment data is to be sent to any active activities event monitor, for those activities that have them. Section environment data is not sent.

SECTION
Specifies that statement, compilation environment, section environment data, and section actuals are to be sent to any active activities event monitor for those activities that have them. DETAILS must be specified if SECTION is specified. Section actuals will be collected on any member where the activity data is collected.

INCLUDE ACTUALS BASE
Specifies that section actuals should also be collected on any partition where the activity data is collected. For section actuals to be collected, either INCLUDE ACTUALS clause must be specified or the **section_actuals** database configuration parameter must be set.

The effective setting for the collection of section actuals is the combination of the INCLUDE ACTUALS clause, the **section_actuals** database configuration parameter, and the <collectsectionactuals> setting specified on the WLM_SET_CONN_ENV routine. For example, if INCLUDE ACTUALS BASE is

specified, yet the **section_actuals** database configuration parameter value is NONE and <collectsectionactuals> is set to NONE, then the effective setting for the collection of section actuals is BASE.

BASE specifies that the following should be enabled and collected during the activity's execution:

- Basic operator cardinality counts
- Statistics for each object referenced (DML statements only)

AND VALUES
Specifies that input data values are to be sent to any active activities event monitor, for those activities that have them. This data does not include SQL statements that are compiled by using the REOPT ALWAYS bind option.

COLLECT AGGREGATE ACTIVITY DATA
Specifies that aggregate activity data is to be captured for activities that are associated with the work class for which this work action is defined and sent to the statistics event monitor, if one is active. This information is collected periodically on an interval that is specified by the **wlm_collect_int** database configuration parameter. The default is COLLECT AGGREGATE ACTIVITY DATA BASE. This clause cannot be specified for a work action defined in a work action set that is applied to a database.

BASE
Specifies that basic aggregate activity data should be captured for activities associated with the work class for which this work action is defined and sent to the statistics event monitor, if one is active. Basic aggregate activity data includes:

- Estimated activity cost high watermark
- Rows returned high watermark
- Temporary table space usage high watermark. Only activities that have an SQLTEMPSPACE threshold applied to them participate in this high watermark.
- Activity life time histogram
- Activity queue time histogram
- Activity execution time histogram

EXTENDED
Specifies that all aggregate activity data should be captured for activities associated with the work class for which this work action is defined and sent to the statistics event monitor, if one is active. This includes all basic aggregate activity data plus:

- Activity data manipulation language (DML) estimated cost histogram
- Activity DML inter-arrival time histogram

ENABLE or DISABLE
Specifies whether or not the work action is to be considered when database activities are submitted. The default is ENABLE.

ENABLE
Specifies that the work action is enabled and will be considered when database activities are submitted.

DISABLE
Specifies that the work action is disabled and will not be considered when database activities are submitted.

ENABLE or DISABLE
Specifies whether or not the work action set is to be considered when database activities are submitted. The default is ENABLE.

ENABLE
Specifies that the work action set is enabled and will be considered when database activities are submitted.

DISABLE
Specifies that the work action set is disabled and will not be considered when database activities are submitted.

histogram-template-clause
Specifies histogram templates to use when collecting aggregate activity data for activities associated with the work class to which this work action is assigned. Aggregate activity data is only collected for the work class when the work action type is COLLECT AGGREGATE ACTIVITY DATA.

ACTIVITY LIFETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the duration, in milliseconds, of database activities running during a specific interval. The database activities are those associated with the work class to which this work action is assigned. This time includes both time queued and time executing. The default is SYSDEFAULTHISTOGRAM. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

ACTIVITY QUEUETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, that database activities are queued during a specific interval. The database activities are those associated with the work class to which this work action is assigned. The default is SYSDEFAULTHISTOGRAM. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

ACTIVITY EXECUTETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, that database activities are executing during a specific interval. The database activities are those associated with the work class to which this work action is assigned. This time does not include the time spent queued. Activity execution time is collected in this histogram at each member where the activity executes. On the activity's coordinator member, this is the end-to-end execution time (that is, the life time less the time spent queued). On non-coordinator members, this is the time that these members spend working on behalf of the activity. During the execution of a given activity, the database manager might present work to a non-coordinator member more than once, and each time the non-coordinator member will collect the execution time for that occurrence of the activity. Therefore, the counts in the execution time histogram might not represent the actual number of unique activities that executed on a member. The default is SYSDEFAULTHISTOGRAM. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

ACTIVITY ESTIMATEDCOST HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the estimated cost, in timerons, of DML activities associated with the work class to which this work action is assigned. The default is SYSDEFAULTHISTOGRAM. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified with the EXTENDED option.

ACTIVITY INTERARRIVALTIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, between the arrival of one DML activity and the arrival of the next DML activity, for any activity associated with the work class to which this work action is assigned. The default is SYSDEFAULTHISTOGRAM. This information is only collected when the COLLECT AGGREGATE ACTIVITY DATA clause is specified with the EXTENDED option.

Rules

- A workload management (WLM)-exclusive SQL statement must be followed by a COMMIT or a ROLLBACK statement (SQLSTATE 5U021). WLM-exclusive SQL statements are:
  - CREATE HISTOGRAM TEMPLATE, ALTER HISTOGRAM TEMPLATE, or DROP (histogram template)

  - CREATE SERVICE CLASS, ALTER SERVICE CLASS, or DROP (service class)
  - CREATE THRESHOLD, ALTER THRESHOLD, or DROP (threshold)
  - CREATE WORK ACTION SET, ALTER WORK ACTION SET, or DROP (work action set)
  - CREATE WORK CLASS SET, ALTER WORK CLASS SET, or DROP (work class set)
  - CREATE WORKLOAD, ALTER WORKLOAD, or DROP (workload)
  - GRANT (workload privileges) or REVOKE (workload privileges)
- A WLM-exclusive SQL statement cannot be issued within a global transaction (SQLSTATE 51041) such as, for example, an XA transaction.

Notes

- Changes are written to the system catalog, but do not take effect until they are committed, even for the connection that issues the statement.
- Only one uncommitted WLM-exclusive SQL statement at a time is allowed across all partitions. If an uncommitted WLM-exclusive SQL statement is executing, subsequent WLM-exclusive SQL statements will wait until the current WLM-exclusive SQL statement commits or rolls back.
- ***Syntax alternatives***: The following are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - DATABASE PARTITION can be specified in place of MEMBER, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
  - DATABASE PARTITIONS can be specified in place of MEMBERS, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
- The enforcement scope is determined automatically based on the threshold type. For CONCURRENTDBCOORDACTIVITIES type thresholds, the environment is also used to determine the enforcement scope where it defaults to the DATABASE enforcement scope in environments other than Db2 pureScale, and the MEMBER enforcement scope in Db2 pureScale environments.

Examples

- *Example 1:* Create a work action set named DATABASE_ACTIONS to apply to all database activities. Use the LARGE_QUERIES work class set and define the following work actions. Work action ONE_CONCURRENT_QUERY has a threshold action that allows one concurrent query to run on the system at a time for queries that fall within the LARGE_ESTIMATED_COST work class. If that threshold is exceeded, the database manager is to queue the activity, but is not to allow more than one database activity to be queued at a time. If the queue threshold is exceeded, the database activity is not to be allowed to run. Work action TWO_CONCURRENT_QUERIES has a threshold action that allows two concurrent queries to execute at the same time for queries that fall within the LARGE_CARDINALITY work class, and allows no more than two to be queued. If more than two queries are to be queued, the database activity is to continue putting the queries in the queue and is to collect the database activity data in the activities event monitor, if one is active.

```
CREATE WORK ACTION SET DATABASE_ACTIONS
  FOR DATABASE USING WORK CLASS SET LARGE_QUERIES
    (WORK ACTION ONE_CONCURRENT_QUERY ON WORK CLASS LARGE_ESTIMATED_COST
     WHEN CONCURRENTDBCOORDACTIVITIES > 1 AND QUEUEDACTIVITIES > 1
       STOP EXECUTION,
     WORK ACTION TWO_CONCURRENT_QUERIES ON WORK CLASS LARGE_CARDINALITY
     WHEN CONCURRENTDBCOORDACTIVITIES > 2 AND QUEUEDACTIVITIES > 2
       COLLECT ACTIVITY DATA CONTINUE)
```

- *Example 2:* Create a work action set named ADMIN_APPS_ACTIONS with one work action named MAP_SELECTS that is to apply to database activities that run under service superclass ADMIN_APPS. The work action is to map all database activity that falls within the SELECT_CLASS work class to service subclass SELECTS_SERVICE_CLASS, which is in the DML_SELECTS work class set.

```
CREATE WORK ACTION SET ADMIN_APPS_ACTIONS
  FOR SERVICE CLASS ADMIN_APPS USING
```

```
        WORK CLASS SET DML_SELECTS
          (WORK ACTION MAP_SELECTS ON WORK CLASS SELECT_CLASS
           MAP ACTIVITY TO SELECTS_SERVICE_CLASS)
```

CREATE WORK CLASS SET

The CREATE WORK CLASS SET statement defines a work class set.

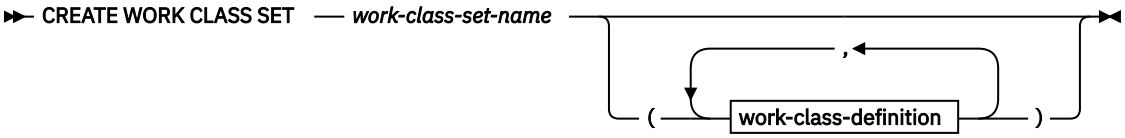
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

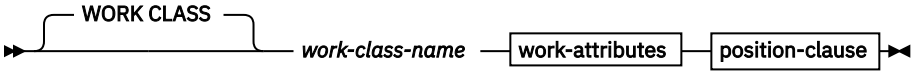
Authorization

The privileges held by the authorization ID of the statement must include WLMADM or DBADM authority.

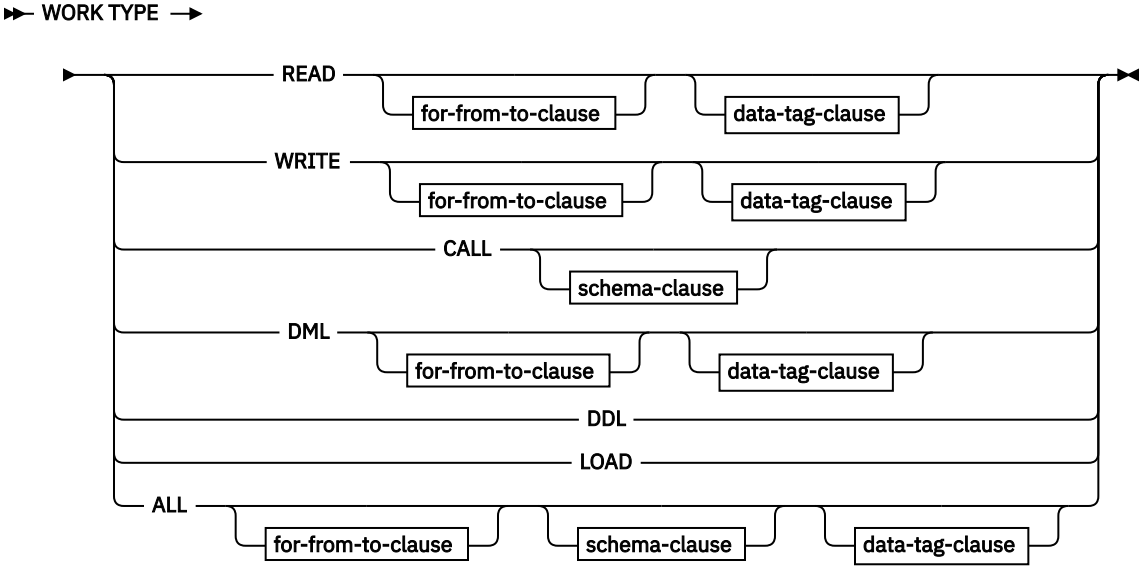
Syntax



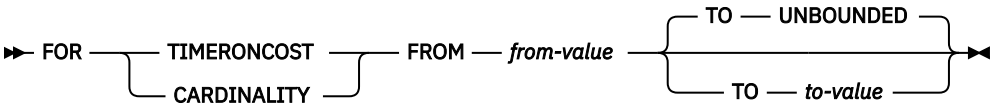
work-class-definition



work-attributes

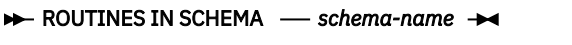


for-from-to-clause

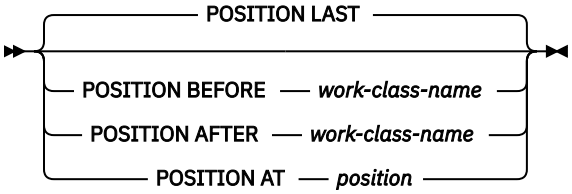


data-tag-clause

schema-clause



position-clause



Description

work-class-set-name
Names the work class set. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *work-class-set-name* must not identify a work class set that already exists at the current server (SQLSTATE 42710). The name must not begin with the characters 'SYS' (SQLSTATE 42939).

work-class-definition
Specifies the definition of the work class.

WORK CLASS *work-class-name*
Names the work class. The *work-class-name* must not identify a work class that already exists within the work class set at the current server (SQLSTATE 42710). The *work-class-name* cannot begin with 'SYS' (SQLSTATE 42939).

work-attributes
The attributes of the database activity must match all of the attributes specified in this work class if that activity is to be associated with this work class.

WORK TYPE
Specifies the type of database activity.

READ
This activity includes the following statements:

- All SELECT or SELECT INTO statements that do not contain a DELETE, INSERT, MERGE, or UPDATE statement, and all VALUES INTO statements
- All XQuery statements

WRITE
This activity includes the following statements:

- UPDATE
- DELETE
- INSERT
- MERGE
- All SELECT statements that contain a DELETE, INSERT, or UPDATE statement, and all VALUES INTO statements

CALL
Includes the CALL statement. A CALL statement is considered for a work class with a work type of CALL or ALL.

DML
Includes the statements listed under READ and WRITE.

DDL
This activity includes the following statements:

- ALTER

- CREATE
- COMMENT
- DECLARE GLOBAL TEMPORARY TABLE
- DROP
- FLUSH PACKAGE CACHE
- GRANT
- REFRESH TABLE
- RENAME
- REVOKE
- SET INTEGRITY

LOAD
Db2 load operations.

ALL
All recognized workload management (WLM) activity that falls under any one of the keywords previously listed within the description for WORK TYPE.

FOR
Indicates the type of information that is being specified in the FROM *from-value* TO *to-value* clause. The FOR clause is only used for the following work types:

- ALL
- DML
- READ
- WRITE

TIMERONCOST
The estimated cost of the work, in timerons. This value is used to determine whether the work falls within the range specified in the FROM *from-value* TO *to-value* clause.

CARDINALITY
The estimated cardinality of the work. This value is used to determine whether the work falls within the range specified in the FROM *from-value* TO *to-value* clause.

FROM *from-value* TO UNBOUNDED or FROM *from-value* TO *to-value*
Specifies the range of either timeron value (for estimated cost) or cardinality within which the database activity must fall if it is to be part of this work class. The range is inclusive of *from-value* and *to-value*. If this clause is not specified for the work class, all work that falls within the specified work type will be included (that is, the default is FROM 0 TO UNBOUNDED). This range is only used for the following work types:

- ALL
- DML
- READ
- WRITE

FROM *from-value* TO UNBOUNDED
The *from-value* must be zero or a positive DOUBLE value (SQLSTATE 5U019). The range has no upper bound.

FROM *from-value* TO *to-value*
The *from-value* must be zero or a positive DOUBLE value and the *to-value* must be a positive DOUBLE value. The *from-value* must be smaller than or equal to the *to-value* (SQLSTATE 5U019).

DATA TAG LIST CONTAINS *integer-constant*
Specifies the value of the tag given to any data which the database activity might touch if it is to be part of this work class. If the clause is not specified for the work class, all work that falls

within the specified work type, regardless of what data it might touch, will be included (that is, the default is to ignore the data tag). This clause is used only if the work type is READ, WRITE, DML, or ALL and the database activity is a DML statement. Valid values for *integer-constant* are integers from 1 to 9.

schema-clause

ROUTINES IN SCHEMA *schema-name*
Specifies the schema name of the procedure that the CALL statement will be calling. This clause is only used if the work type is CALL or ALL and the database activity is a CALL statement. If no value is specified, all schemas are included.

position-clause

POSITION
Specifies where this work class is to be placed within the work class set, which determines the order in which work classes are evaluated. When performing work class assignment at run time, the database manager first determines the work class set that is associated with the object, either the database or a service superclass. The first matching work class within that work class set is then selected. If this keyword is not specified, the work class is placed in the last position.

LAST
Specifies that the work class is to be placed last in the ordered list of work classes within the work class set. This is the default.

BEFORE *work-class-name*
Specifies that the work class is to be placed before work class *work-class-name* in the list. The *work-class-name* must identify a work class in the work class set that exists at the current server (SQLSTATE 42704).

AFTER *work-class-name*
Specifies that the work class is to be placed after work class *work-class-name* in the list. The *work-class-name* must identify a work class in the work class set that exists at the current server (SQLSTATE 42704).

AT *position*
Specifies the absolute position at which the work class is to be placed within the work class set in the ordered list of work classes. This value can be any positive integer (not zero) (SQLSTATE 42615). If *position* is greater than the number of existing work classes plus one, the work class is placed at the last position within the work class set.

Rules

- A workload management (WLM)-exclusive SQL statement must be followed by a COMMIT or a ROLLBACK statement (SQLSTATE 5U021). WLM-exclusive SQL statements are:
  - CREATE HISTOGRAM TEMPLATE, ALTER HISTOGRAM TEMPLATE, or DROP (HISTOGRAM TEMPLATE)
  - CREATE SERVICE CLASS, ALTER SERVICE CLASS, or DROP (SERVICE CLASS)
  - CREATE THRESHOLD, ALTER THRESHOLD, or DROP (THRESHOLD)
  - CREATE WORK ACTION SET, ALTER WORK ACTION SET, or DROP (WORK ACTION SET)
  - CREATE WORK CLASS SET, ALTER WORK CLASS SET, or DROP (WORK CLASS SET)
  - CREATE WORKLOAD, ALTER WORKLOAD, or DROP (WORKLOAD)
  - GRANT (Workload Privileges) or REVOKE (Workload Privileges)
- A WLM-exclusive SQL statement cannot be issued within a global transaction (SQLSTATE 51041) such as, for example, an XA transaction.

Notes

- Changes are written to the system catalog, but do not take effect until they are committed, even for the connection that issues the statement.

- Only one uncommitted WLM-exclusive SQL statement at a time is allowed across all partitions. If an uncommitted WLM-exclusive SQL statement is executing, subsequent WLM-exclusive SQL statements will wait until the current WLM-exclusive SQL statement commits or rolls back.

Examples

- *Example 1:* Create a work class set named LARGE_QUERIES that has a set of work classes representing all DML with an estimated cost greater than 9999 and an estimated cardinality greater than 1000.

```
CREATE WORK CLASS SET LARGE_QUERIES
  (WORK CLASS LARGE_ESTIMATED_COST WORK TYPE DML
   FOR TIMERONCOST FROM 9999 TO UNBOUNDED,
   WORK CLASS LARGE_CARDINALITY WORK TYPE DML
   FOR CARDINALITY FROM 1000 TO UNBOUNDED)
```

- *Example 2:* Create a work class set named DML_SELECTS that has a work class representing all DML SELECT statements that do not contain a DELETE, INSERT, MERGE, or UPDATE statement.

```
CREATE WORK CLASS SET DML_SELECTS
  (WORK CLASS SELECT_CLASS WORK TYPE READ)
```

CREATE WORKLOAD

The CREATE WORKLOAD statement defines a workload.

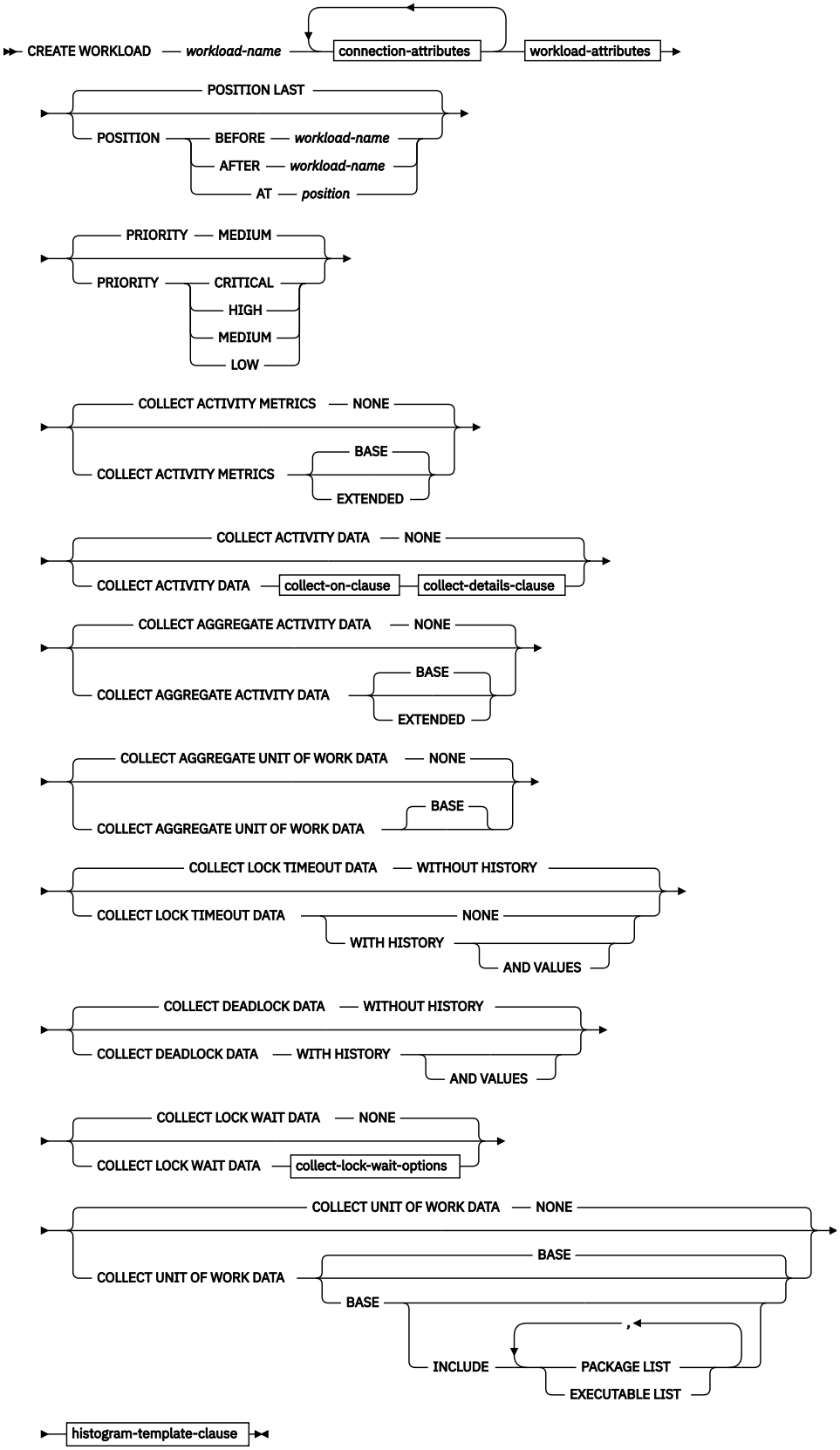
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

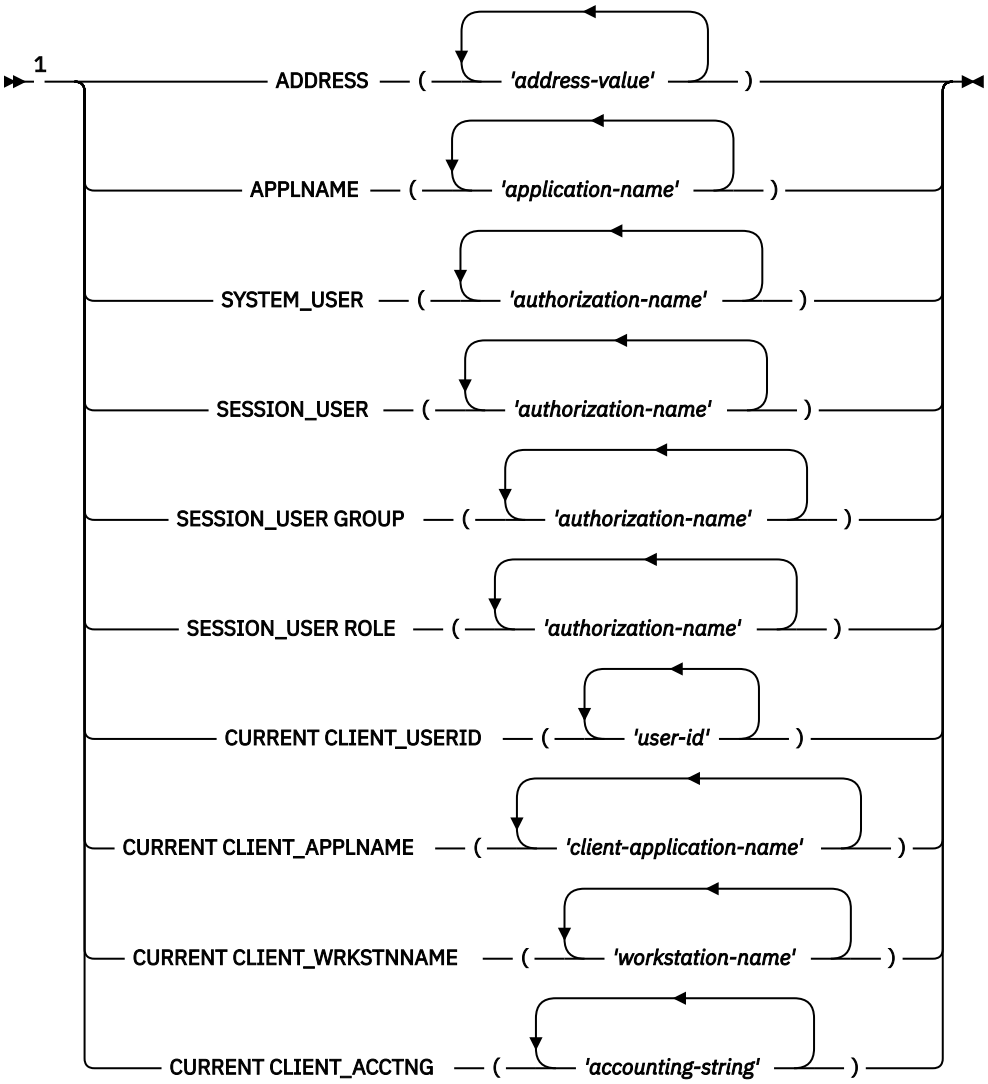
Authorization

The privileges held by the authorization ID of the statement must include WLMADM or DBADM authority.

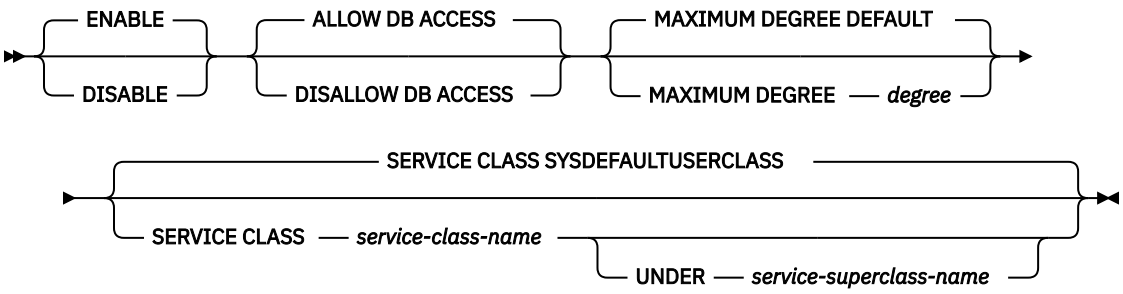
Syntax



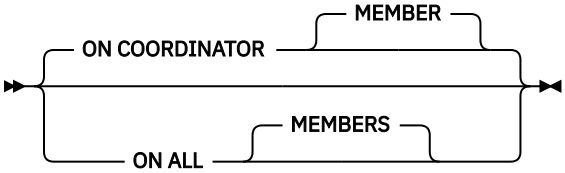
connection-attributes



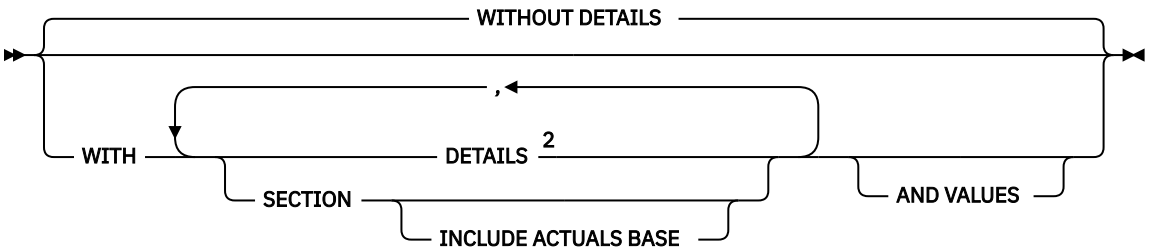
workload-attributes



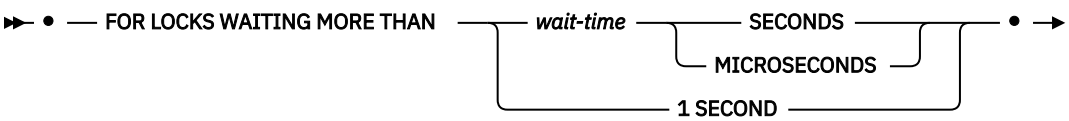
collect-on-clause

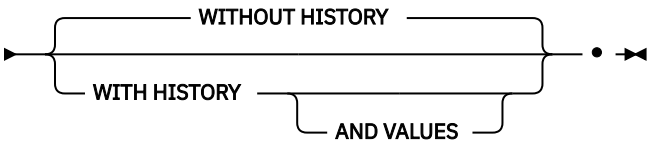


collect-details-clause

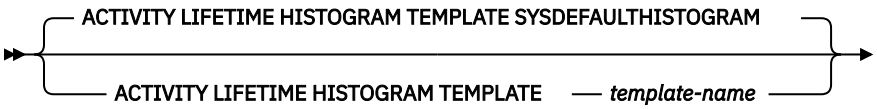


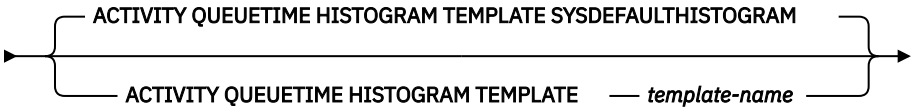
collect-lock-wait-options

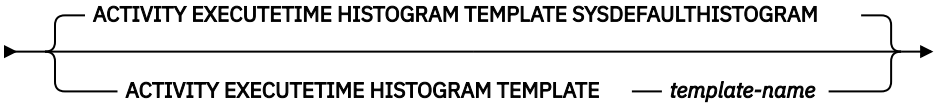


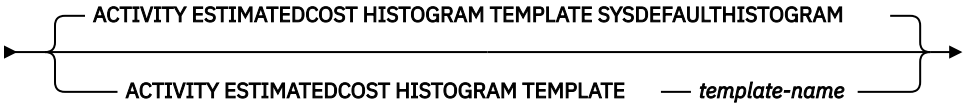


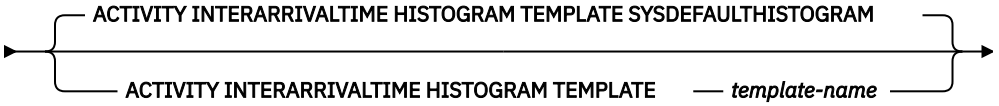
histogram-template-clause

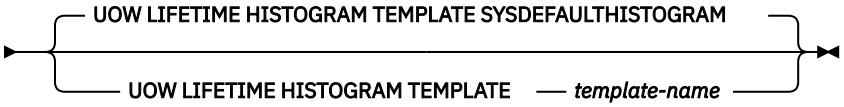












Notes:

[1] Each connection attribute clause can only be specified once.

[2] The DETAILS keyword is the minimum to be specified, followed by the option separated by a comma.

Description

workload-name
Names the workload. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *workload-name* must not identify a workload that already exists at the current server (SQLSTATE 42710). The name must not begin with the characters 'SYS' (SQLSTATE 42939).

connection-attributes
The attributes of the connection must match all attributes specified in this workload definition if it is to be associated with this workload when the connection is established. If a list of values is specified for a connection attribute in the workload definition, the corresponding attribute of the connection must match at least one of the values in the list. If a connection attribute is not specified in the workload definition, the connection can have any value for the corresponding connection attribute.

Note: All connection attributes are case sensitive, except for ADDRESS.

ADDRESS ('*address-value*', ...)
Specifies one or more IPv4 addresses, IPv6 addresses, or secure domain names for the ADDRESS connection attribute. An address value cannot appear more than once in the list (SQLSTATE 42713). The only supported protocol is TCP/IP. Each address value must be an IPv4 address, an IPv6 address, or a secure domain name.

An IPv4 address must not contain leading spaces and is represented as a dotted decimal address. An example of an IPv4 address is `192.0.2.1`. The value `localhost` or its equivalent representation `127.0.0.1` will not result in a match; the real IPv4 address of the host must be specified instead. An IPv6 address must not contain leading spaces and is represented as a colon hexadecimal address. An example of an IPv6 address is `2001:0DB8:0000:0000:0008:0800:200C:417A`. IPv4-mapped IPv6 addresses (`::ffff:192.0.2.1`, for example) will not result in a match. Similarly, `localhost` or its IPv6 short representation `::1` will not result in a match. A domain name is converted to an IP address by the domain name server where a resulting IPv4 or IPv6 address is determined. An example of a domain name is `corona.example.com`. When a domain name is converted to an IP address, the result of this conversion could be a set of one or more IP addresses. In this case, an incoming connection is said to match the ADDRESS attribute of a workload object if the IP address from which the connection originates matches any of the IP addresses to which the domain name was converted.

When creating a workload object, you should specify domain name values for the ADDRESS attribute instead of static IP addresses, particularly in Dynamic Host Configuration Protocol (DHCP) environments where a device can have a different IP address each time it connects to the network.

APPLNAME ('*application-name*', ...)
Specifies one or more applications for the APPLNAME connection attribute. An application name cannot appear more than once in the list (SQLSTATE 42713). If *application-name* does not contain a single asterisk character (*), is equivalent to the value shown in the "Application name" field in system monitor output and in output from the LIST APPLICATIONS command. If *application-name* does contain a single asterisk character (*), the value is used as an expression to represent a set of application names, where the asterisk (*) represents a string of zero or more characters. If the expression needs to include an asterisk character in the application name, use a sequence of two asterisk characters (**).

SYSTEM_USER ('*authorization-name*', ...)
Specifies one or more authorization IDs for the SYSTEM USER connection attribute. An authorization ID cannot appear more than once in the list (SQLSTATE 42713).

SESSION_USER ('*authorization-name*', ...)
Specifies one or more authorization IDs for the SESSION USER connection attribute. An authorization ID cannot appear more than once in the list (SQLSTATE 42713).

SESSION_USER GROUP ('*authorization-name*', ...)
Specifies one or more authorization IDs for the SESSION_USER GROUP connection attribute. An authorization ID cannot appear more than once in the list (SQLSTATE 42713).

SESSION_USER ROLE ('*authorization-name*', ...)
Specifies one or more authorization IDs for the SESSION_USER ROLE connection attribute. The roles of a session authorization ID in this context refer to all the roles that are available to the session authorization ID, regardless of how the roles were obtained. An authorization ID cannot appear more than once in the list (SQLSTATE 42713).

CURRENT CLIENT_USERID ('*user-id*', ...)
Specifies one or more client user IDs for the CURRENT CLIENT_USERID connection attribute. A client user ID cannot appear more than once in the list (SQLSTATE 42713). If *user-id* contains a single asterisk character (*), the value is used as an expression to represent a set of user IDs, where the asterisk (*) represents a string of zero or more characters. If the expression needs to include an asterisk character in the user ID, use a sequence of two asterisk characters (**).

CURRENT CLIENT_APPLNAME ('*client-application-name*', ...)
Specifies one or more applications for the CURRENT CLIENT_APPLNAME connection attribute. An application name cannot appear more than once in the list (SQLSTATE 42713). If *client-application-name* does not contain a single asterisk character (*), is equivalent to the value shown in the "TP Monitor client application name" field in system monitor output. If *client-application-name* does contain a single asterisk character (*), the value is used as an expression to represent a set of application names, where the asterisk (*) represents a string of zero or more characters. If the expression needs to include an asterisk character in the application name, use a sequence of two asterisk characters (**).

CURRENT CLIENT_WRKSTNNAME ('*workstation-name*', ...)
Specifies one or more client workstation names for the CURRENT CLIENT_WRKSTNNAME connection attribute. A client workstation name cannot appear more than once in the list (SQLSTATE 42713). If *workstation-name* contains a single asterisk character (*), the value is used as an expression to represent a set of workstation names, where the asterisk (*) represents a string of zero or more characters. If the expression needs to include an asterisk character in the workstation name, use a sequence of two asterisk characters (**).

CURRENT CLIENT_ACCTNG ('*accounting-string*', ...)
Specifies one or more client accounting strings for the CURRENT CLIENT_ACCTNG connection attribute. A client accounting string cannot appear more than once in the list (SQLSTATE 42713). If *accounting-string* contains a single asterisk character (*), the value is used as an expression to represent a set of accounting strings, where the asterisk (*) represents a string of zero or more characters. If the expression needs to include an asterisk character in the accounting string, use a sequence of two asterisk characters (**).

workload-attributes
Specifies attributes of the workload.

ENABLE or DISABLE
Specifies whether or not this workload will be considered when a workload is chosen. The default is ENABLE.

ENABLE
Specifies that the workload is enabled and will be considered when a workload is chosen.

DISABLE
Specifies that the workload is disabled and will not be considered when a workload is chosen.

ALLOW DB ACCESS or DISALLOW DB ACCESS
Specifies whether or not a workload occurrence associated with this workload is allowed access to the database. The default is ALLOW DB ACCESS.

ALLOW DB ACCESS
Specifies that workload occurrences associated with this workload are allowed access to the database.

DISALLOW DB ACCESS
Specifies that workload occurrences associated with this workload are not allowed access to the database. The next unit of work associated with this workload will be rejected (SQLSTATE 5U020). Workload occurrences that are already running are allowed to complete.

MAXIMUM DEGREE
Specifies the maximum runtime degree of parallelism for this workload. The default is DEFAULT.

DEFAULT
If DB2_WORKLOAD=ANALYTICS, this setting enables intrapartition parallelism for this workload. Otherwise, this setting specifies that this workload inherits the intrapartition parallelism setting from the database manager configuration parameter **intra_parallel**. When **intra_parallel** is set to NO, this workload runs with intrapartition parallelism disabled. When **intra_parallel** is set to YES, this workload runs with intrapartition parallelism enabled. This workload does not specify a maximum runtime degree for assigned applications. Therefore, the actual runtime degree is determined as the lower of the value of **max_querydegree** configuration parameter, the MAXIMUM DEGREE set on the query service class, the value set by SET RUNTIME DEGREE command, and the SQL statement compilation degree.

degree
Specifies the maximum degree of parallelism for this workload. Valid values are 1 to 32,767. With value 1, the associated requests run with intrapartition parallelism disabled. With value 2 to 32,767, the associated requests run with intrapartition parallelism enabled. The actual runtime degree is determined as the lower of this *degree*, the MAXIMUM DEGREE set on the query service class, the value of **max_querydegree** configuration parameter, the value set by SET RUNTIME DEGREE command and the SQL statement compilation degree.

Note: A MAXIMUM DEGREE value greater than 1 will not enable intrapartition parallelism unless the shared sort heap is available.

SERVICE CLASS *service-class-name*
Specifies that requests associated with this workload are to be executed in the service class *service-class-name*. The *service-class-name* must identify a service class that exists at the current server (SQLSTATE 42704). The *service-class-name* cannot be 'SYSDEFAULTSUBCLASS', 'SYSDEFAULTSYSTEMCLASS', or 'SYSDEFAULTMAINTENANCECLASS' (SQLSTATE 5U032). The default is SYSDEFAULTUSERCLASS.

UNDER *service-superclass-name*
This clause is used when specifying a service subclass. The *service-superclass-name* identifies the service superclass of *service-class-name*. The *service-superclass-name* must identify a service superclass that exists at the current server (SQLSTATE 42704). The *service-superclass-name* cannot be 'SYSDEFAULTSYSTEMCLASS' or 'SYSDEFAULTMAINTENANCECLASS' (SQLSTATE 5U032).

POSITION
Specifies where this workload is to be placed within the ordered list of workloads. At run time, this list is searched in order for the first workload that matches the required connection attributes. The default is LAST.

LAST
Specifies that the workload is to be last in the list, before the default workloads SYSDEFAULTUSERWORKLOAD and SYSDEFAULTADMWORKLOAD.

BEFORE *relative-workload-name*
Specifies that the workload is to be placed before workload *relative-workload-name* in the list. The *relative-workload-name* must identify a workload that exists at the current server (SQLSTATE 42704). The BEFORE option cannot be specified if *relative-workload-name* is 'SYSDEFAULTUSERWORKLOAD' or 'SYSDEFAULTADMWORKLOAD' (SQLSTATE 42832).

AFTER *relative-workload-name*
Specifies that the workload is to be placed after workload *relative-workload-name* in the list. The *relative-workload-name* must identify a workload that exists at the current server (SQLSTATE 42704). The AFTER option cannot be specified if *relative-workload-name* is 'SYSDEFAULTUSERWORKLOAD' or 'SYSDEFAULTADMWORKLOAD' (SQLSTATE 42832).

AT *position*
Specifies the absolute position at which the workload is to be placed in the list. This value can be any positive integer (not zero) (SQLSTATE 42615). If *position* is greater than the

number of existing workloads plus one, the workload is placed at the last position, just before SYSDEFAULTUSERWORKLOAD and SYSDEFAULTADMWORKLOAD.

PRIORITY
Specifies the priority of the work from this workload compared to that of the work in other workloads in the same service superclass. Within a service superclass priority is used to prioritize more important jobs over less important jobs. Work scheduling across superclasses does not use the priority for scheduling, but instead uses only resource-based scheduling.

COLLECT ACTIVITY METRICS
Specifies that monitor metrics should be collected for an activity submitted by an occurrence of the workload. The default is COLLECT ACTIVITY METRICS NONE.

Note: The effective activity metrics collection setting is the combination of the attribute specified by the COLLECT ACTIVITY METRICS clause on the workload submitting the activity, and the **mon_act_metrics** database configuration parameter. If either the workload attribute or the configuration parameter has a value other than NONE, metrics will be collected for the activity.

NONE
Specifies that no metrics will be collected for any activity submitted by an occurrence of the workload.

BASE
Specifies that basic metrics will be collected for any activity submitted by an occurrence of the workload.

EXTENDED
Specifies that basic metrics will be collected for any activity submitted by an occurrence of the workload. In addition, specifies that the values for the following monitor elements should be determined with additional granularity:

- **total_section_time**
- **total_section_proc_time**
- **total_routine_user_code_time**
- **total_routine_user_code_proc_time**
- **total_routine_time**

COLLECT ACTIVITY DATA
Specifies that data about each activity associated with this workload is to be sent to any active activities event monitor when the activity completes. The default is COLLECT ACTIVITY DATA NONE.

collect-on-clause
Specifies where the activity data is to be collected. The default is ON COORDINATOR MEMBER.

ON COORDINATOR MEMBER
Specifies that activity data is to be collected only at the coordinator member of the activity.

ON ALL MEMBERS
Specifies that activity data is to be collected at all members where the activity is processed. On remote members, a record for the activity may be captured multiple times as the activity comes and goes on those members. If the AND VALUES clause is specified, activity input values will be collected only for the members of the coordinator.

NONE
Specifies that activity data is not collected for each activity that is associated with this workload.

collect-details-clause
Specifies what type of activity data is to be collected. The default is WITHOUT DETAILS.

WITHOUT DETAILS
Specifies that data about each activity that is associated with this workload is to be sent to any active activities event monitor, when the activity completes execution. Details about statement, compilation environment, and section environment data are not sent.

WITH

DETAILS
Specifies that statement and compilation environment data is to be sent to any active activities event monitor, for those activities that have them. Section environment data is not sent.

SECTION
Specifies that statement, compilation environment, section environment data, and section actuals are to be sent to any active activities event monitor for those activities that have them. DETAILS must be specified if SECTION is specified. Section actuals will be collected on any member where the activity data is collected.

INCLUDE ACTUALS BASE
Specifies that section actuals should also be collected on any partition where the activity data is collected. For section actuals to be collected, either INCLUDE ACTUALS clause must be specified or the **section_actuals** database configuration parameter must be set.

The effective setting for the collection of section actuals is the combination of the INCLUDE ACTUALS clause, the **section_actuals** database configuration parameter, and the <collectsectionactuals> setting specified on the WLM_SET_CONN_ENV routine. For example, if INCLUDE ACTUALS BASE is specified, yet the **section_actuals** database configuration parameter value is NONE and <collectsectionactuals> is set to NONE, then the effective setting for the collection of section actuals is BASE.

BASE specifies that the following should be enabled and collected during the activity's execution:

- Basic operator cardinality counts
- Statistics for each object referenced (DML statements only)

AND VALUES
Specifies that input data values are to be sent to any active activities event monitor, for those activities that have them. This data does not include SQL statements that are compiled by using the REOPT ALWAYS bind option.

COLLECT AGGREGATE ACTIVITY DATA
Specifies that aggregate activity data about the activities associated with this workload is to be sent to the statistics event monitor, if one is active. This information is collected periodically on an interval that is specified by the **wlm_collect_int** database configuration parameter. The default when COLLECT AGGREGATE ACTIVITY DATA is not specified is COLLECT AGGREGATE ACTIVITY DATA NONE. The default when COLLECT AGGREGATE ACTIVITY DATA is specified is COLLECT AGGREGATE ACTIVITY DATA BASE.

BASE
Specifies that basic aggregate activity data about the activities associated with this workload is to be sent to the statistics event monitor, if one is active. Basic aggregate activity data includes:

- Activity CPU time high watermark
- Activity execution time histogram
- Activity life time histogram
- Activity queue time histogram
- Activity rows read high watermark
- Estimated activity cost high watermark
- Rows returned high watermark
- Temporary table space usage high watermark. Only activities that have an SQLTEMPSPACE threshold applied to them participate in this high watermark.

EXTENDED
Specifies that all aggregate activity data about the activities associated with this workload is to be sent to the statistics event monitor, if one is active. This includes all basic aggregate activity data plus:

- Activity data manipulation language (DML) estimated cost histogram
- Activity DML inter-arrival time histogram

NONE
Specifies that no aggregate activity data is to be collected for this workload.

COLLECT AGGREGATE UNIT OF WORK DATA
Specifies that aggregate unit of work data about the units of work associated with this workload is to be sent to the statistics event monitor, if one is active. This information is collected periodically on an interval that is specified by the **wlm_collect_int** database configuration parameter. The default when COLLECT AGGREGATE UNIT OF WORK DATA is not specified is COLLECT AGGREGATE UNIT OF WORK DATA NONE.

BASE
Specifies that basic aggregate unit of work data about the units of work associated with this workload is to be sent to the statistics event monitor, if one is active. Basic aggregate unit of work includes:

- Unit of work lifetime histogram

NONE
Specifies that no aggregate unit of work data is to be collected for this workload.

COLLECT LOCK TIMEOUT DATA
Specifies that data about lock timeout events that occur within this workload is sent to the applicable event monitor when the lock event occurs. The lock timeout data is collected on all members. The default is COLLECT LOCK TIMEOUT DATA WITHOUT HISTORY. This setting works in conjunction with the **mon_locktimeout** database configuration parameter setting. The setting that produces the most detailed output is honored.

WITHOUT HISTORY
Specifies that data about lock events that occur within this workload is sent to any active locking event monitor when the lock event occurs. Past activity history and input values are not sent to the event monitor.

NONE
Specifies that lock timeout data for the workload is not collected at any member.

WITH HISTORY
Specifies to collect past activity history in the current unit of work for all of this type of lock events. The activity history buffer will wrap after the maximum size limit is used.

The default limit on the number of past activities to be kept by any one application is 250. If the number of past activities is greater than the limit, only the newest activities are reported. This default value can be overridden using the registry variable DB2_MAX_INACT_STMTS to specify a different value. You can choose a different value for the limit to increase or reduce the amount of system monitor heap used for past activity information.

AND VALUES
Specifies that input data values are to be sent to any active locking event monitor for those activities that have them. These data values will not include LOB data, LONG VARCHAR data, LONG VARGRAPHIC data, structured type data, or XML data. For SQL statements compiled using the REOPT ALWAYS bind option, there will be no REOPT compilation or statement execution data values provided in the event information.

COLLECT DEADLOCK DATA
Specifies that data about deadlock events that occur within this workload is sent to any active locking event monitor when the lock event occurs. The deadlock data is collected on all members. The default

is COLLECT DEADLOCK DATA WITHOUT HISTORY. This setting is only honored if the **mon_deadlock** database configuration parameter is not set to NONE.

WITHOUT HISTORY
Specifies that data about lock events that occur within this workload is sent to any active locking event monitor when the lock event occurs. Past activity history and input values are not sent to the event monitor.

WITH HISTORY
Specifies to collect past activity history in the current unit of work for all of this type of lock events. The activity history buffer will wrap after the maximum size limit is used.

The default limit on the number of past activities to be kept by any one application is 250. If the number of past activities is greater than the limit, only the newest activities are reported. This default value can be overridden using the registry variable DB2_MAX_INACT_STMTS to specify a different value. You can choose a different value for the limit to increase or reduce the amount of system monitor heap used for past activity information.

AND VALUES
Specifies that input data values are to be sent to any active locking event monitor for those activities that have them. These data values will not include LOB data, LONG VARCHAR data, LONG VARGRAPHIC data, structured type data, or XML data. For SQL statements compiled using the REOPT ALWAYS bind option, there will be no REOPT compilation or statement execution data values provided in the event information.

COLLECT LOCK WAIT DATA
Specifies that data about lock wait events that occur within this workload is sent to any active locking event monitor when the lock has not been acquired within *wait-time*. The default is COLLECT LOCK WAIT DATA NONE with a default *wait-time* value of 0 microseconds. This setting works in conjunction with the **mon_lockwait** and **mon_lw_thresh** database configuration parameters. The setting that produces the most detailed output is honored.

NONE
Specifies that the lock wait event for the workload is not collected at any member.

FOR LOCKS WAITING MORE THAN *wait-time* (SECONDS | MICROSECONDS) | 1 SECOND
Specifies that data about lock wait events that occur within this workload is sent to any active locking event monitor when the lock has not been acquired within *wait-time*.

This value can be any non-negative integer. Use a valid duration keyword to specify an appropriate unit of time for *wait-time*. The minimum valid value for the *wait-time* parameter is 1000 microseconds.

WITH HISTORY
Specifies to collect past activity history in the current unit of work for all of this type of lock events. The activity history buffer will wrap after the maximum size limit is used.

The default limit on the number of past activities to be kept by any one application is 250. If the number of past activities is greater than the limit, only the newest activities are reported. This default value can be overridden using the registry variable DB2_MAX_INACT_STMTS to specify a different value. You can choose a different value for the limit to increase or reduce the amount of system monitor heap used for past activity information.

AND VALUES
Specifies that input data values are to be sent to any active locking event monitor for those activities that have them. These data values will not include LOB data, LONG VARCHAR data, LONG VARGRAPHIC data, structured type data, or XML data. For SQL statements compiled using the REOPT ALWAYS bind option, there will be no REOPT compilation or statement execution data values provided in the event information.

COLLECT UNIT OF WORK DATA
Specifies that data about each transaction associated with this workload is to be sent to the unit of work event monitor, if any are active, when the unit of work ends. The default, when COLLECT UNIT OF WORK DATA is not specified, is COLLECT UNIT OF WORK DATA NONE. The default, when COLLECT

UNIT OF WORK DATA is specified, is COLLECT UNIT OF WORK DATA BASE. If the **mon_uow_data** database configuration parameter is set to BASE, it takes precedence over the COLLECT UNIT OF WORK DATA parameter. A value of NONE for the **mon_uow_data** indicates that the COLLECT UNIT OF WORK DATA parameters of individual workloads is used.

NONE
Specifies that no unit of work data for transactions associated with this workload is sent to the unit of work event monitor. The default is COLLECT UNIT OF WORK DATA NONE.

BASE
Specifies that base level of data for transactions associated with this workload is sent to the unit of work event monitors.

Some of the information reported in a unit of work event are system level request metrics. The collection of these metrics is controlled independently from the collection of the unit of work data. The request metrics are controlled with the COLLECT REQUEST METRICS clause on superclass, or using the **mon_req_metrics** database configuration parameter. The service super class which the workload is associated with, or the service super class of the service subclass which the workload is associated with, must have the collection of request metrics enabled in order for the request metrics to be present in the unit of work event. If the request metrics collection is not enabled, the value of the request metrics will be zero.

INCLUDE PACKAGE LIST
Specifies that base level of data and the package list for transactions associated with this workload are sent to the unit of work event monitor.

The size of the collected package list is determined by the value of the **mon_pkglist_sz** database configuration parameter. If this value is 0, then the package list is not collected even if the PACKAGE LIST option is specified.

In a partitioned database environment, the package list is only available on the coordinator member. The BASE level will be collected on remote members.

Some of the information reported in a unit of work event are system level request metrics. The collection of these metrics is controlled independently from the collection of the unit of work data. The request metrics are controlled with the COLLECT REQUEST METRICS clause on superclass, or using the **mon_req_metrics** database configuration parameter. The service super class which the workload is associated with, or the service super class of the service subclass which the workload is associated with, must have the collection of request metrics enabled in order for the request metrics to be present in the unit of work event. If the request metrics collection is not enabled, the value of the request metrics will be zero.

INCLUDE EXECUTABLE LIST
Specifies that executable ID list will be collected for a unit of work together with base level of data and sent to the unit of work event monitor.

histogram-template-clause
Specifies the histogram templates to use when collecting aggregate activity data for activities executing in the workload.

ACTIVITY LIFETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the duration, in milliseconds, of database activities running in the workload during a specific interval. This time includes both time queued and time executing. The default is SYSDEFAULTHISTOGRAM. This information is collected only when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

ACTIVITY QUEUETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, that database activities running in the workload are queued during a specific interval. The default is SYSDEFAULTHISTOGRAM. This information is collected only when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option.

ACTIVITY EXECUTETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, that database activities running in the workload are executing during a specific interval. This time does not include the time spent queued. Activity execution time is collected in this histogram at the coordinator member only. The time does not include idle time. Idle time is the time between the execution of requests belonging to the same activity when no work is being done. An example of idle time is the time between the end of opening a cursor and the start of fetching from that cursor. The default is SYSDEFAULTHISTOGRAM. This information is collected only when the COLLECT AGGREGATE ACTIVITY DATA clause is specified, with either the BASE or EXTENDED option. Only activities at nesting level 0 are considered for inclusion in the histogram.

ACTIVITY ESTIMATEDCOST HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the estimated cost, in timerons, of DML activities running in the workload. The default is SYSDEFAULTHISTOGRAM. This information is collected only when the COLLECT AGGREGATE ACTIVITY DATA clause is specified with the EXTENDED option. Only activities at nesting level 0 are considered for inclusion in the histogram.

ACTIVITY INTERARRIVALTIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the length of time, in milliseconds, between the arrival of one DML activity into this workload and the arrival of the next DML activity into this workload. The default is SYSDEFAULTHISTOGRAM. This information is collected only when the COLLECT AGGREGATE ACTIVITY DATA clause is specified with the EXTENDED option.

UOW LIFETIME HISTOGRAM TEMPLATE *template-name*
Specifies the template that describes the histogram used to collect statistical data about the duration, in milliseconds, of units of work running in the workload during a specific interval. The default is SYSDEFAULTHISTOGRAM. This information is collected only when the COLLECT AGGREGATE UNIT OF WORK DATA clause is specified with the BASE option.

Rules

- A workload management (WLM)-exclusive SQL statement must be followed by a COMMIT or a ROLLBACK statement (SQLSTATE 5U021). WLM-exclusive SQL statements are:
  - CREATE HISTOGRAM TEMPLATE, ALTER HISTOGRAM TEMPLATE, or DROP (HISTOGRAM TEMPLATE)
  - CREATE SERVICE CLASS, ALTER SERVICE CLASS, or DROP (SERVICE CLASS)
  - CREATE THRESHOLD, ALTER THRESHOLD, or DROP (THRESHOLD)
  - CREATE WORK ACTION SET, ALTER WORK ACTION SET, or DROP (WORK ACTION SET)
  - CREATE WORK CLASS SET, ALTER WORK CLASS SET, or DROP (WORK CLASS SET)
  - CREATE WORKLOAD, ALTER WORKLOAD, or DROP (WORKLOAD)
  - GRANT (Workload Privileges) or REVOKE (Workload Privileges)
- A WLM-exclusive SQL statement cannot be issued within a global transaction (SQLSTATE 51041) such as, for example, an XA transaction.

Notes

- Changes are written to the system catalog, but do not take effect until they are committed, even for the connection that issues the statement.
- Only one uncommitted WLM-exclusive SQL statement at a time is allowed across all partitions. If an uncommitted WLM-exclusive SQL statement is executing, subsequent WLM-exclusive SQL statements will wait until the current WLM-exclusive SQL statement commits or rolls back.
- When a database connection is established, the database manager looks for a matching workload based on the connection attributes that were specified in the POSITION clause (in order of specification). If a matching workload is found, the database manager checks whether the current session user has USAGE

privilege on that workload. If the session user does not have USAGE privilege on the workload, the database manager looks for the next matching workload. If the session user has USAGE privilege on this workload, the connection is associated with the workload. If a matching workload is not found, the connection is associated with the default user workload, SYSDEFAULTUSERWORKLOAD. If the session user does not have USAGE privilege on SYSDEFAULTUSERWORKLOAD, an error is returned (SQLSTATE 42501).

- The workload association is re-evaluated at the beginning of each new unit of work if the database manager detects one of the following conditions.
  - The connection attributes have changed. This can happen if any of the following events has occurred:
    - The set client information API (`sqleseti`) has been invoked and it changed the connection attributes that were included in the workload definition. Note that although the client information can be set by the end user so that it could initiate a workload re-evaluation, the workload remapping itself cannot happen if the session user does not have the USAGE privilege on the workload.
    - The SET SESSION AUTHORIZATION statement has been invoked and it changed the current session user.
    - The roles that are available to a session user have changed.
  - A workload is created.
  - A workload is dropped.
  - A workload is altered.
  - The USAGE privilege on a workload is granted to a user, group, or role.
  - The USAGE privilege on a workload is revoked from a user, group, or role.

  If the workload re-evaluation results in no workload reassignment, the current workload occurrence continues to run; that is, a new workload occurrence will not be started.
- A connection cannot be reassigned to a different workload when an activity is still active. Examples of such activities are a load operation, an executing procedure, or statements that maintain resources across multiple units of work, such as an open WITH HOLD cursor. The current workload occurrence continues to run until all executing activities complete. Workload reassignment occurs at the beginning of the next unit of work.
- After a service class has been referenced by a workload, it cannot be dropped until it is no longer referenced by any workload. Either of the following actions can be taken to remove a service class reference from a workload:
  - Alter the workload to change the service class name
  - Drop the workload
- After a role has been referenced by a workload, it cannot be dropped until it is no longer referenced by any workload. Either of the following actions can be taken to remove a role reference from a workload:
  - Alter the workload to remove the role
  - Drop the workload
- ***Privileges***: The USAGE privilege is not granted to any user, group, or role when a workload is created. To enable use of a workload, grant USAGE privilege on that workload to a user, a group, or a role using the GRANT USAGE ON WORKLOAD statement.
- ***Syntax alternatives***: The following are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - DATABASE PARTITION can be specified in place of MEMBER, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
  - DATABASE PARTITIONS can be specified in place of MEMBERS, except when the DB2_ENFORCE_MEMBER_SYNTAX registry variable is set to ON.
  - COLLECT UNIT OF WORK DATA PACKAGE LIST can be specified in place of COLLECT UNIT OF WORK DATA BASE INCLUDE PACKAGE LIST.

Examples

- *Example 1:* Create a workload named CAMPAIGN for requests that are submitted by a session user belonging to group FINANCE. These requests are to be executed in the default user service class SYSDEFAULTUSERCLASS.

```
CREATE WORKLOAD CAMPAIGN
  SESSION_USER GROUP ('FINANCE')
```

- *Example 2:* Create a workload named PAYROLL for a session user with role HR that has the CURRENT CLIENT_APPLNAME special register set to SALARYSYS. Units of work associated with this workload are to be executed in service class MEDIUMSC that is under the service superclass HRSC. When a workload is chosen at run time, this workload should be evaluated only after the workload CAMPAIGN has been evaluated and determined to not match.

```
CREATE WORKLOAD PAYROLL
  SESSION_USER ROLE ('HR')
  CURRENT CLIENT_APPLNAME ('SALARYSYS') SERVICE CLASS MEDIUMSC
  UNDER HRSC POSITION AFTER CAMPAIGN
```

- *Example 3:* An occurrence of workload CAMPAIGN (from example 1) is currently running on the system. Create a workload named NEWCAMPAIGN, also for requests that are submitted by a session user belonging to group FINANCE, but only those requests submitted through application DB2BP.EXE. Requests associated with this workload are to be executed in service class MARKETINGSC. NEWCAMPAIGN should be evaluated before CAMPAIGN.

```
CREATE WORKLOAD NEWCAMPAIGN
  SESSION_USER GROUP ('FINANCE')
  APPLNAME ('DB2BP.EXE') SERVICE CLASS MARKETINGSC
  POSITION BEFORE CAMPAIGN
```

  The running workload occurrence of CAMPAIGN continues to run until the current unit of work completes, at which time a workload re-evaluation takes place, and the connection could then be remapped to workload NEWCAMPAIGN.

- *Example 4:* Create a workload named REPORTS for requests that are submitted through application appl1, appl2, or appl3 by system user BOB or MARY.

```
CREATE WORKLOAD REPORTS
  APPLNAME ('appl1', 'appl2', 'appl3')
  SYSTEM_USER ('BOB', 'MARY')
```

- *Example 5:* Assuming a lock event monitor called PAYROLL exists and is active, create lock event records with statement history for lock timeout events that occur within the workload EMPLOYEES.

```
CREATE WORKLOAD EMPLOYEES
  APPLNAME ("app1", "app2")
  COLLECT LOCK TIMEOUT DATA WITH HISTORY
```

- *Example 6:* Assuming a lock event monitor called PAYROLL exists and is active, create lock event records for only deadlock and lock timeout events that occur within the workload FINANCE on all partitions.

```
CREATE WORKLOAD FINANCE
  APPLNAME ("app1", "app2")
  COLLECT DEADLOCK DATA
  COLLECT LOCK TIMEOUT DATA
```

- *Example 7:* Assuming a lock event monitor called PAYROLL exists and is active, create lock event records with statement history and values for deadlock events that occur within the workload MANAGERS.

```
CREATE WORKLOAD MANAGERS
  APPLNAME ("app1", "app2")
  COLLECT DEADLOCK DATA WITH HISTORY AND VALUES
```

- *Example 8:* Assuming a lock event monitor called PAYROLL exists and is active, create lock event records with statement history for locks that are acquired after waiting 5000 milliseconds within the MANAGERS workload.

```
CREATE WORKLOAD MANAGERS
  APPLNAME ("app1", "app2")
  COLLECT LOCK WAIT DATA FOR LOCKS WAITING MORE THAN 5 SECONDS WITH HISTORY
```

- *Example 9*: Create a workload named ACCRECS for all accounts receivable applications that share a similar name (*accrec01, accrec02 ... accrec15*) and assign them to the service class ACCOUNTNGSC. Application names are identified through the APPLNAME connection attribute with the help of a wild card (*) and do not need to be specified individually.

```
CREATE WORKLOAD ACCRECS
  SESSION_USER GROUP ('ACCOUNTING')
  APPLNAME ('accrec*')
  SERVICE CLASS ACCOUNTNGSC
```

- *Example 10:* Create a workload named CAMPAIGN for requests submitted through the application appl1, and have unit of work data collected and sent to any active unit of work event monitors.

```
CREATE WORKLOAD CAMPAIGN
  APPLNAME ('appl1')
  COLLECT UNIT OF WORK DATA BASE
```

- *Example 11:* The following statements show how you can specify the different address value formats supported by the ADDRESS connection attribute when creating a workload.
  - To specify a secure domain name:

```
CREATE WORKLOAD DOMAINWORKLOAD
  ADDRESS ('aviator.example.com')
```

  - To specify a IPv4 address value:

```
CREATE WORKLOAD IPWORKLOAD1
  ADDRESS ('192.0.2.11')
```

  - To specify a IPv6 address value (long format):

```
CREATE WORKLOAD IPWORKLOAD2
  ADDRESS ('2001:db8:519:13:204:acff:fe57:6135')
```

  - To specify a IPv6 address value (short format):

```
CREATE WORKLOAD IPWORKLOAD3
  ADDRESS ('2001:db8::202:55ff:fe9a:6eee')
```

CREATE WRAPPER

The CREATE WRAPPER statement registers a wrapper with a federated server. A wrapper is a mechanism by which a federated server can interact with certain types of data sources.

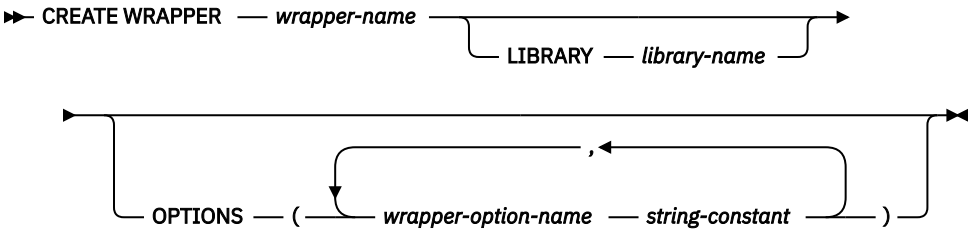
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include DBADM authority.

Syntax



Description

wrapper-name
Names the wrapper. It can be:

- A predefined name. If a predefined name is specified, the federated server automatically assigns a default value to *library-name*.
- A user-supplied name. If a user-supplied name is provided, it is necessary to also specify the appropriate *library-name* to be used with that wrapper and operating system.

LIBRARY *library-name*
The name of the file that contains the wrapper library module. This option is required when a user-supplied wrapper name is used, and cannot be specified when a predefined wrapper name is used. The library name must be enclosed in single quotation marks.

The library name can be specified either as an absolute path name or as a base name (without the path). If a base name is specified, the library must reside in the following subdirectory of your database's installation path:

Operating system	Subdirectory for the wrapper library module
Linux AIX	`lib`
Windows	`bin`

OPTIONS
Specify configuration options for the wrapper to be created. Which options you can specify depends on the data source of the object for which a wrapper is being created. For a list of data sources and the wrapper options that apply to each, see Data source options. Each option value is a character string constant that must be enclosed in single quotation marks.

Notes

- **Syntax alternatives:** The following syntax is supported for compatibility with previous versions of Db2:
  - ADD can be specified before *wrapper-option-name string-constant*.

Examples

1. Register the NET8 wrapper on a federated server to access Oracle data sources. *NET8* is the predefined name for the wrapper that you can use to access Oracle data sources.

```
CREATE WRAPPER NET8
```

2. Register a wrapper on a Db2 federated server that uses the Linux operating system to access ODBC data sources. Assign the name *odbc* to the wrapper that is being registered in the federated database.

The full path of the library that contains the ODBC Driver Manager is defined in the wrapper option *MODULE '/usr/lib/odbc.so'*.

```
CREATE WRAPPER odbc OPTIONS (MODULE '/usr/lib/odbc.so')
```

3. Register a wrapper on a Db2 federated server that uses the Windows operating system to access ODBC data sources. The library name for the ODBC wrapper is *'db2rcodbc.dll'*.

```
CREATE WRAPPER odbc LIBRARY 'db2rcodbc.dll'
```

DECLARE CURSOR

The DECLARE CURSOR statement defines a cursor.

Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is not an executable statement and cannot be dynamically prepared.

When invoked using the command line processor, additional options can be specified.

For more information, refer to "Using command line SQL statements and XQuery statements" in *Command Reference*.

Authorization

The term "SELECT statement of the cursor" is used to specify the authorization rules. The SELECT statement of the cursor is one of the following statements:

- The prepared select-statement identified by *statement-name*
- The specified *select-statement*

The privileges held by the authorization ID of the statement must include the privileges necessary to execute the *select-statement*. See the Authorization section in "SQL queries".

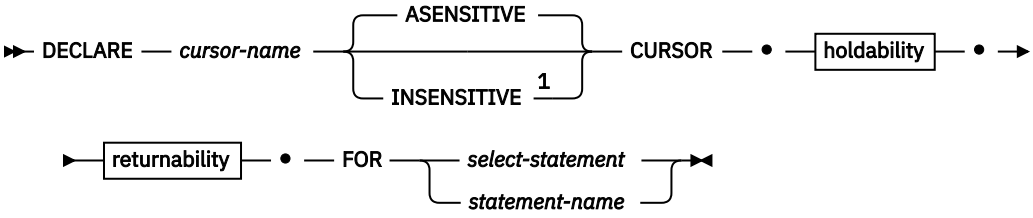
If *statement-name* is specified:

- The authorization ID of the statement is the runtime authorization ID.
- The authorization check is performed when the SELECT-statement is prepared.
- The cursor cannot be opened unless the SELECT-statement is in a prepared state.

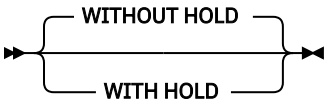
If *select-statement* is specified:

- GROUP privileges are not checked.
- The authorization ID of the statement is the authorization ID specified during program preparation.

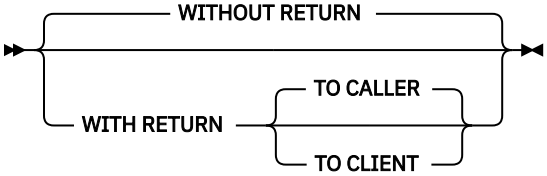
Syntax



holdability



returnability



Notes:

[1] This option can be used only in the context of a compound SQL (compiled) statement

Description

cursor-name
Specifies the name of the cursor created when the source program is run. The name must not be the same as the name of another cursor declared in the source program. The cursor must be opened before use.

ASENSITIVE or INSENSITIVE
Specifies whether the cursor is asensitive or insensitive to changes.

ASENSITIVE
Specifies that the cursor should be as sensitive as possible to insert, update, or delete operations made to the rows underlying the result table, depending on how the *select-statement* is optimized. This option is the default.

INSENSITIVE
Specifies that the cursor does not have sensitivity to insert, update, or delete operations that are made to the rows underlying the result table. If INSENSITIVE is specified, the cursor is read-only and the result table is materialized when the cursor is opened. As a result, the size of the result table, the order of the rows, and the values for each row do not change after the cursor is opened. The SELECT statement cannot contain a FOR UPDATE clause, and the cursor cannot be used for positioned updates or deletes.

WITHOUT HOLD or WITH HOLD
Specifies whether or not the cursor should be prevented from being closed as a consequence of a commit operation.

WITHOUT HOLD
Does not prevent the cursor from being closed as a consequence of a commit operation. This is the default.

WITH HOLD
Maintains resources across multiple units of work. The effect of the WITH HOLD cursor attribute is as follows:

- For units of work ending with COMMIT:
  - Open cursors defined WITH HOLD remain open. The cursor is positioned before the next logical row of the results table.

    If a DISCONNECT statement is issued after a COMMIT statement for a connection with WITH HOLD cursors, the held cursors must be explicitly closed or the connection will be assumed to have performed work (simply by having open WITH HELD cursors even though no SQL statements were issued) and the DISCONNECT statement will fail.
  - All locks are released, except locks protecting the current cursor position of open WITH HOLD cursors. The locks held include the locks on the table, and for parallel environments, the locks

on rows where the cursors are currently positioned. Locks on packages and dynamic SQL sections (if any) are held.

  - Valid operations on cursors defined WITH HOLD immediately following a COMMIT request are:
    - FETCH: Fetches the next row of the cursor.
    - CLOSE: Closes the cursor.
  - UPDATE and DELETE CURRENT OF CURSOR are valid only for rows that are fetched within the same unit of work.
  - LOB locators are freed.
  - The set of rows modified by:
    - A data change statement
    - Routines that modify SQL data embedded within open WITH HOLD cursors

    is committed.
- For units of work ending with ROLLBACK:
  - All open cursors are closed.
  - All locks acquired during the unit of work are released.
  - LOB locators are freed.
- For special COMMIT case:
  - Packages can be recreated either explicitly, by binding the package, or implicitly, because the package has been invalidated and then dynamically recreated the first time it is referenced. All held cursors are closed during package rebind. This might result in errors during subsequent execution.

WITHOUT RETURN or WITH RETURN

Specifies whether or not the result table of the cursor is intended to be used as a result set that will be returned from a procedure.

WITHOUT RETURN

Specifies that the result table of the cursor is not intended to be used as a result set that will be returned from a procedure.

WITH RETURN

Specifies that the result table of the cursor is intended to be used as a result set that will be returned from a procedure. WITH RETURN is relevant only if the DECLARE CURSOR statement is contained with the source code for a procedure. In other cases, the precompiler might accept the clause, but it has no effect.

Within an SQL procedure, cursors declared using the WITH RETURN clause that are still open when the SQL procedure ends, define the result sets from the SQL procedure. All other open cursors in an SQL procedure are closed when the SQL procedure ends. Within an external procedure (one not defined using LANGUAGE SQL), the default for all cursors is WITH RETURN TO CALLER. Therefore, all cursors that are open when the procedure ends will be considered result sets. Cursors that are returned from a procedure cannot be declared as scrollable cursors.

TO CALLER

Specifies that the cursor can return a result set to the caller. For example, if the caller is another procedure, the result set is returned to that procedure. If the caller is a client application, the result set is returned to the client application.

TO CLIENT

Specifies that the cursor can return a result set to the client application. This cursor is invisible to any intermediate nested procedures. If a function, method, or trigger called the procedure either directly or indirectly, result sets cannot be returned to the client and the cursor will be closed after the procedure finishes.

select-statement
Identifies the SELECT statement of the cursor. The *select-statement* must not include parameter markers, but can include references to host variables. The declarations of the host variables must precede the DECLARE CURSOR statement in the source program.

statement-name
The SELECT statement of the cursor is the prepared SELECT statement identified by the *statement-name* when the cursor is opened. The *statement-name* must not be identical to a *statement-name* specified in another DECLARE CURSOR statement of the source program.

For an explanation of prepared SELECT statements, see "PREPARE".

Notes

- A program called from another program, or from a different source file within the same program, cannot use the cursor that was opened by the calling program.
- Unnested procedures, with LANGUAGE other than SQL, will have WITH RETURN TO CALLER as the default behavior if DECLARE CURSOR is specified without a WITH RETURN clause, and the cursor is left open in the procedure. This provides compatibility with procedures from previous versions that allow procedures to return result sets to applicable client applications. To avoid this behavior, close all cursors opened in the procedure.
- If the SELECT statement of a cursor contains CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP, all references to these special registers will yield the same respective datetime value on each FETCH. This value is determined when the cursor is opened.
- For more efficient processing of data, the database manager can block data for read-only cursors when retrieving data from a remote server. The use of the FOR UPDATE clause helps the database manager decide whether a cursor is updatable or not. Updatability is also used to determine the access path selection as well. If a cursor is not going to be used in a Positioned UPDATE or DELETE statement, it should be declared as FOR READ ONLY.
- A cursor in the open state designates a result table and a position relative to the rows of that table. The table is the result table specified by the SELECT statement of the cursor.
- A cursor is *deletable* if each of the following conditions is true:
  - Each FROM clause of the outer fullselect identifies only one base table or deletable view (cannot identify a nested or common table expression or a nickname) without use of the OUTER clause
  - The outer fullselect does not include a VALUES clause
  - The outer fullselect does not include a GROUP BY clause or HAVING clause
  - The outer fullselect does not include column functions in the select list
  - The outer fullselect does not include SET operations (UNION, EXCEPT, or INTERSECT) with the exception of UNION ALL
  - The outer fullselect does not contain a FOR SYSTEM_TIME period specification.
  - The select list of the outer fullselect does not include DISTINCT
  - The outer fullselect does not include an ORDER BY clause (even if the ORDER BY clause is nested in a view), and the FOR UPDATE clause has not been specified
  - The select-statement does not include a FOR READ ONLY clause
  - The FROM clause of the outer fullselect does not include a *data-change-table-reference*
  - One or more of the following conditions is true:
    - The FOR UPDATE clause is specified
    - The cursor is statically defined, unless the STATICREADONLY bind option is YES
    - The LANGLEVEL bind option is MIA or SQL92E

  A column in the select list of the outer fullselect associated with a cursor is *updatable* if each of the following conditions is true:

- The cursor is deletable
- The column resolves to a column of the base table
- The LANGLEVEL bind option is MIA, SQL92E or the select-statement includes the FOR UPDATE clause (the column must be specified explicitly or implicitly in the FOR UPDATE clause)

A cursor is *read-only* if it is not deletable.

A cursor is *ambiguous* if each of the following conditions is true:

- The select-statement is dynamically prepared
- The select-statement does not include either the FOR READ ONLY clause or the FOR UPDATE clause
- The LANGLEVEL bind option is SAA1
- The cursor otherwise satisfies the conditions of a deletable cursor

An ambiguous cursor is considered read-only if the BLOCKING bind option is ALL, otherwise it is considered updatable.

- Cursors in procedures that are called by application programs written using CLI can be used to define result sets that are returned directly to the client application. Cursors in SQL procedures can also be returned to a calling SQL procedure only if they are defined using the WITH RETURN clause.
- Cursors declared in routines that are invoked directly or indirectly from a cursor declared WITH HOLD, do not inherit the WITH HOLD option. Thus, unless the cursor in the routine is explicitly defined WITH HOLD, a COMMIT in the application will close it.

  Consider the following application and two UDFs:

```
Application:

   DECLARE APPCUR CURSOR WITH HOLD FOR SELECT UDF1() ...
   OPEN APPCUR
   FETCH APPCUR ...
   COMMIT

UDF1:

   DECLARE UDF1CUR CURSOR FOR SELECT UDF2() ...
   OPEN UDF1CUR
   FETCH UDF1CUR ...

UDF2:

   DECLARE UDF2CUR CURSOR WITH HOLD FOR SELECT UDF2() ...
   OPEN UDF2CUR
   FETCH UDF2CUR ...
```

  After the application fetches cursor APPCUR, all three cursors are open. When the application issues the COMMIT statement, APPCUR remains open, because it was declared WITH HOLD. In UDF1, however, the cursor UDF1CUR is closed, because it was not defined with the WITH HOLD option. When the cursor UDF1CUR is closed, all routine invocations in the corresponding select-statement complete (receiving a final call, if so defined). UDF2 completes, which causes UDF2CUR to close.

Examples

Example 1: The DECLARE CURSOR statement associates the cursor name C1 with the results of the SELECT.

```
  EXEC SQL DECLARE C1 CURSOR FOR
    SELECT DEPTNO, DEPTNAME, MGRNO
    FROM DEPARTMENT
    WHERE ADMRDEPT = 'A00';
```

Example 2: Assume that the EMPLOYEE table has been altered to add a generated column, WEEKLYPAY, that calculates the weekly pay based on the yearly salary. Declare a cursor to retrieve the system-generated column value from a row to be inserted.

```
EXEC SQL DECLARE C2 CURSOR FOR
  SELECT E.WEEKLYPAY
  FROM NEW TABLE
    (INSERT INTO EMPLOYEE
     (EMPNO, FIRSTNME, MIDINIT, LASTNAME, EDLEVEL, SALARY)
     VALUES('000420', 'Peter', 'U', 'Bender', 16, 31842) AS E;
```

DECLARE GLOBAL TEMPORARY TABLE

The DECLARE GLOBAL TEMPORARY TABLE statement defines a temporary table for the current session.

The declared temporary table description does not appear in the system catalog. It is not persistent and cannot be shared with other sessions. Each session that defines a declared global temporary table of the same name has its own unique description of the temporary table. When the session terminates, the rows of the table are deleted, and the description of the temporary table is dropped.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

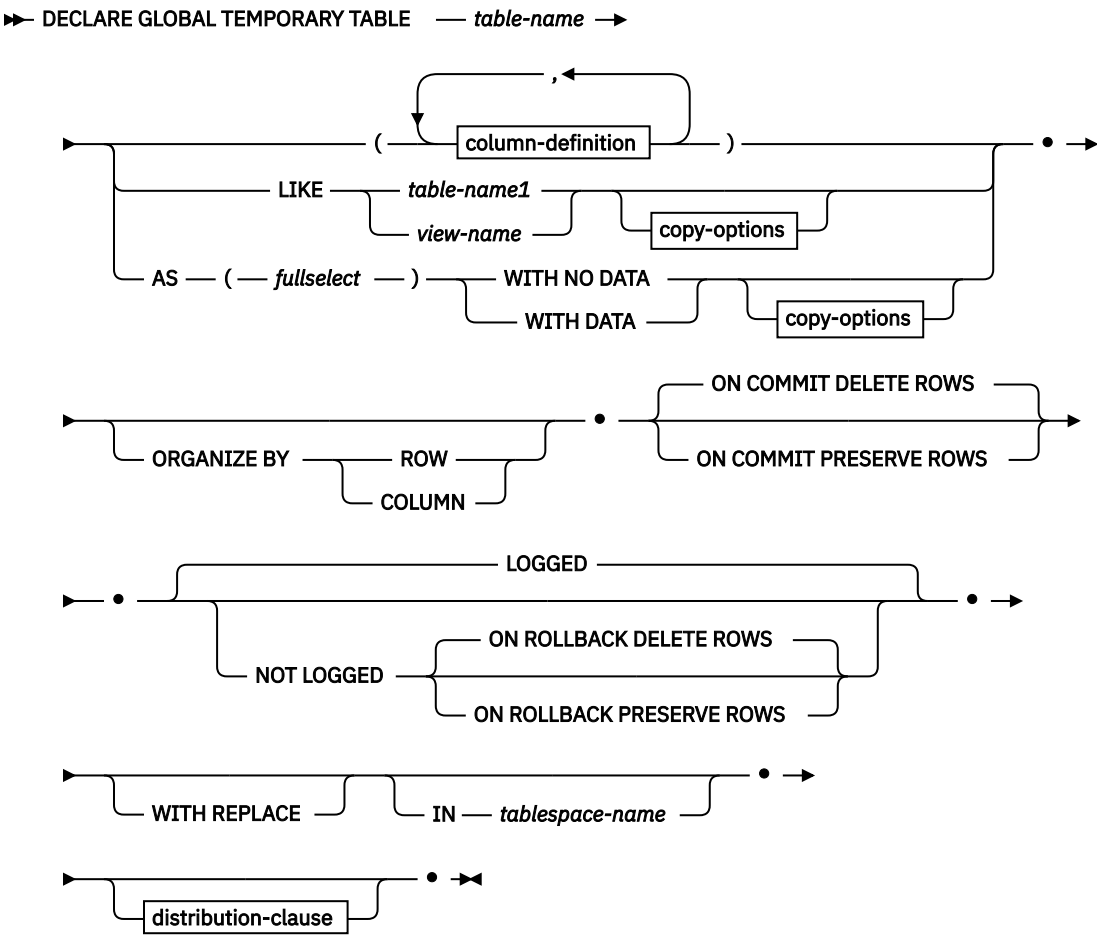
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- USE privilege on the USER TEMPORARY table space
- DBADM authority
- SYSADM authority
- SYSCTRL authority

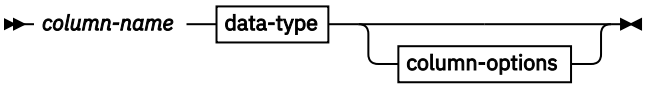
When defining a table using LIKE or a fullselect, the privileges held by the authorization ID of the statement must also include at least one of the following authorities on each identified table or view:

- SELECT privilege on the table or view
- SELECTIN privilege on the schema containing the table or view
- CONTROL privilege on the table or view
- DATAACCESS authority on the schema containing the table or view
- DATAACCESS authority

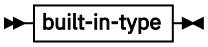
Syntax



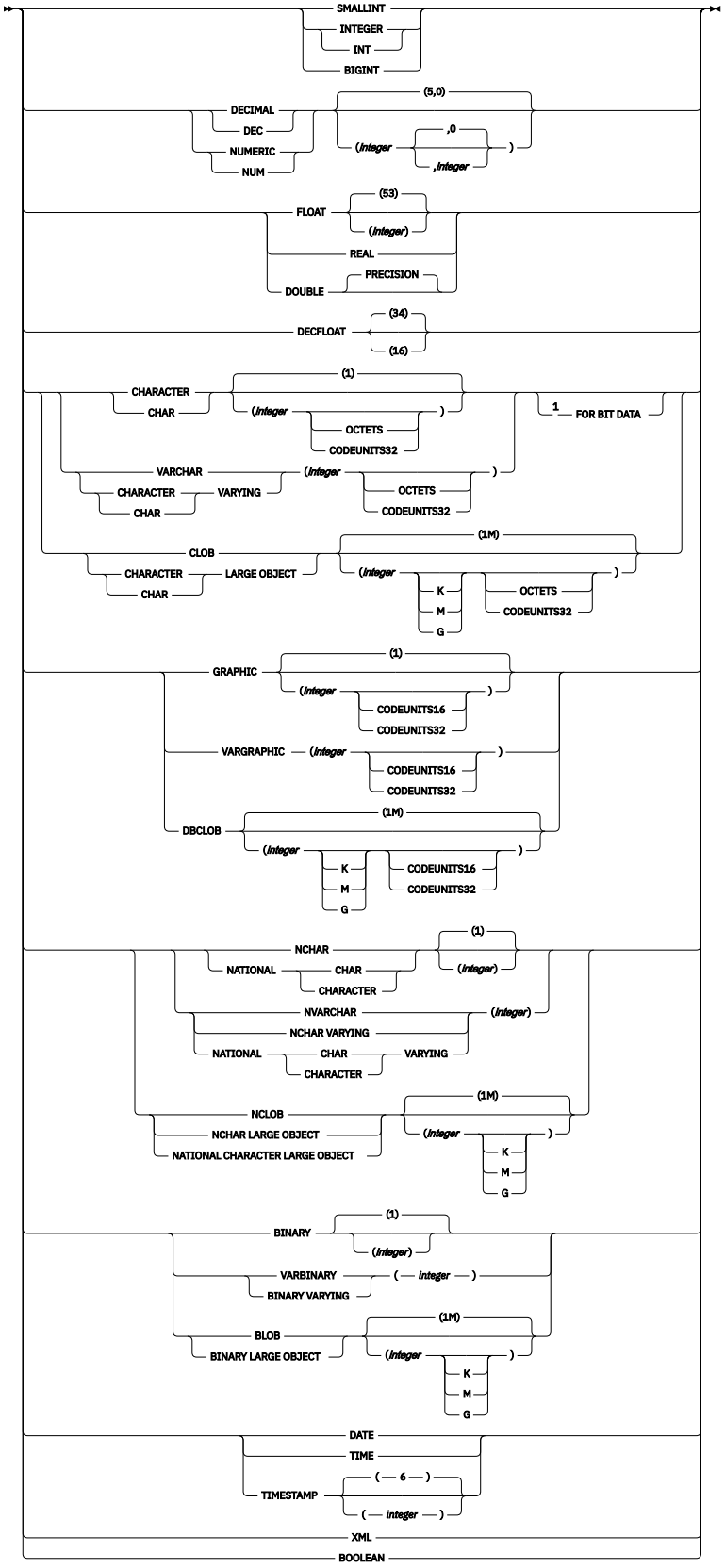
column-definition



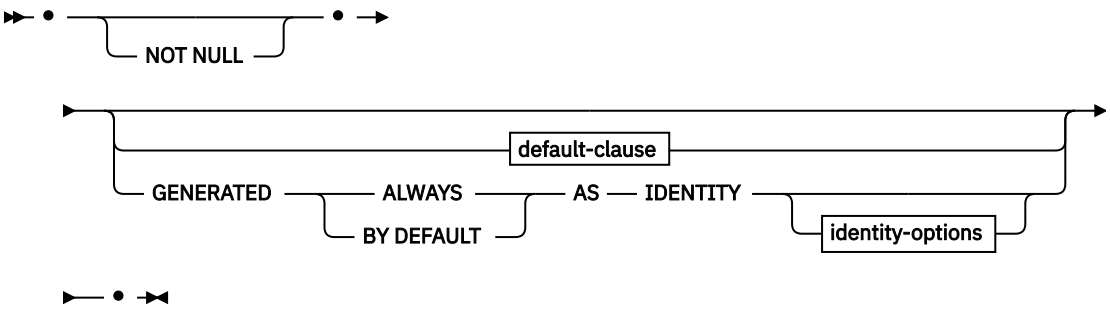
data-type



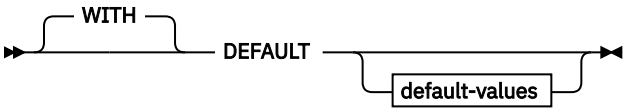
built-in-type



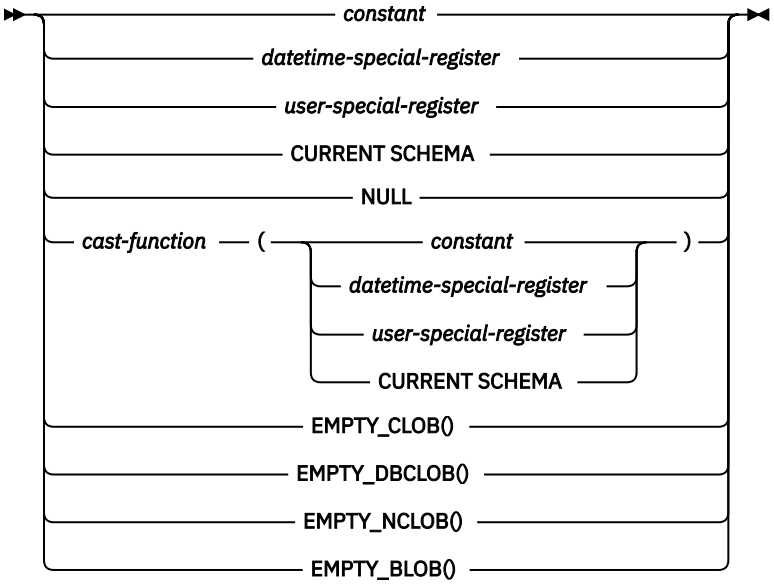
column-options



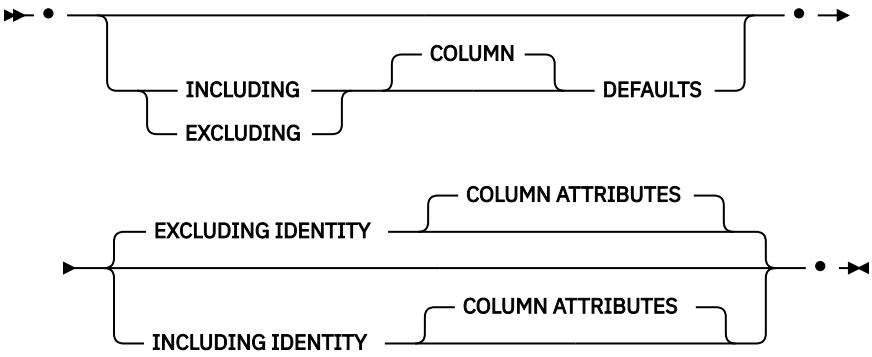
default-clause



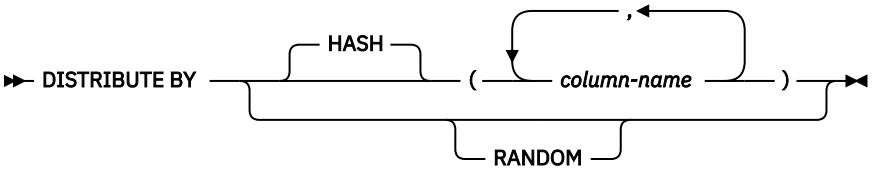
default-values



copy-options



distribution-clause



Notes:

[1] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

Description

table-name
Names the temporary table. The qualifier, if specified explicitly, must be SESSION, otherwise an error is returned (SQLSTATE 428EK). If the qualifier is not specified, SESSION is implicitly assigned.

Each session that defines a declared temporary table with the same *table-name* has its own unique description of that declared temporary table. The WITH REPLACE clause must be specified if *table-name* identifies a declared temporary table that already exists in the session (SQLSTATE 42710).

It is possible that a table, view, alias, or nickname already exists in the catalog, with the same name and the schema name SESSION. In this case:

- A declared temporary table *table-name* may still be defined without any error or warning
- Any references to SESSION.*table-name* will resolve to the declared temporary table rather than the SESSION.*table-name* already defined in the catalog.

column-definition
Defines the attributes of a column of the temporary table.

column-name
Names a column of the table. The name cannot be qualified, and the same name cannot be used for more than one column of the table (SQLSTATE 42711).

A row-organized table can have the following attributes:

- A 4K page size with a maximum of 500 columns, where the byte counts of the columns must not be greater than 4005.
- An 8K page size with a maximum of 1012 columns, where the byte counts of the columns must not be greater than 8101.
- A 16K page size with a maximum of 1012 columns, where the byte counts of the columns must not be greater than 16,293.
- A 32K page size with a maximum of 1012 columns, where the byte counts of the columns must not be greater than 32,677.

A column-organized table can have a maximum of 1012 columns, regardless of its page size. The byte count of each column must not exceed 32,677. Extended row size support does not apply to column-organized tables.

A created temporary table cannot have a row-begin column, row-end column, or a transaction-start-ID column.

For more details, see "Row Size" in “CREATE TABLE ” on page 1351.

data-type
Specifies the data type of the column

built-in-type
Specifies a built-in data type. See "CREATE TABLE" for a description of *built-in-type*.

A SYSPROC.DB2SECURITYLABEL data type cannot be specified for a declared temporary table.

column-options
Defines additional options related to the columns of the table.

NOT NULL
Prevents the column from containing null values. For specification of null values, see NOT NULL in “CREATE TABLE ” on page 1351.

default-clause
Specifies a default value for the column.

WITH
An optional keyword.

DEFAULT
Provides a default value in the event a value is not supplied on INSERT or is specified as DEFAULT on INSERT or UPDATE. If a default value is not specified following the DEFAULT keyword, the default value depends on the data type of the column as shown in "ALTER TABLE".

If the column is based on a column of a typed table, a specific default value must be specified when defining a default. A default value cannot be specified for the object identifier column of a typed table (SQLSTATE 42997).

If a column is defined using a distinct type, then the default value of the column is the default value of the source data type cast to the distinct type.

If a column is defined using a structured type, the *default-clause* cannot be specified (SQLSTATE 42842).

Omission of DEFAULT from a *column-definition* results in the use of the null value as the default for the column. If such a column is defined NOT NULL, then the column does not have a valid default.

default-values
Specific types of default values that can be specified are as follows.

constant
Specifies the constant as the default value for the column. The specified constant must:

- represent a value that could be assigned to the column in accordance with the rules of assignment
- not be a floating-point constant unless the column is defined with a floating-point data type
- be a numeric constant or a decimal floating-point special value if the data type of the column is a decimal floating-point. Floating-point constants are first interpreted as DOUBLE and then converted to decimal floating-point if the target column is DECFLOAT. For DECFLOAT(16) columns, decimal constants having precision greater than 16 digits will be rounded using the rounding modes specified by the CURRENT DECFLOAT ROUNDING MODE special register.
- not have nonzero digits beyond the scale of the column data type if the constant is a decimal constant (for example, 1.234 cannot be the default for a DECIMAL(5,2) column)
- be expressed with no more than 254 bytes including the quote characters, any introducer character such as the X for a hexadecimal constant, and characters from the fully qualified function name and parentheses when the constant is the argument of a *cast-function*

datetime-special-register
Specifies the value of the datetime special register (CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP) at the time of INSERT, UPDATE, or LOAD as the default for the column. The data type of the column must be the data type that corresponds to the special register specified (for example, data type must be DATE when CURRENT DATE is specified).

user-special-register
Specifies the value of the user special register (CURRENT USER, SESSION_USER, SYSTEM_USER) at the time of INSERT, UPDATE, or LOAD as the default for the column. The data type of the column must be a character string with a length not less than the length attribute of a user special register. Note that USER can be specified in place of SESSION_USER and CURRENT_USER can be specified in place of CURRENT USER.

CURRENT SCHEMA
Specifies the value of the CURRENT SCHEMA special register at the time of INSERT, UPDATE, or LOAD as the default for the column. If CURRENT SCHEMA is specified, the data type of the column must be a character string with a length greater than or equal to the length attribute of the CURRENT SCHEMA special register.

NULL
Specifies NULL as the default for the column. If NOT NULL was specified, DEFAULT NULL may be specified within the same column definition but will result in an error on any attempt to set the column to the default value.

cast-function
This form of a default value can only be used with columns defined as a distinct type, BLOB or datetime (DATE, TIME or TIMESTAMP) data type. For distinct type, with the exception of distinct types based on BLOB or datetime types, the name of the function must match the name of the distinct type for the column. If qualified with a schema name, it must be the same as the schema name for the distinct type. If not qualified, the schema name from function resolution must be the same as the schema name for the distinct type. For a distinct type based on a datetime type, where the default value is a constant, a function must be used and the name of the function must match the name of the source type of the distinct type with an implicit or explicit schema name of SYSIBM. For other datetime columns, the corresponding datetime function may also be used. For a BLOB or a distinct type based on BLOB, a function must be used and the name of the function must be BLOB with an implicit or explicit schema name of SYSIBM.

constant
Specifies a constant as the argument. The constant must conform to the rules of a constant for the source type of the distinct type or for the data type if not a distinct type. If the *cast-function* is BLOB, the constant must be a string constant.

datetime-special-register
Specifies CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP. The source type of the distinct type of the column must be the data type that corresponds to the specified special register.

user-special-register
Specifies CURRENT USER, SESSION_USER, or SYSTEM_USER. The data type of the source type of the distinct type of the column must be a string data type with a length of at least 8 bytes. If the *cast-function* is BLOB, the length attribute must be at least 8 bytes.

CURRENT SCHEMA
Specifies the value of the CURRENT SCHEMA special register. The data type of the source type of the distinct type of the column must be a character string with a length greater than or equal to the length attribute of the CURRENT SCHEMA special register. If the cast-function is BLOB, the length attribute must be at least 8 bytes.

EMPTY_CLOB(), EMPTY_DBCLOB(), or EMPTY_BLOB()
Specifies a zero-length string as the default for the column. The column must have the data type that corresponds to the result data type of the function.

If the value specified is not valid, an error is returned (SQLSTATE 42894).

IDENTITY and *identity-options*
For specification of identity columns, see IDENTITY and *identity-options* in "CREATE TABLE".

LIKE *table-name1* or *view-name* or *nickname*
Specifies that the columns of the table have exactly the same name and description as the columns of the identified table (*table-name1*), view (*view-name*), or nickname (*nickname*). The name specified after LIKE must identify a table, view, or nickname that exists in the catalog or a declared temporary table. A typed table or typed view cannot be specified (SQLSTATE 428EC). A protected table cannot

be specified (SQLSTATE 42962).A table that has a column defined as IMPLICITLY HIDDEN cannot be specified (SQLSTATE 560AE).

The use of LIKE is an implicit definition of *n* columns, where *n* is the number of columns in the identified table (including implicitly hidden columns), view, or nickname. The implicit definition depends on what is identified after LIKE.

- If a table is identified, then the implicit definition includes the column name, data type and nullability characteristic of each of the columns of *table-name1*. If EXCLUDING COLUMN DEFAULTS is not specified, then the column default is also included.
- If a view is identified, then the implicit definition includes the column name, data type, and nullability characteristic of each of the result columns of the fullselect defined in *view-name*. The data types of the view columns must be data types that are valid for columns of a table.
- If a nickname is identified, then the implicit definition includes the column name, data type, and nullability characteristic of each column of *nickname*.
- If a random distribution table using the random by generation method is identified, then the **RANDOM_DISTRIBUTION_KEY** column used for generation of random distribution values is not included. Unless the new table being created shares the same table distribution.

Column default and identity column attributes may be included or excluded, based on the *copy-attributes* clauses. The implicit definition does not include any other attributes of the identified table, view, or nickname. Thus the new table does not have any unique constraints, foreign key constraints, triggers, indexes, table partitioning keys, or distribution keys. The table is created in the table space implicitly or explicitly specified by the IN clause, and the table has any other optional clause only if the optional clause is specified.

When a table is identified in the LIKE clause and that table contains a ROW CHANGE TIMESTAMP column, the corresponding column of the new table inherits only the data type of the ROW CHANGE TIMESTAMP column. The new column is not considered to be a generated column.

If row or column level access control (RCAC) is enforced for *table-name1*, RCAC is not inherited by the new table.

AS (*fullselect*)
Specifies that, for each column in the derived result table of the *fullselect*, a corresponding column is to be defined for the table. Each defined column adopts the following attributes from its corresponding column of the result table (if applicable to the data type):

- Column name
- Column description
- Data type, length, precision, and scale
- Nullability

The following attributes are not included (although the default value and identity attributes can be included by using the *copy-options*):

- Default value
- Identity attributes
- Hidden attribute
- ROW CHANGE TIMESTAMP
- Any other optional attributes of the tables or views referenced in the fullselect

The following restrictions apply:

- Every select list element must have a unique name (SQLSTATE 42711). The AS clause can be used in the select clause to provide unique names.
- The fullselect cannot refer to host variables or include parameter markers.
- The data types of the result columns of the fullselect must be data types that are valid for columns of a table.

- If row or column level access control (RCAC) is activated for any table that is specified in the fullselect, RCAC is not cascaded to the new table.

WITH NO DATA | WITH DATA
Determines whether to fill the columns of the table with data:

WITH NO DATA
Do not execute the fullselect. It is used only to define the table, which is not populated with the results of the query.

WITH DATA
Execute the fullselect and populate the table with the results of the query.

copy-options
These options specify whether to copy additional attributes of the source result table definition (table, view, or fullselect).

INCLUDING COLUMN DEFAULTS
Column defaults for each updatable column of the source result table definition are copied. Columns that are not updatable will not have a default defined in the corresponding column of the created table.

If LIKE *table-name1* is specified, and *table-name1* identifies a base table, created temporary table, or declared temporary table, then INCLUDING COLUMN DEFAULTS is the default.

EXCLUDING COLUMN DEFAULTS
Column defaults are not copied from the source result table definition.

This clause is the default, except when LIKE *table-name* is specified and *table-name* identifies a base table, created temporary table, or declared temporary table.

INCLUDING IDENTITY COLUMN ATTRIBUTES
If available, identity column attributes (START WITH, INCREMENT BY, and CACHE values) are copied from the source's result table definition. It is possible to copy these attributes if the element of the corresponding column in the table, view, or fullselect is the name of a column of a table, or the name of a column of a view which directly or indirectly maps to the column name of a base table or created temporary table with the identity property. In all other cases, the columns of the new temporary table will not get the identity property. For example:

- The select list of the fullselect includes multiple instances of the name of an identity column (that is, selecting the same column more than once)
- The select list of the fullselect includes multiple identity columns (that is, it involves a join)
- The identity column is included in an expression in the select list
- The fullselect includes a set operation (union, except, or intersect).

EXCLUDING IDENTITY COLUMN ATTRIBUTES
Identity column attributes are not copied from the source result table definition.

ORGANIZE BY
Specifies how the data is organized in the data pages of the table:

ROW
The data is stored by row in the data pages of the table. Each data page stores the data for one or more rows of the table.

COLUMN
The data is stored by column in the data pages of the table. Each data page stores data for one column of the table.

The default is determined by the value of the dft_table_org database configuration parameter.

ON COMMIT
Specifies the action taken on the global temporary table when a COMMIT operation is performed. The default is DELETE ROWS.

DELETE ROWS
All rows of the table will be deleted if no WITH HOLD cursor is open on the table.

PRESERVE ROWS
Rows of the table will be preserved.

LOGGED or NOT LOGGED
Specifies whether operations for the table are logged. The default is LOGGED.

LOGGED
Specifies that insert, update, or delete operations against the table as well as the creation or dropping of the table are to be logged.

NOT LOGGED
Specifies that insert, update, or delete operations against the table are not to be logged, but that the creation or dropping of the table is to be logged. During a ROLLBACK (or ROLLBACK TO SAVEPOINT) operation:

- If the table had been created within a unit of work (or savepoint), the table is dropped
- If the table had been dropped within a unit of work (or savepoint), the table is recreated, but without any data

ON ROLLBACK
Specifies the action that is to be taken on the not logged global temporary table when a ROLLBACK (or ROLLBACK TO SAVEPOINT) operation is performed. The default is DELETE ROWS.

DELETE ROWS
If the table data has been changed, all the rows will be deleted.

PRESERVE ROWS
Rows of the table will be preserved.

Note: Declared global temporary tables using NOT LOGGED ON ROLLBACK PRESERVE ROWS cannot be column-organized.

WITH REPLACE
Indicates that, in the case that a declared temporary table already exists with the specified name, the existing table is replaced with the temporary table defined by this statement (and all rows of the existing table are deleted).

When WITH REPLACE is not specified, then the name specified must not identify a declared temporary table that already exists in the current session (SQLSTATE 42710).

IN *tablespace-name*
Identifies the table space in which the declared temporary table will be instantiated. The table space must exist and be a USER TEMPORARY table space (SQLSTATE 42838), over which the authorization ID of the statement has USE privilege (SQLSTATE 42501). If this clause is not specified, a table space for the table is determined by choosing the USER TEMPORARY table space with the smallest sufficient page size over which the authorization ID of the statement has USE privilege. When more than one table space qualifies, preference is given according to who was granted the USE privilege:

1. The authorization ID
2. A group to which the authorization ID belongs
3. PUBLIC

If more than one table space still qualifies, the final choice is made by the database manager. When no USER TEMPORARY table space qualifies, an error is raised (SQLSTATE 42727).

Determination of the table space can change when:

- Table spaces are dropped or created
- USE privileges are granted or revoked

The sufficient page size of a table is determined by either the byte count of the row or the number of columns. For more details, see "Row Size" in “CREATE TABLE ” on page 1351.

distribution-clause

Specifies the database partitioning or the way the data is distributed across multiple database partitions.

DISTRIBUTE BY HASH (*column-name*, ...)

Specifies the use of the default hashing function on the specified columns, called a *distribution key*, as the distribution method across database partitions. The *column-name* must be an unqualified name that identifies a column of the table (SQLSTATE 42703). The same column must not be identified more than once (SQLSTATE 42709). No column whose data type is BLOB, CLOB, DBCLOB, XML, distinct type based on any of these types, or structured type can be used as part of a distribution key (SQLSTATE 42962).

If this clause is not specified, and the table resides in a multiple partition database partition group with multiple database partitions, a default distribution key is automatically defined.

If none of the columns satisfies the requirements for a default distribution key, the table is created without one. Such tables are allowed only in table spaces that are defined on single-partition database partition groups.

For tables in table spaces that are defined on single-partition database partition groups, any collection of columns with data types that are valid for a distribution key can be used to define the distribution key. If this clause is not specified, no distribution key is created.

DISTRIBUTE BY RANDOM

Specifies that the database manager will select a distribution key to spread data evenly across all database partitions of the database partitioning group. Data distribution is accomplished by using a *random by generation* method. In this method, the database manager will include a column in the table to generate and store a generated value to use in the hashing function. The column will be created with the **IMPLICITLY HIDDEN** clause so that it does not appear in queries unless explicitly included. The value of the column will be automatically generated as new rows are added to the table. By default, the column name is **RANDOM_DISTRIBUTION_KEY**. If it collides with the existing column, a non-conflicting name will be generated by the database manager.

Notes

- A user temporary table space must exist before a declared temporary table can be declared (SQLSTATE 42727).
- ***Referencing a declared temporary table:*** The description of a declared temporary table does not appear in the database catalog (SYSCAT.TABLES); therefore, it is not persistent and is not shareable across database connections. This means that each session that defines a declared temporary table called *table-name* has its own possibly unique description of that declared global temporary table.

  In order to reference the declared temporary table in an SQL statement (other than the DECLARE GLOBAL TEMPORARY TABLE statement), the table must be explicitly or implicitly qualified by the schema name SESSION. If *table-name* is not qualified by SESSION, declared temporary tables are not considered when resolving the reference.

  A reference to SESSION.*table-name* in a connection that has not declared a declared temporary table by that name will attempt to resolve from persistent objects in the catalog. If no such object exists, an error occurs (SQLSTATE 42704).
- When binding a package that has static SQL statements that refer to tables implicitly or explicitly qualified by SESSION, those statements will not be bound statically. When these statements are invoked, they will be incrementally bound, regardless of the VALIDATE option chosen while binding the package. At runtime, each table reference will be resolved to a declared temporary table, if it exists, or a created temporary table, or permanent table. If none exist, an error will be raised (SQLSTATE 42704).
- ***Privileges:*** When a declared temporary table is defined, the definer of the table is granted all table privileges on the table, including the ability to drop the table. Additionally, these privileges are granted to PUBLIC. (None of the privileges are granted with the GRANT option, and none of the privileges appear in the catalog table.) This enables any SQL statement in the session to reference a declared temporary table that has already been defined in that session.

- ***Instantiation and termination:*** For the following explanations, P denotes a session and T is a declared temporary table in the session P:
  - An empty instance of T is created as a result of the DECLARE GLOBAL TEMPORARY TABLE statement that is executed in P.
  - Any SQL statement in P can make reference to T and any reference to T in P is a reference to that same instance of T.
  - If a DECLARE GLOBAL TEMPORARY TABLE statement is specified within the SQL procedure compound statement (defined by BEGIN and END), the scope of the declared temporary table is the connection, not just the compound statement, and the table is known outside of the compound statement. The table is not implicitly dropped at the END of the compound statement. A declared temporary table cannot be defined multiple times by the same name in other compound statements in that session, unless the table has been explicitly dropped.
  - Assuming that the ON COMMIT DELETE ROWS clause was specified implicitly or explicitly, then when a commit operation terminates a unit of work in P, and there is no open WITH HOLD cursor in P that is dependent on T, the commit includes the operation DELETE FROM SESSION.T.
  - When a rollback operation terminates a unit of work or a savepoint in P, and that unit of work or savepoint includes a modification to SESSION.T:
    - If NOT LOGGED was specified, all rows from SESSION.T are deleted unless ON ROLLBACK PRESERVE ROWS was also specified
    - If NOT LOGGED was not specified, the changes to T are undone
  - If NOT LOGGED was specified and an INSERT, UPDATE or DELETE statement fails during execution (as opposed to a compilation error), all rows from SESSION.T are deleted.
  - When a rollback operation terminates a unit of work or a savepoint in P, and that unit of work or savepoint includes the declaration of SESSION.T, then the rollback includes the operation DROP SESSION.T.
  - If a rollback operation terminates a unit of work or a savepoint in P, and that unit of work or savepoint includes the drop of a declared temporary table SESSION.T, then the rollback will undo the drop of the table. If NOT LOGGED was specified, then the table will also have been emptied.
  - When the application process that declared T terminates or disconnects from the database, T is dropped and its instantiated rows are destroyed.
  - When the connection to the server at which T was declared terminates, T is dropped and its instantiated rows are destroyed.
- ***Restrictions on the use of declared temporary tables:*** Declared temporary tables cannot:
  - Be specified in an ALTER, COMMENT, GRANT, LOCK, RENAME or REVOKE statement (SQLSTATE 42995).
  - Be referenced in an AUDIT, CREATE ALIAS, or CREATE VIEW statement (SQLSTATE 42995).
  - Be specified in referential constraints (SQLSTATE 42995).
- Data row compression is enabled for a declared temporary table. When the database manager determines that there is a performance gain, table row data including XML documents stored inline in the base table object will be compressed. However, data compression of the XML storage object of a declared temporary table is not supported.
- Index compression is enabled for indexes that are created on declared temporary tables.
- Index compression is enabled by default for indexes that are created on declared temporary tables. Compression will be shown as on, but indexes will not be compressed if the correct license (IBM Db2 Storage Optimization Feature) is not applied.
- ***Syntax alternatives***: The following alternatives are non-standard. They are supported for compatibility with earlier product versions or with other database products.
  - DEFINITION ONLY can be specified in place of WITH NO DATA.
  - The PARTITIONING KEY clause or DISTRIBUTE ON clause can be specified in place of the DISTRIBUTE BY clause.

– When specifying the value of the datetime special register, NOW() can be specified in place of CURRENT_TIMESTAMP.
– In a CHAR or VARCHAR column definition, you do not need to specify the CCSID explicitly; the correct CCSID will be used automatically. However, if you do specify the CCSID explicitly, it must correspond to the type of database being used:
  - `CCSID ASCII` for a non-unicode database
  - `CCSID UNICODE` for a unicode database

Examples

- *Example 1:* Define a declared temporary table with column definitions for an employee number, salary, bonus, and commission.

```
DECLARE GLOBAL TEMPORARY TABLE SESSION.TEMP_EMP
   (EMPNO  CHAR(6) NOT NULL,
    SALARY DECIMAL(9, 2),
    BONUS  DECIMAL(9, 2),
    COMM   DECIMAL(9, 2)) ON COMMIT PRESERVE ROWS
```

- *Example 2:* Assume that base table USER1.EMPTAB exists and that it contains three columns, one of which is an identity column. Declare a temporary table that has the same column names and attributes (including identity attributes) as the base table.

```
DECLARE GLOBAL TEMPORARY TABLE TEMPTAB1
   LIKE USER1.EMPTAB
   INCLUDING IDENTITY
   ON COMMIT PRESERVE ROWS
```

  In this example, SESSION is used as the implicit qualifier for TEMPTAB1.

Compatibility for Netezza Performance Server (NPS) temporary tables

If you need to declare a temporary table, it is recommended that you use a DECLARE GLOBAL TEMPORARY TABLE statement. However, to provide compatibility with IBM Netezza SQL, a CREATE TEMPORARY TABLE statement can be used instead.

Unlike a DECLARE GLOBAL TEMPORARY TABLE statement, the table name specified by a CREATE TEMPORARY TABLE statement can be qualified with a schema name other than SESSION. The default schema is the current schema. For a schema other than SESSION:

- In order to be able to reference a declared temporary table within STATIC SQL, the package must be bound with VALIDATE RUN.
- The session's ability to use the package cache might be impeded as a consequence.
- The schema name cannot begin with SYS (SQLSTATE 42939).

The qualified name of the temporary table must not identify a table, view, nickname, or alias that is described in the catalog (SQLSTATE 42710).

A CREATE TEMPORARY TABLE statement can include the following additional syntax elements, which correspond to those of DECLARE GLOBAL TEMPORARY TABLE:

- A list of column definitions.
- A DISTRIBUTE BY HASH clause. For compatibility reasons, DISTRIBUTE ON HASH can be specified as an alternative.
- A DISTRIBUTE BY HASH clause. For compatibility reasons, DISTRIBUTE BY RANDOM can be specified as an alternative.
- An AS *fullselect* clause. Note that the fullselect can be enclosed in parentheses.

The following two statements are equivalent and achieve the same result:

```
CREATE TEMPORARY TABLE table_name (column_definition) AS fullselect
```

```
DECLARE GLOBAL TEMPORARY TABLE table_name (column_definition) AS (fullselect) WITH DATA
  ON COMMIT PRESERVE ROWS LOGGED
```

DELETE

The DELETE statement deletes rows from a table, nickname, or view, or the underlying tables, nicknames, or views of the specified fullselect.

Deleting a row from a nickname deletes the row from the data source object to which the nickname refers. Deleting a row from a view deletes the row from the table on which the view is based if no INSTEAD OF trigger is defined for the delete operation on this view. If such a trigger is defined, the trigger will be executed instead.

There are two forms of this statement:

- The *Searched* DELETE form is used to delete one or more rows (optionally determined by a search condition).
- The *Positioned* DELETE form is used to delete exactly one row (as determined by the current position of a cursor).

Invocation

A DELETE statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

To execute either form of this statement, the privileges held by the authorization ID of the statement must include at least one of the following authorities:

- DELETE privilege on the table, view, or nickname from which rows are to be deleted
- CONTROL privilege on the table, view, or nickname from which rows are to be deleted
- DELETEIN privilege on the schema containing the table, view or nickname from which rows are to be deleted
- Schema DATAACCESS authority on the schema containing the table table, view or nickname from which rows are to be deleted
- DATAACCESS authority

To execute a Searched DELETE statement, the privileges held by the authorization ID of the statement must also include at least one of the following authorities for each table, view, or nickname referenced by a subquery:

- SELECT privilege
- CONTROL privilege
- SELECTIN privilege on the schema containing the table, view, or nickname
- Schema DATAACCESS authority on the schema containing the table, view, or nickname
- DATAACCESS authority

If the package used to process the statement is precompiled with SQL92 rules (option LANGLEVEL with a value of SQL92E or MIA), and the searched form of a DELETE statement includes a reference to a column of the table or view in the *search-condition*, the privileges held by the authorization ID of the statement must also include at least one of the following authorities:

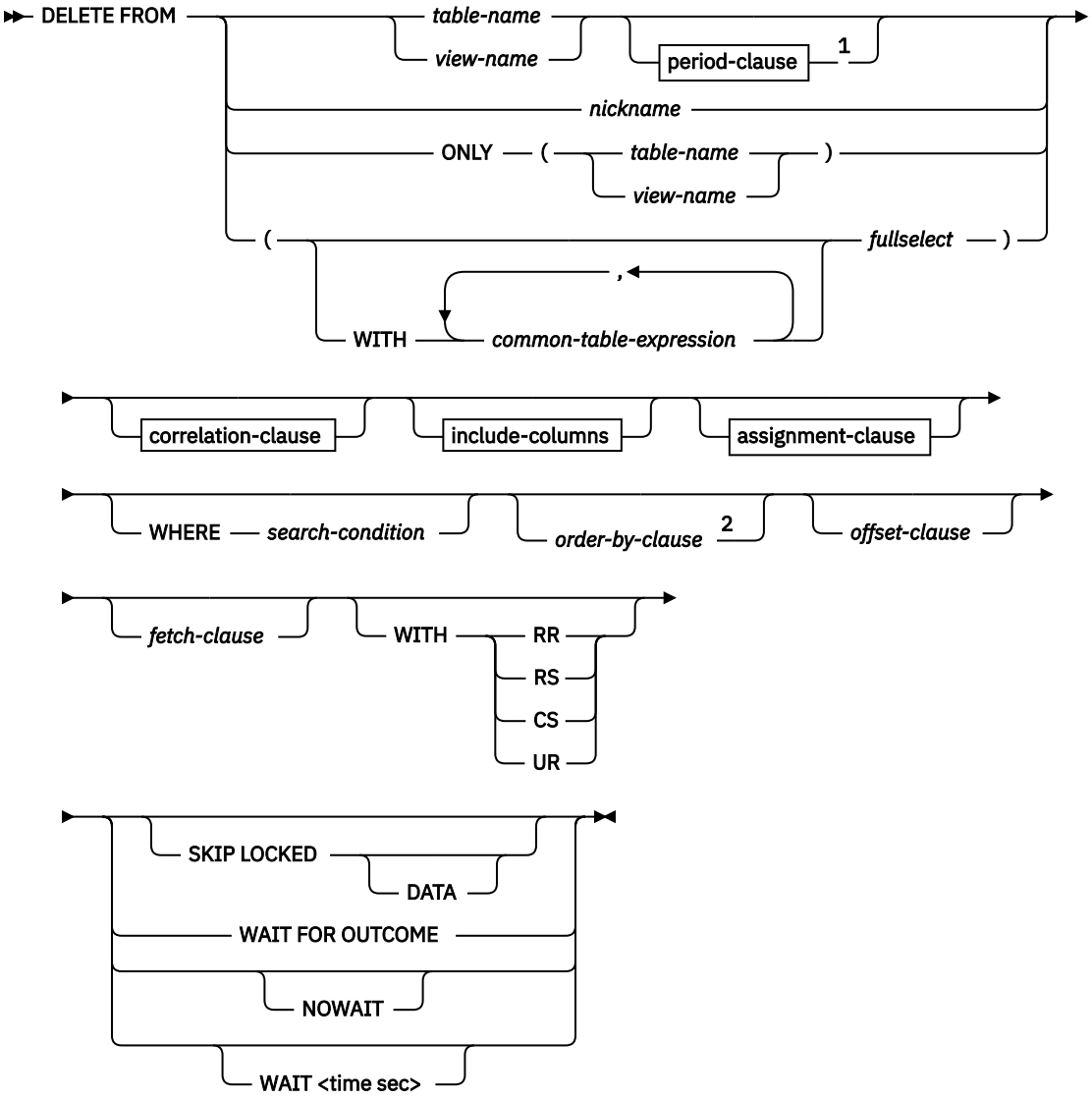
- SELECT privilege
- CONTROL privilege
- SELECTIN privilege on the schema containing the table, view, or nickname
- Schema DATAACCESS authority on the schema containing the table or view
- DATAACCESS authority

If the specified table or view is preceded by the ONLY keyword, the privileges held by the authorization ID of the statement must also include the SELECT privilege for every subtable or subview of the specified table or view.

Group privileges are not checked for static DELETE statements.

If the target of the delete operation is a nickname, the privileges on the object at the data source are not considered until the statement is executed at the data source. At this time, the authorization ID that is used to connect to the data source must have the privileges required for the operation on the object at the data source. The authorization ID of the statement can be mapped to a different authorization ID at the data source.

Syntax (searched-delete)



Notes:

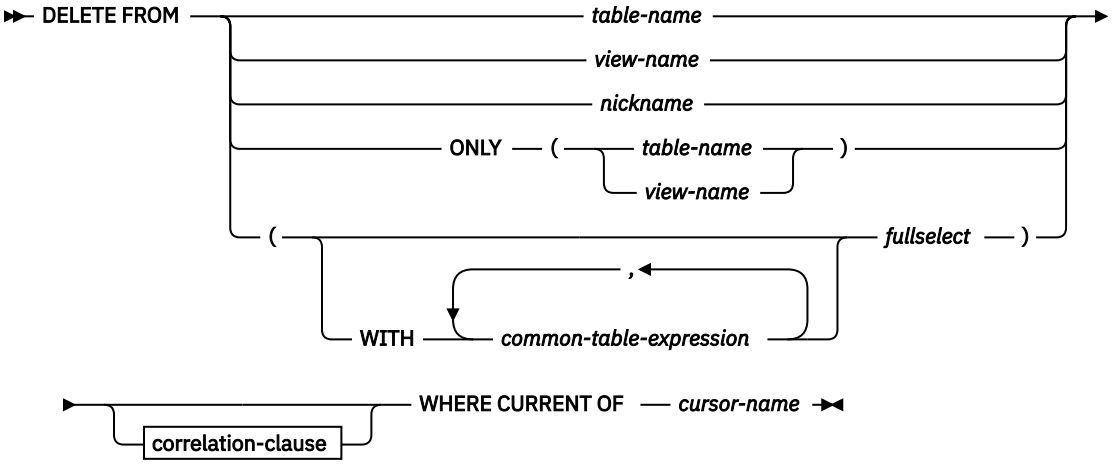
[1] If the *period-clause* is specified, neither the *offset-clause* nor the *fetch-clause* can be specified (SQLSTATE 42601).

[2] If the *order-by-clause* is specified, either the *offset-clause* or *fetch-clause* must also be specified (SQLSTATE 42601).

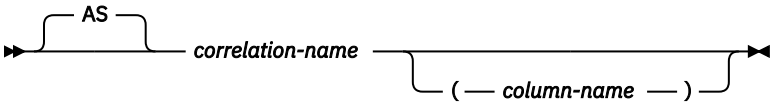
period-clause

include-columns

Syntax (positioned-delete)



correlation-clause



Description

FROM *table-name*, *view-name*, *nickname*, or *(fullselect)*

Identifies the object of the delete operation. The name must identify one of the following objects:

- A table or view that exists in the catalog at the current server
- A table or view at a remote server specified using a remote-object-name

The object must not be a catalog table, a catalog view, a system-maintained materialized query table, or a read-only view.

If *table-name* is a typed table, rows of the table or any of its proper subtables may get deleted by the statement.

If *view-name* is a typed view, rows of the underlying table or underlying tables of the view's proper subviews may get deleted by the statement. If *view-name* is a regular view with an underlying table that is a typed table, rows of the typed table or any of its proper subtables may get deleted by the statement.

If the object of the delete operation is a fullselect, the fullselect must be deletable, as defined in the "Deletable views" Notes item in the description of the CREATE VIEW statement.

For additional restrictions related to temporal tables and use of a view or fullselect as the target of the delete operation, see "Considerations for a system-period temporal table" and "Considerations for an application-period temporal table" in the Notes section.

Only the columns of the specified table can be referenced in the WHERE clause. For a positioned DELETE, the associated cursor must also have specified the table or view in the FROM clause without using ONLY.

FROM ONLY (*table-name*)
Applicable to typed tables, the ONLY keyword specifies that the statement should apply only to data of the specified table and rows of proper subtables cannot be deleted by the statement. For a positioned DELETE, the associated cursor must also have specified the table in the FROM clause using ONLY. If *table-name* is not a typed table, the ONLY keyword has no effect on the statement.

FROM ONLY (*view-name*)
Applicable to typed views, the ONLY keyword specifies that the statement should apply only to data of the specified view and rows of proper subviews cannot be deleted by the statement. For a positioned DELETE, the associated cursor must also have specified the view in the FROM clause using ONLY. If *view-name* is not a typed view, the ONLY keyword has no effect on the statement.

period-clause
Specifies that a period clause applies to the target of the delete operation.

If the target of the delete operation is a view, the following conditions apply to the view:

- The FROM clause of the outer fullselect of the view definition must include a reference, directly or indirectly, to an application-period temporal table (SQLSTATE 42724M).
- An INSTEAD OF DELETE trigger must not be defined for the view (SQLSTATE 428HY).

FOR PORTION OF BUSINESS_TIME
Specifies that the delete only applies to row values for the portion of the period in the row that is specified by the period clause. The BUSINESS_TIME period must exist in the table (SQLSTATE 4274M). FOR PORTION OF BUSINESS_TIME must not be specified if the value of the CURRENT TEMPORAL BUSINESS_TIME special register is not NULL when the BUSTIMESENSITIVE bind option is set to YES (SQLSTATE 428HY).

FROM *value1* TO *value2*
Specifies that the delete applies to rows for the period specified from *value1* up to *value2*. No rows are deleted if *value1* is greater than or equal to *value2*, or if *value1* or *value2* is the null value (SQLSTATE 02000).

For the period specified with FROM *value1* TO *value2*, the BUSINESS_TIME period in a row in the target of the delete is in any of the following states:

- **Overlaps the beginning** of the specified period if the value of the begin column is less than *value1* and the value of the end column is greater than *value1*.
- **Overlaps the end**of the specified period if the value of the end column is greater than or equal to *value2* and the value of the begin column is less than *value2*.
- Is **fully contained** within the specified period if the value for the begin column for BUSINESS_TIME is greater than or equal to *value1* and the value for the corresponding end column is less than or equal to *value2*.
- Is **partially contained** in the specified period if the row overlaps the beginning of the specified period or the end of the specified period, but not both.
- **Fully overlaps** the specified period if the period in the row overlaps the beginning and end of the specified period.
- Is **not contained** in the period if both columns of BUSINESS_TIME are less than or equal to *value1* or greater than or equal to *value2*.

If the BUSINESS_TIME period in a row is not contained in the specified period, the row is not deleted. Otherwise, the delete is applied based on how the values in the columns of the BUSINESS_TIME period overlap the specified period as follows:

- If the BUSINESS_TIME period in a row is fully contained within the specified period, the row is deleted.
- If the BUSINESS_TIME period in a row is partially contained in the specified period and overlaps the beginning of the specified period:
  - The row is deleted.

- A row is inserted using the original values from the row, except that the end column is set to *value1*.

- If the BUSINESS_TIME period in a row is partially contained in the specified period and overlaps the end of the specified period:
  - The row is deleted.
  - A row is inserted using the original values from the row, except that the begin column is set to *value2*.
- If the BUSINESS_TIME period in a row fully overlaps the specified period:
  - The row is deleted.
  - A row is inserted using the original values from the row, except that the end column is set to *value1*.
  - An additional row is inserted using the original values from the row, except that the begin column is set to *value2*.

value1* and *value2
Each expression must return a value that has a date data type, timestamp data type, or a valid data type for a string representation of a date or timestamp (SQLSTATE 428HY). The result of each expression must be comparable to the data type of the columns of the specified period (SQLSTATE 42884). See the comparison rules described in "Assignments and comparisons".

Each expression can contain any of the following supported operands (SQLSTATE 428HY):

- Constant
- Special register
- Variable. For details, refer to "References to variables" in the "Identifiers" topic, in *SQL Reference Volume 1* .
- Scalar function whose arguments are supported operands (though user-defined functions and non-deterministic functions cannot be used)
- CAST specification where the cast operand is a supported operand
- Expression using arithmetic operators and operands

correlation-clause
Can be used within the *search-condition* to designate a table, view, nickname, or fullselect. For a description of *correlation-clause*, see "table-reference" in the description of "Subselect".

include-columns
Specifies a set of columns that are included, along with the columns of *table-name* or *view-name*, in the intermediate result table of the DELETE statement when it is nested in the FROM clause of a fullselect. The *include-columns* are appended at the end of the list of columns that are specified for *table-name* or *view-name*.

INCLUDE
Specifies a list of columns to be included in the intermediate result table of the DELETE statement.

column-name
Specifies a column of the intermediate result table of the DELETE statement. The name cannot be the same as the name of another include column or a column in *table-name* or *view-name* (SQLSTATE 42711).

data-type
Specifies the data type of the include column. The data type must be one that is supported by the CREATE TABLE statement.

assignment-clause
See the description of *assignment-clause* under the UPDATE statement. The same rules apply. The *include-columns* are the only columns that can be set using the *assignment-clause* (SQLSTATE 42703).

WHERE
Specifies a condition that selects the rows to be deleted. The clause can be omitted, a search condition specified, or a cursor named. If the clause is omitted, all rows of the table or view are deleted.

search-condition
Each *column-name* in the search condition, other than in a subquery must identify a column of the table or view.

The *search-condition* is applied to each row of the table, view, or nickname, and the deleted rows are those for which the result of the *search-condition* is true.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the *search condition* is applied to a row, and the results used in applying the *search condition*. In actuality, a subquery with no correlated references is executed once, whereas a subquery with a correlated reference may have to be executed once for each row. If a subquery refers to the object table of a DELETE statement or a dependent table with a delete rule of CASCADE or SET NULL, the subquery is completely evaluated before any rows are deleted.

CURRENT OF *cursor-name*
Identifies a cursor that is defined in a DECLARE CURSOR statement of the program. The DECLARE CURSOR statement must precede the DELETE statement.

The table, view, or nickname named must also be named in the FROM clause of the SELECT statement of the cursor, and the result table of the cursor must not be read-only. (For an explanation of read-only result tables, see "DECLARE CURSOR".)

When the DELETE statement is executed, the cursor must be positioned on a row: that row is the one deleted. After the deletion, the cursor is positioned before the next row of its result table. If there is no next row, the cursor is positioned after the last row.

order-by-clause
Specifies the order of the rows for application of the *offset-clause* and *fetch-clause*. Specify an *order-by-clause* to ensure a predictable order for determining the set of rows to be deleted based on the *offset-clause* and *fetch-clause*. For details on the *order-by-clause*, see “order-by-clause” on page 703.

offset-clause
Limits the effect of the delete by skipping a subset of the qualifying rows. For details on the *offset-clause*, refer to “offset-clause” on page 706.

fetch-clause
Limits the effect of the delete to a subset of the qualifying rows. For details on the *fetch-clause*, refer to “fetch-clause” on page 705.

WITH
Specifies the isolation level used when locating the rows to be deleted.

RR
Repeatable Read

RS
Read Stability

CS
Cursor Stability

UR
Uncommitted Read

The default isolation level of the statement is the isolation level of the package in which the statement is bound. The WITH clause has no effect on nicknames, which always use the default isolation level of the statement.

SKIP LOCKED DATA
The SKIP LOCKED DATA clause specifies that rows are skipped when incompatible locks that would block the progress of the statement are held on the rows by other transactions. These rows can belong to any accessed table addressed in the statement, including tables accessed in a subquery.

This clause applies when the isolation level is CS or RS and is ignored when an isolation level of UR or RR is in effect. It applies to row and block level locks.

Invocation

SKIP LOCKED DATA is ignored if it is specified when WITH RR or WITH UR. The default isolation level of the statement depends on the isolation of the package or plan with which the statement is bound, and whether the result table is read-only. If the default isolation level of the statement is Repeatable Read or Uncommitted Read, then SKIP LOCKED DATA is ignored.

NOWAIT / WAIT <time sec>

Attention: The following feature is available in Db2 11.5.6 and later versions.

The NOWAIT and WAIT clauses specify the number of seconds to wait for a lock before returning an error indicating that a lock cannot be obtained.

When using the WAIT clause, `<time sec>` is an integer between `-1` and `32767`.

Note: For `NOWAIT` and `WAIT 0`, locks are not waited for. If no lock is available at the time of the request, a `-911` error is returned.

When a WAIT value of `-1` is specified, lock timeout detection is turned off. In this situation a lock is waited for (if one is not available at the time of the request) until either of the following events occur:

- The lock is granted.
- A deadlock occurs.

Use of the NOWAIT and WAIT clauses overwrites the value of the LOCKTIMEOUT database configuration variable and the value of the CURRENT LOCK TIMEOUT special register for this delete statement. This means that adding the NOWAIT/WAIT clause with a wait time value of **t** has the same effect as executing the delete statement with a LOCKTIMEOUT value or CURRENT LOCK TIMEOUT value of **t**.

While the NOWAIT and WAIT clauses are not allowed for positioned updates and deletes, you can use them in the declaration of the cursor. When used in the cursor declaration, the specified wait time value is inherited by the statements that use this cursor.

Rules

- ***Triggers:*** DELETE statements may cause triggers to be executed. A trigger may cause other statements to be executed, or may raise error conditions based on the deleted rows. If a DELETE statement on a view causes an INSTEAD OF trigger to fire, referential integrity will be checked against the updates performed in the trigger, and not against the underlying tables of the view that caused the trigger to fire.
- ***Referential integrity:*** If the identified table or the base table of the identified view is a parent, the rows selected for delete must not have any dependents in a relationship with a delete rule of RESTRICT, and the DELETE must not cascade to descendent rows that have dependents in a relationship with a delete rule of RESTRICT.

  If the delete operation is not prevented by a RESTRICT delete rule, the selected rows are deleted. Any rows that are dependents of the selected rows are also affected:

  – The nullable columns of the foreign keys of any rows that are their dependents in a relationship with a delete rule of SET NULL are set to the null value.

  – Any rows that are their dependents in a relationship with a delete rule of CASCADE are also deleted, and the preceding rules apply, in turn, to those rows.

  The delete rule of NO ACTION is checked to enforce that any non-null foreign key refers to an existing parent row after the other referential constraints have been enforced.
- ***Security policy:*** If the identified table or the base table of the identified view is protected with a security policy, the session authorization ID must have the label-based access control (LBAC) credentials that allow:

- Write access to all protected columns (SQLSTATE 42512)
- Read and write access to all of the rows that are selected for deletion (SQLSTATE 42519)

Notes

- If an error occurs during the execution of a multiple row DELETE, no changes are made to the database.
- Unless appropriate locks already exist, one or more exclusive locks are acquired during the execution of a successful DELETE statement. Issuing a COMMIT or ROLLBACK statement will release the locks. Until the locks are released by a commit or rollback operation, the effect of the delete operation can only be perceived by:
  - The application process that performed the deletion
  - Another application process using isolation level UR.

  The locks can prevent other application processes from performing operations on the table.
- If an application process deletes a row on which any of its cursors are positioned, those cursors are positioned before the next row of their result table. Let C be a cursor that is positioned before row R (as a result of an OPEN, a DELETE through C, a DELETE through some other cursor, or a searched DELETE). In the presence of INSERT, UPDATE, and DELETE operations that affect the base table from which R is derived, the next FETCH operation referencing C does not necessarily position C on R. For example, the operation can position C on R', where R' is a new row that is now the next row of the result table.
- SQLERRD(3) in the SQLCA shows the number of rows that qualified for the delete operation. In the context of an SQL procedure statement, the value can be retrieved using the ROW_COUNT variable of the GET DIAGNOSTICS statement. SQLERRD(5) in the SQLCA shows the number of rows affected by referential constraints and by triggered statements. It includes rows that were deleted as a result of a CASCADE delete rule and rows in which foreign keys were set to the null value as the result of a SET NULL delete rule. With regards to triggered statements, it includes the number of rows that were inserted, updated, or deleted.
- If an error occurs that prevents deleting all rows matching the search condition and all operations required by existing referential constraints, no changes are made to the table and the error is returned.
- For nicknames, the external server option `iud_app_svpt_enforce` poses an additional limitation. Refer to the Federated documentation for more information.
- For some data sources, the SQLCODE -20190 may be returned on a delete against a nickname because of potential data inconsistency. Refer to the Federated documentation for more information.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - The FROM keyword can be omitted.
- **Considerations for a system-period temporal table:** The target of the DELETE statement must not be a fullselect that references a view in the FROM clause followed by a period specification for SYSTEM_TIME if the view is defined with the WITH CHECK OPTION and the view definition includes a WHERE clause containing one of the following syntax elements (SQLSTATE 51046):
  - A subquery that references a system-period temporal table (directly or indirectly)
  - An invocation of an SQL routine that has a package associated with it
  - An invocation of an external routine with a data access indication other than NO SQL

  If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a non-null value, an underlying target of the UPDATE statement must not be a system-period temporal table (SQLSTATE 51046), and the target of the DELETE statement must not be a view defined with the WITH CHECK OPTION if the view definition includes a WHERE clause containing one of the following syntax elements (SQLSTATE 51046):
  - A subquery that references a system-period temporal table (directly or indirectly)
  - An invocation of an SQL routine that has a package associated with it

– An invocation of an external routine with a data access indication other than NO SQL

If the DELETE statement has a search condition containing a correlated subquery that references historical rows (explicitly referencing the name of the history table name or implicitly through the use of a period specification in the FROM clause), the deleted rows that are stored as historical rows are potentially visible for delete operations for the rows subsequently processed for the statement.

The mass delete algorithm is not used for a DELETE statement for a table defined as a system-period temporal table that does not contain a search condition.

- **Considerations for a history table:** When a row of a system-period temporal table is deleted, a historical copy of the row is inserted into the corresponding history table and the end timestamp of the historical row is captured in the form of a system determined value that corresponds to the time of the data change operation. The database manager assigns the value that is generated using a reading of the time-of-day clock during execution of the first data change statement in the transaction that requires a value to be assigned to the row begin or transaction start-ID column in a table, or a row in a system-period temporal table is deleted. The database manager ensures uniqueness of the generated values for an end column in a history table across transactions. The timestamp value might be adjusted to ensure that rows inserted into the history table have the end timestamp value greater than the begin timestamp value which can happen when a conflicting transaction is updating the same row in the system-period temporal table (SQLSTATE 01695). The database configuration parameter **systime_period_adj** must be set to Yes for this adjustment in the timestamp value to occur otherwise and error is returned (SQLSTATE 57062).

  For a delete operation, the adjustment only affects the value for the end column in the history table that corresponds to the row-end column in the associated system-period temporal table. Take these adjustments into consideration on subsequent references to the table when there is a search for the transaction start time in the row-begin column and row-end column for the SYSTEM_TIME period of the associated system-period temporal table.
- **Considerations for an application-period temporal table:** The target of the DELETE statement must not be a fullselect that references a view in the FROM clause followed by a period specification for BUSINESS_TIME if the view is defined with the WITH CHECK OPTION and the view definition includes a WHERE clause containing one of the following syntax elements (SQLSTATE 51046):

  – A subquery that references an application-period temporal table (directly or indirectly)

  – An invocation of an SQL routine that has a package associated with it

  – An invocation of an external routine with a data access indication other than NO SQL

  If the CURRENT TEMPORAL BUSINESS_TIME special register is set to a non-null value, the target of the DELETE statement must not be a view defined with the WITH CHECK option if the view definition includes a WHERE clause containing one of the following syntax elements (SQLSTATE 51046):

  – A subquery that references an application-period temporal table (directly or indirectly)

  – An invocation of an SQL routine that has a package associated with it

  – An invocation of an external routine with a data access indication other than NO SQL

  A DELETE statement for an application-period temporal table that contains a FOR PORTION OF BUSINESS_TIME clause indicates between which two points in time that the deletes are effective. When FOR PORTION OF BUSINESS_TIME is specified and the period value for a row, specified by the values of the row-begin column and row-end column, is only partially contained in the period specified from *value1* up to *value2*, the row is deleted and one or two rows are automatically inserted to represent the portion of the row that is not deleted. New values are generated for each generated column in an application-period temporal table for each row that is automatically inserted as a result of a delete operation on the table. If a generated column is defined as part of a unique or primary key, parent key in a referential constraint, or unique index, it is possible that an automatic insert will violate a constraint or index in which case an error is returned.

  When an application-period temporal table is the target of an DELETE statement, the value in effect for the CURRENT TEMPORAL BUSINESS_TIME special register is not the null value, and the BUSTIMESENSITIVE bind option is set to YES, the following additional predicates are implicit:

```
    bt_begin <= CURRENT TEMPORAL BUSINESS_TIME
AND bt_end > CURRENT TEMPORAL BUSINESS_TIME
```

where bt_begin and bt_end are the begin and end columns of the BUSINESS_TIME period of the target table of the DELETE statement.

- **Considerations for application-period temporal tables and triggers:** When a row is deleted and the FOR PORTION OF BUSINESS_TIME clause is specified, additional rows may be implicitly inserted to reflect any portion of the row that was not deleted. Any existing delete triggers are activated for the rows deleted, and any existing insert triggers are activated for rows that are implicitly inserted.

Examples

- *Example 1:* Delete department (DEPTNO) 'D11' from the DEPARTMENT table.

```
DELETE FROM DEPARTMENT
  WHERE DEPTNO = 'D11'
```

- *Example 2:* Delete all the departments from the DEPARTMENT table (that is, empty the table).

```
DELETE FROM DEPARTMENT
```

- *Example 3:* Delete from the EMPLOYEE table any sales rep or field rep who didn't make a sale in 1995.

```
DELETE FROM EMPLOYEE
  WHERE LASTNAME NOT IN
    (SELECT SALES_PERSON
      FROM SALES
      WHERE YEAR(SALES_DATE)=1995)
      AND JOB IN ('SALESREP','FIELDREP')
```

-
- *Example 4:* Delete all the duplicate employee rows from the EMPLOYEE table. An employee row is considered to be a duplicate if the last names match. Keep the employee row with the smallest first name in lexical order.

```
DELETE FROM
  (SELECT ROWNUMBER() OVER (PARTITION BY LASTNAME ORDER BY FIRSTNME)
    FROM EMPLOYEE) AS E(RN)
    WHERE RN > 1
```

DESCRIBE

The DESCRIBE statement obtains information about an object.

There are two types of information that can be obtained with this statement. Each of these is described separately.

- Input parameter markers of a prepared statement. Gets information about the input parameter markers in a prepared statement. This information is put into a descriptor.
- The output of a prepared statement. Gets information about a prepared statement or information about the select list columns in a prepared SELECT statement. This information is put into a descriptor.

DESCRIBE INPUT

The DESCRIBE INPUT statement obtains information about the input parameter markers of a prepared statement.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax

Description

statement-name
Identifies the prepared statement. When the DESCRIBE INPUT statement is executed, the name must identify a statement that has been prepared by the application process at the current server.

For a CALL statement, the information returned describes the input parameters, defined as IN or INOUT, of the procedure. Input parameter markers are always considered nullable, regardless of usage.

INTO *descriptor-name*
Identifies an SQL descriptor area (SQLDA). Before the DESCRIBE INPUT statement is executed, the following variable in the SQLDA must be set:

SQLN
Specifies the number of SQLVAR occurrences provided in the SQLDA. SQLN must be set to a value greater than or equal to zero before the DESCRIBE INPUT statement is executed.

When the DESCRIBE INPUT statement is executed, the database manager assigns values to the variables of the SQLDA as follows:

SQLDAID
The first 6 bytes are set to 'SQLDA ' (that is, 5 letters followed by the space character).

The seventh byte, defined as SQLDOUBLED, is set based on the parameter markers described:

- If the SQLDA contains two SQLVAR entries for every input parameter, the seventh byte is set to '2'. This technique is used to accommodate LOB or structured type input parameters.
- Otherwise, the seventh byte is set to the space character.

The seventh byte is set to the space character if there is not enough room in the SQLDA to contain the description of all input parameter markers.

The eighth byte is set to the space character.

SQLDABC
Length of the SQLDA in bytes.

SQLD
The number of IN and INOUT parameters of the procedure.

SQLVAR
If the value of SQLD is 0, or greater than the value of SQLN, no values are assigned to occurrences of SQLVAR.

If the value of SQLD is *n*, where *n* is greater than 0 but less than or equal to the value of SQLN, values are assigned to the first *n* occurrences of SQLVAR. The values describe parameter markers for the input parameters of the procedure. The first occurrence of SQLVAR describes the first input parameter marker, the second occurrence of SQLVAR describes the second input parameter marker, and so on.

Base SQLVAR

SQLTYPE
A code showing the data type of the parameter and whether or not it can contain null values.

SQLLEN
A length value depending on the data type of the parameter. SQLLEN is 0 for LOB data types.

SQLNAME
The sqlname is derived as follows:

- If the SQLVAR corresponds to a parameter marker that is in the parameter list of a procedure and is not part of an expression, sqlname contains the name of the parameter if one was specified on the CREATE PROCEDURE statement.
- If the SQLVAR corresponds to a named parameter marker, sqlname contains the name of the parameter marker.
- Otherwise, sqlname contains an ASCII numeric literal value that represents the SQLVAR's position within the SQLDA.

Secondary SQLVAR
These variables are only used if the number of SQLVAR entries are doubled to accommodate LOB, distinct type, structured type, or reference type parameters.

SQLLONGLEN
The length attribute of a BLOB, CLOB, or DBCLOB parameter.

SQLDATATYPE_NAME
For any user-defined type (distinct or structured) parameter, the database manager sets this to the fully qualified user-defined type name. For a reference type parameter, the database manager sets this to the fully qualified user-defined type name of the target type of the reference. Otherwise, schema name is SYSIBM and the type name is the name in the TYPENAME column of the SYSCAT.DATATYPES catalog view.

Notes

- **Preparing the SQLDA:** Before the DESCRIBE INPUT statement is executed, the SQLDA must be allocated and the value of SQLN must be set to a value greater than or equal to zero to indicate how many occurrences of SQLVAR are provided in the SQLDA. Enough storage must be allocated to contain SQLN occurrences. To obtain the description of the input parameter markers in the prepared statement, the number of occurrences of SQLVAR must not be less than the number of input parameter markers. Furthermore, if the input parameter markers include LOBs or structured types, the number of occurrences of SQLVAR should be two times the number of input parameter markers.
- Code page conversions between extended UNIX code (EUC) code pages and DBCS code pages, or between Unicode and non-Unicode code pages, can result in expansion or contraction of character lengths.
- If a structured type is being selected, but no FROM SQL transform is defined (either because no TRANSFORM GROUP was specified using the CURRENT DEFAULT TRANSFORM GROUP special register (SQLSTATE 428EM), or because the named group does not have a FROM SQL transform function defined (SQLSTATE 42744), an error is returned.
- **Allocating the SQLDA:** Three of the possible ways to allocate the SQLDA are as follows:

  First Technique: Allocate an SQLDA with enough occurrences of SQLVAR to accommodate any select list that the application will have to process. If the table contains any LOB, distinct type, structured type, or reference type columns, the number of SQLVARs should be double the maximum number of columns; otherwise the number should be the same as the maximum number of columns. Having done the allocation, the application can use this SQLDA repeatedly.

  This technique uses a large amount of storage that is never deallocated, even when most of this storage is not used for a particular select list.

  Second Technique: Repeat the following two steps for every processed select list:

  1. Execute a DESCRIBE INPUT statement with an SQLDA that has no occurrences of SQLVAR; that is, an SQLDA for which SQLN is zero. The value returned for SQLD is the number of columns in the result table. This is either the required number of occurrences of SQLVAR or half the required number. Because there were no SQLVAR entries, a warning with SQLSTATE 01005 will be issued. If the SQLCODE accompanying that warning is equal to one of +237, +238 or +239, the number of SQLVAR entries should be double the value returned in SQLD. (The return of these positive

SQLCODEs assumes that the SQLWARN bind option setting was YES (return positive SQLCODEs). If SQLWARN was set to NO, +238 is still returned to indicate that the number of SQLVAR entries must be double the value returned in SQLD.)

2. Allocate an SQLDA with enough occurrences of SQLVAR. Then execute the DESCRIBE statement again, using this new SQLDA.

This technique allows better storage management than the first technique, but it doubles the number of DESCRIBE INPUT statements.

Third Technique: Allocate an SQLDA that is large enough to handle most, and perhaps all, select lists but is also reasonably small. Execute DESCRIBE INPUT and check the SQLD value. Use the SQLD value for the number of occurrences of SQLVAR to allocate a larger SQLDA, if necessary.

This technique is a compromise between the first two techniques. Its effectiveness depends on a good choice of size for the original SQLDA.

Example

Execute a DESCRIBE INPUT statement with an SQLDA that has enough SQLVAR occurrences to describe any number of input parameters a prepared statement might have. Assume that five parameter markers at most will need to be described and that the input data does not contain LOBs.

```
    /* STMT1_STR contains INSERT statement with VALUES clause */
EXEC SQL PREPARE STMT1_NAME FROM :STMT1_STR;
... /* code to set SQLN to 5 and to allocate the SQLDA        */
EXEC SQL DESCRIBE INPUT STMT1_NAME INTO :SQLDA;
.
.
.
```

This example uses the first technique described under "Allocating the SQLDA" in "DESCRIBE OUTPUT".

DESCRIBE OUTPUT

The DESCRIBE OUTPUT statement obtains information about a prepared statement.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax

Description

statement-name

Identifies the prepared statement. When the DESCRIBE OUTPUT statement is executed, the name must identify a statement that has been prepared by the application process at the current server.

If the prepared statement is a SELECT or VALUES INTO statement, the information returned describes the columns in its result table. If the prepared statement is a CALL statement, the information returned describes the output parameters, defined as OUT or INOUT, of the procedure.

INTO *descriptor-name*

Identifies an SQL descriptor area (SQLDA). Before the DESCRIBE OUTPUT statement is executed, the following variable in the SQLDA must be set:

SQLN

Specifies the number of SQLVAR occurrences provided in the SQLDA. SQLN must be set to a value greater than or equal to zero before the DESCRIBE OUTPUT statement is executed.

When the DESCRIBE OUTPUT statement is executed, the database manager assigns values to the variables of the SQLDA as follows:

SQLDAID

The first 6 bytes are set to 'SQLDA ' (that is, 5 letters followed by the space character).

The seventh byte, defined as SQLDOUBLED, is set based on the results columns or parameter markers described:

- If the SQLDA contains two SQLVAR entries for every column or output parameter, the seventh byte is set to '2'. This technique is used to accommodate LOB, distinct type, structured type, or reference type columns, or output parameters.
- Otherwise, the seventh byte is set to the space character.

The seventh byte is set to the space character if there is not enough room in the SQLDA to contain the description of all result columns or output parameter markers.

The eighth byte is set to the space character.

SQLDABC

Length of the SQLDA in bytes.

SQLD

If the prepared statement is a SELECT, SQLD is set to the number of columns in its result table. If the prepared statement is a CALL statement, SQLD is set to the number of OUT and INOUT parameters of the procedure. Otherwise, SQLD is set to 0.

SQLVAR

If the value of SQLD is 0, or greater than the value of SQLN, no values are assigned to occurrences of SQLVAR.

If the value of SQLD is *n*, where *n* is greater than 0 but less than or equal to the value of SQLN, values are assigned to SQLTYPE, SQLLEN, SQLNAME, SQLLONGLEN, and SQLDATATYPE_NAME for the first *n* occurrences of SQLVAR. These values describe either columns of the result table or parameter markers for the output parameters of the procedure. The first occurrence of SQLVAR describes the first column or output parameter marker, the second occurrence of SQLVAR describes the second column or output parameter marker, and so on.

Base SQLVAR

SQLTYPE

A code showing the data type of the column or parameter and whether or not it can contain null values.

SQLLEN

A length value depending on the data type of the column or parameter. SQLLEN is 0 for LOB data types.

SQLNAME

The sqlname is derived as follows:

- If the SQLVAR corresponds to a derived column for a simple column reference in the select list of a select-statement, sqlname is the name of the column.
- If the SQLVAR corresponds to a parameter marker that is in the parameter list of a procedure and is not part of an expression, sqlname contains the name of the parameter if one was specified on CREATE PROCEDURE.

- Otherwise sqlname contains an ASCII numeric literal value that represents the SQLVAR's position within the SQLDA.

Secondary SQLVAR
These variables are only used if the number of SQLVAR entries is doubled to accommodate LOB, distinct type, structured type, or reference type columns or parameters.

SQLLONGLEN
The length attribute of a BLOB, CLOB, or DBCLOB column or parameter.

SQLDATATYPE_NAME
For any user-defined type (distinct or structured) column or parameter, the database manager sets this to the fully qualified user-defined type name. For a reference type column or parameter, the database manager sets this to the fully qualified user-defined type name of the target type of the reference. Otherwise, schema name is SYSIBM and the type name is the name in the TYPENAME column of the SYSCAT.DATATYPES catalog view.

Notes

- Before the DESCRIBE OUTPUT statement is executed, the value of SQLN must be set to indicate how many occurrences of SQLVAR are provided in the SQLDA and enough storage must be allocated to contain SQLN occurrences. For example, to obtain the description of the columns of the result table of a prepared SELECT statement, the number of occurrences of SQLVAR must not be less than the number of columns.
- If a LOB of a large size is expected, then remember that manipulating this large object will affect application memory. Given this condition, consider using locators or file reference variables. Modify the SQLDA after the DESCRIBE OUTPUT statement is executed but before allocating storage so that an SQLTYPE of SQL_TYP_xLOB is changed to SQL_TYP_xLOB_LOCATOR or SQL_TYP_xLOB_FILE with corresponding changes to other fields such as SQLLEN. Then allocate storage based on SQLTYPE and continue.
- Code page conversions between extended UNIX code (EUC) code pages and DBCS code pages, or between Unicode and non-Unicode code pages, can result in the expansion and contraction of character lengths.
- If a structured type is being selected, but no FROM SQL transform is defined (either because no TRANSFORM GROUP was specified using the CURRENT DEFAULT TRANSFORM GROUP special register (SQLSTATE 428EM), or because the named group does not have a FROM SQL transform function defined (SQLSTATE 42744), an error is returned.
- **Allocating the SQLDA:** Three of the possible ways to allocate the SQLDA are as follows:

  First Technique: Allocate an SQLDA with enough occurrences of SQLVAR to accommodate any select list that the application will have to process. If the table contains any LOB, distinct type, structured type, or reference type columns, the number of SQLVARs should be double the maximum number of columns; otherwise the number should be the same as the maximum number of columns. Having done the allocation, the application can use this SQLDA repeatedly.

  This technique uses a large amount of storage that is never deallocated, even when most of this storage is not used for a particular select list.

  Second Technique: Repeat the following two steps for every processed select list:

  1. Execute a DESCRIBE OUTPUT statement with an SQLDA that has no occurrences of SQLVAR; that is, an SQLDA for which SQLN is zero. The value returned for SQLD is the number of columns in the result table. This is either the required number of occurrences of SQLVAR or half the required number. Because there were no SQLVAR entries, a warning with SQLSTATE 01005 will be issued. If the SQLCODE accompanying that warning is equal to one of +237, +238 or +239, the number of SQLVAR entries should be double the value returned in SQLD. (The return of these positive SQLCODEs assumes that the SQLWARN bind option setting was YES (return positive SQLCODEs). If SQLWARN was set to NO, +238 is still returned to indicate that the number of SQLVAR entries must be double the value returned in SQLD.)

2. Allocate an SQLDA with enough occurrences of SQLVAR. Then execute the DESCRIBE OUTPUT statement again, using this new SQLDA.

This technique allows better storage management than the first technique, but it doubles the number of DESCRIBE OUTPUT statements.

Third Technique: Allocate an SQLDA that is large enough to handle most, and perhaps all, select lists but is also reasonably small. Execute DESCRIBE and check the SQLD value. Use the SQLD value for the number of occurrences of SQLVAR to allocate a larger SQLDA, if necessary.

This technique is a compromise between the first two techniques. Its effectiveness depends on a good choice of size for the original SQLDA.

- ***Considerations for implicitly hidden columns:*** A DESCRIBE OUTPUT statement returns only information about an implicitly hidden column if the column is explicitly specified as part of the SELECT list of the final result table of the query being described. If implicitly hidden columns are not part of the result table of a query, a DESCRIBE OUTPUT statement that returns information about that query will not contain information about any implicitly hidden columns.

Example

In a C program, execute a DESCRIBE OUTPUT statement with an SQLDA that has no occurrences of SQLVAR. If SQLD is greater than zero, use the value to allocate an SQLDA with the necessary number of occurrences of SQLVAR and then execute a DESCRIBE statement using that SQLDA.

```
EXEC SQL  BEGIN DECLARE SECTION;
  char  stmt1_str[200];
EXEC SQL  END DECLARE SECTION;
EXEC SQL  INCLUDE SQLDA;
EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;

... /* code to prompt user for a query, then to generate */
    /* a select-statement in the stmt1_str            */
EXEC SQL  PREPARE STMT1_NAME FROM :stmt1_str;

... /* code to set SQLN to zero and to allocate the SQLDA */
EXEC SQL  DESCRIBE STMT1_NAME INTO :sqlda;

... /* code to check that SQLD is greater than zero, to set */
    /* SQLN to SQLD, then to re-allocate the SQLDA          */
EXEC SQL  DESCRIBE STMT1_NAME INTO :sqlda;

... /* code to prepare for the use of the SQLDA             */
    /* and allocate buffers to receive the data             */
EXEC SQL  OPEN DYN_CURSOR;

... /* loop to fetch rows from result table                 */
EXEC SQL  FETCH DYN_CURSOR USING DESCRIPTOR :sqlda;
.
.
.
```

DISCONNECT

The DISCONNECT statement destroys one or more connections when there is no active unit of work (that is, after a commit or rollback operation).

If a single connection is the target of the DISCONNECT statement, the connection is destroyed only if the database has participated in an existing unit of work, regardless of whether there is an active unit of work. For example, if several other databases have done work, but the target in question has not, it can still be disconnected without destroying the connection.

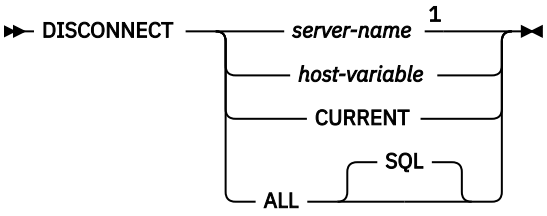
Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax



Notes:

[1] Note that an application server named CURRENT or ALL can only be identified by a host variable.

Description

server-name* or *host-variable

Identifies the application server by the specified *server-name* or a *host-variable* which contains the *server-name*.

If a *host-variable* is specified, it must be a character string variable with a length attribute that is not greater than 8, and it must not include an indicator variable. The *server-name* that is contained within the *host-variable* must be left-aligned and must not be delimited by quotation marks.

Note that the *server-name* is a database alias identifying the application server. It must be listed in the application requester's local directory.

The specified database-alias or the database-alias contained in the host variable must identify an existing connection of the application process. If the database-alias does not identify an existing connection, an error (SQLSTATE 08003) is raised.

CURRENT

Identifies the current connection of the application process. The application process must be in the connected state. If not, an error (SQLSTATE 08003) is raised.

ALL

Indicates that all existing connections of the application process are to be destroyed. An error or warning does not occur if no connections exist when the statement is executed. The optional keyword SQL is included to be consistent with the syntax of the RELEASE statement.

Rules

- Generally, the DISCONNECT statement cannot be executed while within a unit of work. If attempted, an error (SQLSTATE 25000) is raised. The exception to this rule is if a single connection is specified to be disconnected and the database has not participated in an existing unit of work. In this case, it does not matter if there is an active unit of work when the DISCONNECT statement is issued.
- The DISCONNECT statement cannot be executed at all in the Transaction Processing (TP) Monitor environment (SQLSTATE 25000). It is used when the SYNCPOINT precompiler option is set to TWOPHASE.

Notes

- If the DISCONNECT statement is successful, each identified connection is destroyed.

  If the DISCONNECT statement is unsuccessful, the connection state of the application process and the states of its connections are unchanged.
- If DISCONNECT is used to destroy the current connection, the next executed SQL statement should be CONNECT or SET CONNECTION.

- Type 1 CONNECT semantics do not preclude the use of DISCONNECT. However, though DISCONNECT CURRENT and DISCONNECT ALL can be used, they will not result in a commit operation like a CONNECT RESET statement would do.

  If *server-name* or *host-variable* is specified in the DISCONNECT statement, it must identify the current connection because Type 1 CONNECT only supports one connection at a time. Generally, DISCONNECT will fail if within a unit of work with the exception noted in "Rules".
- Resources are required to create and maintain remote connections. Thus, a remote connection that is not going to be reused should be destroyed as soon as possible.
- Connections can also be destroyed during a commit operation because the connection option is in effect. The connection option could be AUTOMATIC, CONDITIONAL, or EXPLICIT, which can be set as a precompiler option or through the SET CLIENT API at run time. For information about the specification of the DISCONNECT option, see "Distributed relational databases".

Examples

- *Example 1:* The SQL connection to IBMSTHDB is no longer needed by the application. The following statement should be executed after a commit or rollback operation to destroy the connection.

```
EXEC SQL DISCONNECT IBMSTHDB;
```

- *Example 2:* The current connection is no longer needed by the application. The following statement should be executed after a commit or rollback operation to destroy the connection.

```
EXEC SQL DISCONNECT CURRENT;
```

- *Example 3:* The existing connections are no longer needed by the application. The following statement should be executed after a commit or rollback operation to destroy all the connections.

```
EXEC SQL DISCONNECT ALL;
```

DROP

The DROP statement deletes an object. Any objects that are directly or indirectly dependent on that object are either deleted or made inoperative. Whenever an object is deleted, its description is deleted from the catalog, and any packages that reference the object are invalidated.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

When dropping objects that allow two-part names, the privileges held by the authorization ID of the statement must include at least one of the following authorities:

- DROPIN privilege on the schema for the object
- Owner of the object, as recorded in the OWNER column of the catalog view for the object
- CONTROL privilege on the object (applicable only to indexes, index specifications, nicknames, packages, tables, and views)
- Owner of the user-defined type, as recorded in the OWNER column of the SYSCAT.DATATYPES catalog view (applicable only when dropping a method that is associated with a user-defined type)
- SCHEMAADM authority on the schema for the object
- DBADM authority

When dropping a table or view hierarchy, the privileges held by the authorization ID of the statement must include one of the previously mentioned privileges for each of the tables or views in the hierarchy.

When dropping an audit policy, the privileges held by the authorization ID of the statement must include SECADM authority.

When dropping a buffer pool, database partition group, storage group, or table space, the privileges held by the authorization ID of the statement must include SYSADM or SYSCTRL authority.

When dropping a data type mapping, function mapping, server definition, or wrapper, the privileges held by the authorization ID of the statement must include DBADM authority.

When dropping an event monitor the privilege held by the authorization ID of the statement must include SQLADM or DBADM authority.

When dropping a role, the privileges held by the authorization ID of the statement must include SECADM authority.

When dropping a row permission or a column mask, the privileges held by the authorization ID of the statement must include SECADM authority.

When dropping a schema, the privileges held by the authorization ID of the statement must include DBADM authority, or be the schema owner, as recorded in the OWNER column of the SYSCAT.SCHEMATA catalog view.

When dropping a security label, a security label component, or a security policy, the privileges held by the authorization ID of the statement must include SECADM authority.

When dropping a service class, work action set, work class set, workload, threshold, or histogram template, the privileges held by the authorization ID of the statement must include WLMADM or DBADM authority.

When dropping a system-period temporal table, the privileges held by the authorization ID of the statement must also include at least one of the following authorities:

- Privileges to drop the associated history table
- Administrative authority

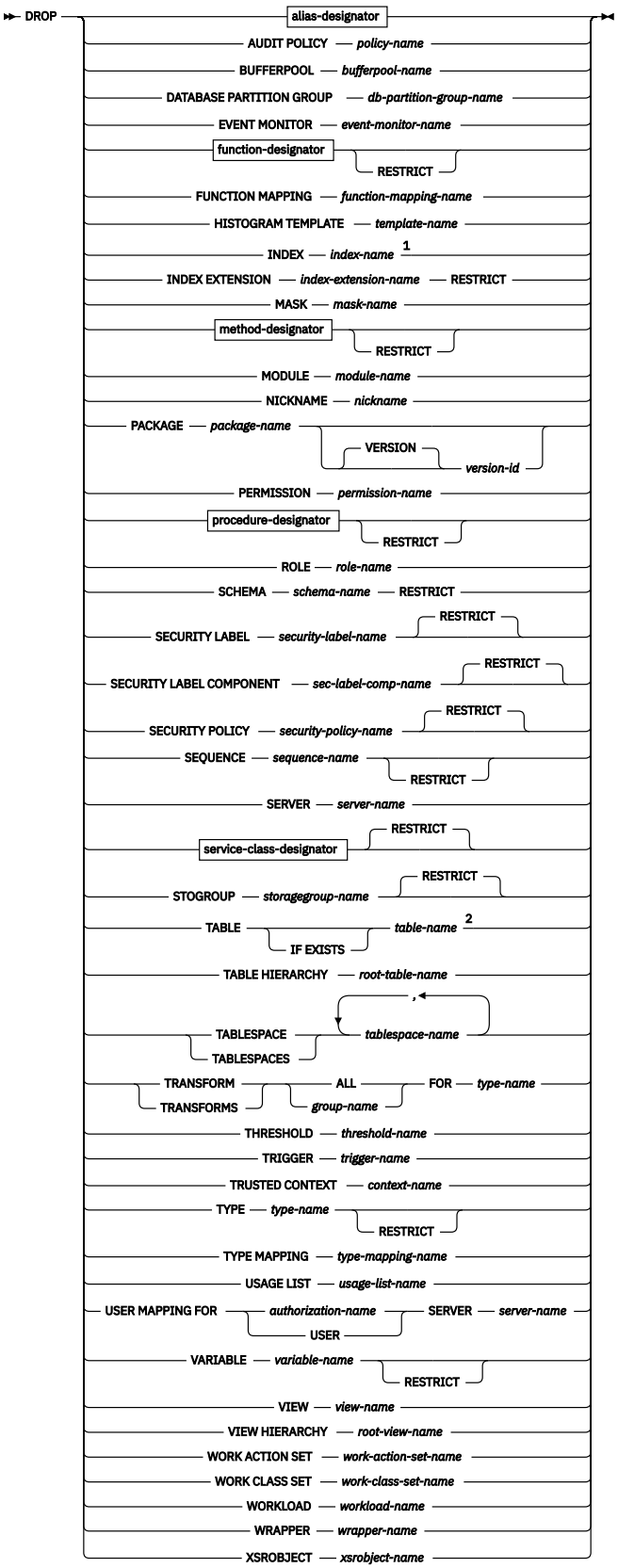
When dropping a transform, the privileges held by the authorization ID of the statement must include DBADM authority, or must be the owner of *type-name*.

When dropping a trusted context, the privileges held by the authorization ID of the statement must include SECADM authority.

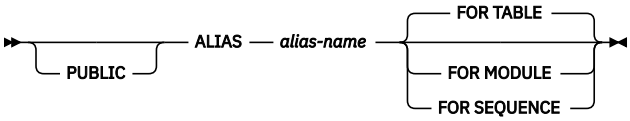
When dropping an event monitor or usage list the privilege held by the authorization ID of the statement must include SQLADM or DBADM authority.

When dropping a user mapping, the privileges held by the authorization ID of the statement must include DBADM authority, if this authorization ID is different from the federated database authorization name within the mapping. Otherwise, if the authorization ID and the authorization name match, no authorities or privileges are required.

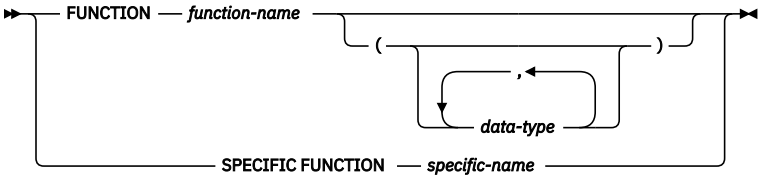
Syntax



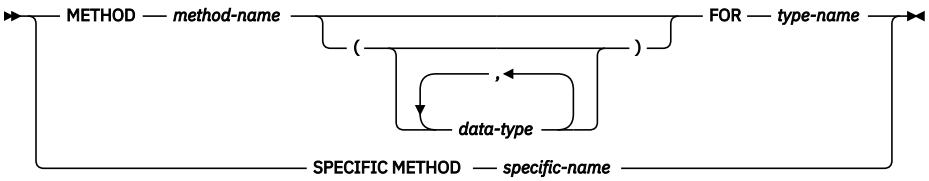
alias-designator



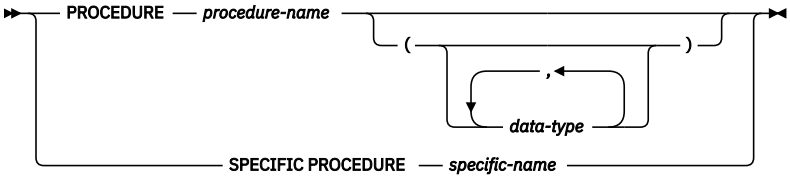
function-designator



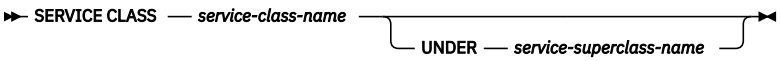
method-designator



procedure-designator



service-class-designator



Notes:

[1] *Index-name* can be the name of either an index or an index specification.

[2] For compatibility with Netezza, you can change the order of IF EXISTS and *table-name*.

Description

alias-designator

ALIAS *alias-name*

Identifies the alias that is to be dropped. The *alias-name* must identify an alias that is described in the catalog (SQLSTATE 42704). The specified alias is deleted.

FOR TABLE, FOR MODULE, or FOR SEQUENCE

Specifies the object type for the alias.

FOR TABLE

The alias is for a table, view, or nickname.

FOR MODULE

The alias is for a module.

FOR SEQUENCE

The alias is for a sequence.

All views and triggers that reference the alias are made inoperative. This includes alias references in both the ON clause of the CREATE TRIGGER statement and within the triggered SQL statements. Any materialized query table or staging table that references the alias is dropped.

If PUBLIC is specified, the *alias-name* must identify a public alias (SQLSTATE 428EK) that exists at the current server (SQLSTATE 42704).

If the alias is referenced in the definition of a row permission or a column mask, the alias cannot be dropped (SQLSTATE 42893).

AUDIT POLICY *policy-name*
Identifies the audit policy that is to be dropped. The *policy-name* must identify an audit policy that exists at the current server (SQLSTATE 42704). The audit policy must not be associated with any database objects (SQLSTATE 42893). The specified audit policy is deleted from the catalog.

BUFFERPOOL *bufferpool-name*
Identifies the buffer pool that is to be dropped. The *bufferpool-name* must identify a buffer pool that is described in the catalog (SQLSTATE 42704). There can be no table spaces assigned to the buffer pool (SQLSTATE 42893). The IBMDEFAULTBP buffer pool cannot be dropped (SQLSTATE 42832).

Buffer pool memory is released immediately. Disk storage may not be released until the next connection to the database.

DATABASE PARTITION GROUP *db-partition-group-name*
Identifies the database partition group that is to be dropped. The *db-partition-group-name* parameter must identify a database partition group that is described in the catalog (SQLSTATE 42704). This is a one-part name.

Dropping a database partition group drops all table spaces defined in the database partition group. All existing database objects with dependencies on the tables in the table spaces (such as packages, referential constraints, and so on) are dropped or invalidated (as appropriate), and dependent views and triggers are made inoperative.

IBMCATGROUP, IBMDEFAULTGROUP, and IBMTEMPGROUP database partition groups cannot be dropped (SQLSTATE 42832).

If a DROP DATABASE PARTITION GROUP statement is issued against a database partition group that is currently undergoing a data redistribution, the drop database partition group operation fails, and an error is returned (SQLSTATE 55038). However, a partially redistributed database partition group can be dropped. A database partition group can become partially redistributed if a REDISTRIBUTE DATABASE PARTITION GROUP command does not execute to completion. This can happen if it is interrupted by either an error or a FORCE APPLICATION ALL command. (For a partially redistributed database partition group, the REDISTRIBUTE_PMAP_ID in the SYSCAT.DBPARTITIONGROUPS catalog is not -1.)

EVENT MONITOR *event-monitor-name*
Identifies the event monitor that is to be dropped. The *event-monitor-name* must identify an event monitor that is described in the catalog (SQLSTATE 42704).

If the identified event monitor is active, an error is returned (SQLSTATE 55034); otherwise, the event monitor is deleted. Note that if an event monitor has been previously activated using the SET EVENT MONITOR STATE statement, and the database has been deactivated and subsequently reactivated, use the SET EVENT MONITOR STATE statement to deactivate the event monitor before issuing the DROP statement.

If there are event files in the target path of a WRITE TO FILE event monitor that is being dropped, the event files are not deleted. However, if a new event monitor that specifies the same target path is created, the event files are deleted.

When dropping WRITE TO TABLE event monitors, table information is removed from the SYSCAT.EVENTTABLES catalog view, but the tables themselves are not dropped.

function-designator
Identifies an instance of a user-defined function (either a complete function or a function template) that is to be dropped. For more information, see "Function, method, and procedure designators" on page 745.

The function instance specified must be a user-defined function described in the catalog. The following functions cannot be dropped:

- A function implicitly generated by a CREATE TYPE statement (SQLSTATE 42917)

- A function that is in the SYSIBM, SYSFUN, SYSIBMADM, or the SYSPROC schema (SQLSTATE 42832)
- A function that is referenced in the definition of a row permission or a column mask (SQLSTATE 42893)
- A function that is referenced in a generated column expression or a check constraint (SQLSTATE 42893)

RESTRICT
The RESTRICT keyword enforces the rule that the function is not to be dropped if any of the following dependencies exists:

- Another function is sourced on the function.
- Another routine uses the function.
- A view uses the function.
- A trigger uses the function.
- A materialized query table uses the function in its definition.

The restrict rule is enforced by default for the same dependencies as in version 9.5 if the **auto_reval** database configuration parameter is set to disabled.

In this case, the following considerations apply:

- Other objects can be dependent upon a function. All such dependencies must be removed before the function can be dropped, with the exception of packages which are marked inoperative. An attempt to drop a function with such dependencies will result in an error (SQLSTATE 42893). See the "Rules" section for a list of these dependencies. If the function can be dropped, it is dropped.
- Any package dependent on the specific function being dropped is marked as inoperative. Such a package is not implicitly rebound. It must either be rebound by use of the BIND or REBIND command, or it must be re-prepared by use of the PREP command.

FUNCTION MAPPING *function-mapping-name*
Identifies the function mapping that is to be dropped. The *function-mapping-name* must identify a user-defined function mapping that is described in the catalog (SQLSTATE 42704). The function mapping is deleted from the database.

Default function mappings cannot be dropped, but can be disabled by using the CREATE FUNCTION MAPPING statement. Dropping a user-defined function mapping that was created to override a default function mapping reinstates the default function mapping.

Packages having a dependency on a dropped function mapping are invalidated.

HISTOGRAM TEMPLATE *template-name*
Identifies the histogram template that is to be dropped. The *template-name* must identify a histogram template that exists at the current server (SQLSTATE 42704). The *template-name* cannot be SYSDEFAULTHISTOGRAM (SQLSTATE 42832). The histogram template cannot be dropped if a service class or a work action is dependent on it (SQLSTATE 42893). The specified histogram template is deleted from the catalog.

INDEX *index-name*
Identifies the index or index specification that is to be dropped. The *index-name* must identify an index or index specification that is described in the catalog (SQLSTATE 42704). It cannot be an index that is required by the system for a primary key or unique constraint, for a replicated materialized query table, or for an XML column (SQLSTATE 42917). The specified index or index specification is deleted.

Modification state indexes (also known as mod state indexes) can be dropped, even though they are classified as system indexes. Dropping modification state indexes is supported in order to facilitate rollback to an earlier fix pack level. If a modification state index exists when dropping the last user index on a table, the modification state index is implicitly dropped.

Packages having a dependency on a dropped index or index specification are invalidated.

INDEX EXTENSION *index-extension-name* RESTRICT
Identifies the index extension that is to be dropped. The *index-extension-name* must identify an index extension that is described in the catalog (SQLSTATE 42704). The RESTRICT keyword enforces the rule that no index can be defined that depends on this index extension definition (SQLSTATE 42893).

MASK *mask-name*
Identifies the column mask to drop. The name must identify a column mask that exists at the current server (SQLSTATE 42704).

method-designator
Identifies a method body that is to be dropped. For more information, see "Function, method, and procedure designators" on page 745. The method body specified must be a method described in the catalog (SQLSTATE 42704). Method bodies that are implicitly generated by the CREATE TYPE statement cannot be dropped.

DROP METHOD deletes the body of a method, but the method specification (signature) remains as a part of the definition of the subject type. After dropping the body of a method, the method specification can be removed from the subject type definition by ALTER TYPE DROP METHOD.

RESTRICT
The RESTRICT keyword enforces the rule that the method is not to be dropped if any of the following dependencies exists:

- A function is sourced on the method.
- Another routine uses the method.
- A view uses the method.
- A trigger uses the method.
- A materialized query table uses the method in its definition.

The restrict rule is enforced by default for the same dependencies as in version 9.5 if the **auto_reval** database configuration parameter is set to disabled.

In this case, the following considerations apply:

- Other objects can be dependent upon a method. All such dependencies must be removed before the method can be dropped, with the exception of packages which will be marked inoperative if the drop is successful. An attempt to drop a method with such dependencies will result in an error (SQLSTATE 42893). If the method can be dropped, it will be dropped.
- Any package dependent on the specific method being dropped is marked as inoperative. Such a package is not implicitly re-bound. Either it must be re-bound by use of the BIND or REBIND command, or it must be re-prepared by use of the PREP command.

If the specific method being dropped overrides another method, all packages dependent on the overridden method - and on methods that override this method in supertypes of the specific method being dropped - are invalidated.

MODULE *module-name*
Identifies the module that is to be dropped. The *module-name* must identify a module that exists at the current server (SQLSTATE 42704). The specified name must not be an alias for a module (SQLSTATE 560CT). The specified module is dropped from the schema, including all module objects. All privileges on the module are also dropped.

If the module is referenced in the definition of a row permission or a column mask, the module cannot be dropped (SQLSTATE 42893).

NICKNAME *nickname*
Identifies the nickname that is to be dropped. The nickname must be listed in the catalog (SQLSTATE 42704). The nickname is deleted from the database.

All information about the columns and indexes associated with the nickname is deleted from the catalog. Any materialized query tables that are dependent on the nickname are dropped. Any index specifications that are dependent on the nickname are dropped. Any views that are dependent on the nickname are marked inoperative. Any packages that are dependent on the dropped

index specifications or inoperative views are invalidated. The data source table that the nickname references is not affected.

If an SQL function or method is dependent on a nickname, that nickname cannot be dropped (SQLSTATE 42893).

PACKAGE *package-name*
Identifies the package that is to be dropped. The package name must identify a package that is described in the catalog (SQLSTATE 42704). The specified package is deleted. If the package being dropped is the only package identified by *package-name* (that is, there are no other versions), all privileges on the package are also deleted.

VERSION *version-id*
Identifies which package version is to be dropped. If a value is not specified, the version defaults to the empty string. If multiple packages with the same package name but different versions exist, only one package version can be dropped in one invocation of the DROP statement. Delimit the version identifier with double quotation marks when it:

- Is generated by the VERSION(AUTO) precompiler option
- Begins with a digit
- Contains lowercase or mixed-case letters

If the statement is invoked from an operating system command prompt, precede each double quotation mark delimiter with a back slash character to ensure that the operating system does not strip the delimiters.

PERMISSION *permission-name*
Identifies the row permission to drop. The name must identify a row permission that exists at the current server (SQLSTATE 42704). The name must not identify the default row permission that was created implicitly by the database manager (SQLSTATE 42917).

procedure-designator
Identifies an instance of a procedure that is to be dropped. For more information, see "Function, method, and procedure designators" on page 745. The procedure instance specified must be a procedure described in the catalog. It is not possible to drop a procedure that is in the SYSIBM, SYSFUN, SYSIBMADM, or the SYSPROC schema (SQLSTATE 42832).

RESTRICT
The RESTRICT keyword prevents the procedure from being dropped if a trigger definition or an SQL routine definition contains a CALL identifying the procedure.

The restrict rule is enforced by default for the same dependencies as in version 9.5 if the following conditions are met:

- The **auto_reval** database configuration parameter is set to disabled
- An inlined trigger definition, inlined SQL function definition, or inlined SQL method definition contains a CALL statement identifying the procedure

It is not possible to drop a procedure that is in the SYSIBM, SYSFUN, or the SYSPROC schema (SQLSTATE 42832).

ROLE *role-name*
Identifies the role that is to be dropped. The *role-name* must identify a role that already exists at the current server (SQLSTATE 42704). The *role-name* must not identify a role, or a role that contains *role-name*, if the role has either EXECUTE privilege on a routine or USAGE privilege on a sequence, and an SQL object other than a package is dependent on the routine or sequence (SQLSTATE 42893). The owner of the SQL object is either *authorization-name* or any user who is a member of *authorization-name*, where *authorization-name* is a role.

A DROP ROLE statement fails (SQLSTATE 42893) if any of the following conditions are true for the role to be dropped:

- A workload exists such that one of the values for the connection attribute SESSION_USER ROLE is *role-name*

- A trusted context using *role-name* exists

The specified role is deleted from the catalog.

SCHEMA *schema-name* RESTRICT
Identifies the particular schema to be dropped. The *schema-name* must identify a schema that is described in the catalog (SQLSTATE 42704).

RESTRICT
The RESTRICT keyword enforces the rule that no objects can be defined in the specified schema for the schema to be deleted from the database (SQLSTATE 42893).

SECURITY LABEL *security-label-name*
Identifies the security label to be dropped. The name must be qualified with a security policy (SQLSTATE 42704) and must identify a security label that exists at the current server (SQLSTATE 42704).

RESTRICT
This option, which is the default, prevents the security label from being dropped if any of the following dependencies exist (SQLSTATE 42893):

- One or more authorization IDs currently hold the security label for read access
- One or more authorization IDs currently hold the security label for write access
- The security label is currently being used to protect one or more columns

SECURITY LABEL COMPONENT *sec-label-comp-name*
Identifies the security label component to be dropped. The *sec-label-comp-name* must identify a security label component that is described in the catalog (SQLSTATE 42704).

RESTRICT
This option, which is the default, prevents the security label component from being dropped if any of the following dependencies exist (SQLSTATE 42893):

- One or more security policies that include the security label component are currently defined

SECURITY POLICY *security-policy-name*
Identifies the security policy to be dropped. The *security-policy-name* must identify a security policy that exists at the current server (SQLSTATE 42704).

RESTRICT
This option, which is the default, prevents the security policy from being dropped if any of the following dependencies exist (SQLSTATE 42893):

- One or more tables are associated with this security policy
- One or more authorization IDs hold an exemption on one of the rules in this security policy
- One or more security labels are defined for this security policy

SEQUENCE *sequence-name*
Identifies the particular sequence that is to be dropped. The *sequence-name*, along with the implicit or explicit schema name, must identify an existing sequence at the current server. If no sequence by this name exists in the explicitly or implicitly specified schema, an error is returned (SQLSTATE 42704).

RESTRICT
The RESTRICT keyword prevents the sequence from being dropped if any of the following dependencies exist:

- A trigger exists such that a NEXT VALUE or PREVIOUS VALUE expression in the trigger body specifies the sequence (SQLSTATE 42893).
- An SQL routine exists such that a NEXT VALUE expression in the routine body specifies the sequence (SQLSTATE 42893).

The restrict rule is enforced by default for the same dependencies as in version 9.5 if the following conditions are met:

- The **auto_reval** database configuration parameter is set to disabled
- An inlined trigger definition, inlined SQL function definition, or inlined SQL method definition references the sequence

SERVER ***server-name***
Identifies the data source whose definition is to be dropped from the catalog. The *server-name* must identify a data source that is described in the catalog (SQLSTATE 42704). The definition of the data source is deleted.

All nicknames for tables and views residing at the data source are dropped. Any index specifications dependent on these nicknames are dropped. Any user-defined function mappings, user-defined type mappings, and user mappings that are dependent on the dropped server definition are also dropped. All packages dependent on the dropped server definition, function mappings, nicknames, and index specifications are invalidated. All federated procedures that are dependent on the server definition are also dropped.

service-class-designator

SERVICE CLASS ***service-class-name***
Identifies the service class to be dropped. The *service-class-name* must identify a service class that is described in the catalog (SQLSTATE 42704). To drop a service subclass, the *service-superclass-name* must be specified using the UNDER clause.

UNDER ***service-superclass-name***
Specifies the service superclass of the service subclass when dropping a service subclass. The *service-superclass-name* must identify a service superclass that is described in the catalog (SQLSTATE 42704).

RESTRICT
This keyword enforces the rule that the service class is not to be dropped if any of the following dependencies exists:

- The service class is a service subclass and there is a work action mapping to the service class (SQLSTATE 5U031). The work action must first be dropped.
- The service class is the target of a REMAP ACTIVITY action in a threshold (SQLSTATE 5U031). Alter the threshold to set a different service subclass as the target of the REMAP ACTIVITY action or drop the threshold.
- The service class is not disabled (SQLSTATE 5U031). The service class must first be disabled.

RESTRICT is the default behavior.

STOGROUP ***storagegroup-name***

Identifies the storage group that is to be dropped; *storagegroup-name* must identify a storage group that exists at the current server (SQLSTATE 42704). This is a one-part name.

RESTRICT
The RESTRICT keyword prevents the storage group from being dropped if a table space exists that uses the storage group (SQLSTATE 42893). RESTRICT is the default behavior.

The current default storage group cannot be dropped (SQLSTATE 42893). A new default can be designated using the ALTER STOGROUP statement.

The DROP STOGROUP statement cannot be executed while a database partition server is being added (SQLSTATE 55071).

TABLE ***table-name***

Identifies the base table, created temporary table, or declared temporary table that is to be dropped. The *table-name* must identify a table that is described in the catalog or, if it is a declared temporary table, the *table-name* must be qualified by the schema name SESSION and exist in the application (SQLSTATE 42704). The subtables of a typed table are dependent on their supertables. All subtables must be dropped before a supertable can be dropped (SQLSTATE 42893). The *table-name* must

not identify a catalog table (SQLSTATE 42832), or a history table associated with a system-period temporal table (SQLSTATE 42893). The specified table is deleted from the database.

All indexes, primary keys, foreign keys, row permissions (including the default row permission), column masks, check constraints, materialized query tables, and staging tables that are defined on the table are dropped. All views and triggers that reference the table are made inoperative, including both the table referenced in the ON clause of the CREATE TRIGGER statement and all tables referenced within the triggered SQL statements. All packages which depend on any object dropped or marked inoperative will be invalidated. This includes packages dependent on any supertables above the subtable in the hierarchy. Any referenced columns for which the dropped table is defined as the scope of the reference become unscoped.

Packages are not dependent on declared temporary tables, and therefore are not invalidated when such a table is dropped. Packages are, however, dependent on created temporary tables, and are invalidated when such a table is dropped.

In a federated system, a remote table that was created using transparent DDL can be dropped. Dropping a remote table also drops the nickname associated with that table, and invalidates any packages that are dependent on that nickname.

When a subtable is dropped from a table hierarchy, the columns associated with the subtable are no longer accessible although they continue to be considered with respect to limits on the number of columns and size of the row. Dropping a subtable has the effect of deleting all the rows of the subtable from the supertables. This may result in activation of triggers or referential integrity constraints defined on the supertables.

When a created temporary table or declared temporary table is dropped, and its creation preceded the active unit of work or savepoint, then the table will be functionally dropped and the application will not be able to access the table. However, the table will still reserve some space in its table space and will prevent that USER TEMPORARY table space from being dropped or the database partition group of the USER TEMPORARY table space from being redistributed until the unit of work is committed or savepoint is ended. Dropping a created temporary table or declared temporary table causes the data in the table to be destroyed, regardless of whether DROP is committed or rolled back.

If *table-name* is a system-period temporal table, any associated history table and any indexes defined on the history table are also dropped. To drop a system-period temporal table, the privilege set must also contain the authorization required to drop the history table (SQLSTATE 42501).

A history table associated with a system-period temporal table cannot be explicitly dropped using the DROP statement (SQLSTATE 42893). A history table is implicitly dropped when the associated system-period temporal table is dropped.

A table cannot be dropped if it has the RESTRICT ON DROP attribute.

A newly detached table is initially inaccessible. This prevents the table from being read, modified, or dropped until the SET INTEGRITY statement can be run to incrementally refresh MQTs or to complete any processing for foreign key constraints. After the SET INTEGRITY statement executes against all dependent tables, the table is fully accessible, its detached attribute is reset, and it can be dropped.

When a table is dropped, all row permissions, including the default row permission, and column masks that are created for the table are also dropped.

If the table is referenced in the definition of a row permission or a column mask, the table cannot be dropped (SQLSTATE 42893).

IF EXISTS

Specifies that no error message is shown if the specified table name does not exist in the current database and schema.

Unless other conditions or dependencies prevent the drop operation, a successful message is returned even if no table is dropped. The condition for the failure is ignored if the table does not exist.

TABLE HIERARCHY *root-table-name*

Identifies the typed table hierarchy that is to be dropped. The *root-table-name* must identify a typed table that is the root table in the typed table hierarchy (SQLSTATE 428DR). The typed table identified by *root-table-name* and all of its subtables are deleted from the database.

All indexes, materialized query tables, staging tables, primary keys, foreign keys, and check constraints referencing the dropped tables are dropped. All views and triggers that reference the dropped tables are made inoperative. All packages depending on any object dropped or marked inoperative will be invalidated. Any reference columns for which one of the dropped tables is defined as the scope of the reference become unscoped.

Unlike dropping a single subtable, dropping the table hierarchy does not result in the activation of delete triggers of any tables in the hierarchy nor does it log the deleted rows.

TABLESPACE or TABLESPACES *tablespace-name*

Identifies the table spaces that are to be dropped; *tablespace-name* must identify a table space that is described in the catalog (SQLSTATE 42704). This is a one-part name. *tablespace-name* must not identify a table space that contains a history table unless the system-period temporal table with which it is associated is also being dropped (SQLSTATE 42893).

The table spaces will not be dropped (SQLSTATE 55024) if there is any table that stores at least one of its parts in a table space being dropped, and has one or more of its parts in another table space that is not being dropped (these tables would need to be dropped first), or if any table that resides in the table space has the RESTRICT ON DROP attribute.

Objects whose names are prefixed with 'SYS' are built-in objects and, with the exception of the SYSTOOLSPACE and SYSTOOLSTMPSPACE table spaces, cannot be dropped (SQLSTATE 42832).

A SYSTEM TEMPORARY table space cannot be dropped (SQLSTATE 55026) if it is the only temporary table space that exists in the database. A USER TEMPORARY table space cannot be dropped if there is an instance of a created temporary table or a declared temporary table created in it (SQLSTATE 55039). Even if a created temporary table has been dropped, the USER TEMPORARY table space will still be considered to be in used until all instances of the created temporary table are dropped. Instances of a created temporary table are dropped when the session terminates or when the created temporary table is referenced in the session. Even if a declared temporary table has been dropped, the USER TEMPORARY table space will still be considered to be in use until the unit of work containing the DROP TABLE statement has been committed.

Dropping a table space drops all objects that are defined in the table space. All existing database objects with dependencies on the table space, such as packages, referential constraints, and so on, are dropped or invalidated (as appropriate), and dependent views and triggers are made inoperative.

Containers that were created by a user are not deleted. Any directories in the path of the container name that were created by the database manager during CREATE TABLESPACE execution are deleted. All containers that are below the database directory are deleted. When the DROP TABLESPACE statement is committed, the DMS file containers or SMS containers for the specified table space are deleted, if possible. If the containers cannot be deleted (because they are being kept open by another agent, for example), the files are truncated to zero length. After all connections are terminated, or the DEACTIVATE DATABASE command is issued, these zero-length files are deleted.

THRESHOLD *threshold-name*

Identifies the threshold that is to be dropped. The *threshold-name* must identify a threshold that exists at the current server (SQLSTATE 42704). This is a one-part name. Thresholds with a queue, for example TOTALSCPARTITIONCONNECTIONS and CONCURRENTDBCOORDACTIVITIES, must be disabled before they can be dropped (SQLSTATE 5U025). The specified threshold is deleted from the catalog.

TRIGGER *trigger-name*

Identifies the trigger that is to be dropped. The *trigger-name* must identify a trigger that is described in the catalog (SQLSTATE 42704). The specified trigger is deleted.

Dropping triggers causes certain packages to be marked invalid.

If *trigger-name* specifies an INSTEAD OF trigger on a view, another trigger may depend on that trigger through an update against the view.

TRANSFORM ALL FOR *type-name*
Indicates that all transforms groups defined for the user-defined data type *type-name* are to be dropped. The transform functions referenced in these groups are not dropped. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *type-name* must identify a user-defined type described in the catalog (SQLSTATE 42704).

If there are not transforms defined for *type-name*, an error is returned (SQLSTATE 42740).

DROP TRANSFORM is the inverse of CREATE TRANSFORM. It causes the transform functions associated with certain groups, for a given data type, to become undefined. The functions formerly associated with these groups still exist and can still be called explicitly, but they no longer have the transform property, and are no longer invoked implicitly for exchanging values with the host language environment.

The transform group is not dropped if there is a user-defined function (or method) written in a language other than SQL that has a dependency on one of the group's transform functions defined for the user-defined type *type-name* (SQLSTATE 42893). Such a function has a dependency on the transform function associated with the referenced transform group defined for type *type-name*. Packages that depend on a transform function associated with the named transform group are marked inoperative.

TRANSFORMS *group-name* FOR *type-name*
Indicates that the specified transform group for the user-defined data type *type-name* is to be dropped. The transform functions referenced in this group are not dropped. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *type-name* must identify a user-defined type described in the catalog (SQLSTATE 42704), and the *group-name* must identify an existing transform group for *type-name*.

TRIGGER *trigger-name*
Identifies the trigger that is to be dropped. The *trigger-name* must identify a trigger that is described in the catalog (SQLSTATE 42704). The specified trigger is deleted.

Dropping triggers causes certain packages to be marked invalid.

If *trigger-name* specifies an INSTEAD OF trigger on a view, another trigger may depend on that trigger through an update against the view.

TRUSTED CONTEXT *context-name*
Identifies the trusted context that is to be dropped. The *context-name* must identify a trusted context that exists at the current server (SQLSTATE 42704). If the trusted context is dropped while trusted connections for this context are active, those connections remain trusted until they terminate or until the next reuse attempt. If an attempt is made to switch the user on these trusted connections, an error is returned (SQLSTATE 42517). The specified trusted context is deleted from the catalog.

TYPE *type-name*
Identifies the user-defined type to be dropped. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. For a structured type, the associated reference type is also dropped. The *type-name* must identify a user-defined type described in the catalog.

RESTRICT
The type is not dropped (SQLSTATE 42893) if any of the following conditions are true:

- The type is used as the type of a column of a table or view.
- The type has a subtype.

- The type is a structured type used as the data type of a typed table or a typed view.
- The type is an attribute of another structured type.
- There exists a column of a table whose type might contain an instance of *type-name*. This can occur if *type-name* is the type of the column or is used elsewhere in the column's associated type hierarchy. More formally, for any type T, T cannot be dropped if there exists a column of a table whose type directly or indirectly uses *type-name*.
- The type is the target type of a reference-type column of a table or view, or a reference-type attribute of another structured type.
- The type, or a reference to the type, is a parameter type or a return value type of a function or method.
- The type is a parameter type or is used in the body of an SQL procedure.
- The type, or a reference to the type, is used in the body of an SQL function or method, but it is not a parameter type or a return value type.
- The type is used in a check constraint, trigger, view definition, or index extension.

If RESTRICT is not specified, the behavior is the same as RESTRICT, except for functions and methods that use the type.

The restrict rule is enforced by default for the same dependencies as in version 9.5 if the **auto_reval** database configuration parameter is set to disabled.

Functions that use the type: If the user-defined type can be dropped, then for every function, F (with specific name SF), that has parameters or a return value of the type being dropped or a reference to the type being dropped, the following DROP FUNCTION statement is effectively executed:

```
DROP SPECIFIC FUNCTION SF
```

It is possible that this statement also would cascade to drop dependent functions. If all of these functions are also in the list to be dropped because of a dependency on the user-defined type, the drop of the user-defined type will succeed (otherwise it fails with SQLSTATE 42893).

Methods that use the type: If the user-defined type can be dropped, then for every method, M of type T1 (with specific name SM), that has parameters or a return value of the type being dropped or a reference to the type being dropped, the following statements are effectively executed:

```
DROP SPECIFIC METHOD SM
ALTER TYPE T1 DROP SPECIFIC METHOD SM
```

The existence of objects that are dependent on these methods may cause the DROP TYPE operation to fail.

All packages that are dependent on methods defined in supertypes of the type being dropped, and that are eligible for overriding, are invalidated.

If the type is referenced in the definition of a row permission or a column mask, the type cannot be dropped (SQLSTATE 42893).

TYPE MAPPING *type-mapping-name*
Identifies the user-defined data type mapping to be dropped. The *type-mapping-name* must identify a data type mapping that is described in the catalog (SQLSTATE 42704). The data type mapping is deleted from the database.

No additional objects are dropped.

USAGE LIST *usage-list-name*
Identifies the usage list that is to be dropped. The *usage-list-name*, including the implicit or explicit qualifier, must identify a usage list that is described in the catalog (SQLSTATE 42704). Memory allocated for the usage list is released and is not under transactional control.

USER MAPPING FOR *authorization-name* | USER SERVER *server-name*
Identifies the user mapping to be dropped. This mapping associates an authorization name that is used to access the federated database with an authorization name that is used to access a data

source. The first of these two authorization names is either identified by the *authorization-name* or referenced by the special register USER. The *server-name* identifies the data source that the second authorization name is used to access.

The *authorization-name* must be listed in the catalog (SQLSTATE 42704). The *server-name* must identify a data source that is described in the catalog (SQLSTATE 42704). The user mapping is deleted.

No additional objects are dropped.

VARIABLE *variable-name*
Identifies the global variable that is to be dropped. The *variable-name* must identify a global variable that exists at the current server (SQLSTATE 42704).

If the variable is referenced in the definition of a row permission or a column mask, the variable cannot be dropped (SQLSTATE 42893).

RESTRICT
The RESTRICT keyword prevents the global variable from being dropped if it is referenced in an SQL routine definition, trigger definition, or view definition (SQLSTATE 42893).

The restrict rule is enforced by default for the same dependencies as in version 9.5 if the following conditions are met:

- The **auto_reval** database configuration parameter is set to disabled
- An inlined trigger definition, inlined SQL function definition, inlined SQL method definition, or view references the variable

VIEW *view-name*
Identifies the view that is to be dropped. The *view-name* must identify a view that is described in the catalog (SQLSTATE 42704). The subviews of a typed view are dependent on their superviews. All subviews must be dropped before a superview can be dropped (SQLSTATE 42893).

The specified view is deleted. The definition of any view or trigger that is directly or indirectly dependent on that view is marked inoperative. Any materialized query table or staging table that is dependent on any view that is marked inoperative is dropped. Any packages dependent on a view that is dropped or marked inoperative will be invalidated. This includes packages dependent on any superviews above the subview in the hierarchy. Any reference columns for which the dropped view is defined as the scope of the reference become unscoped.

If the view is referenced in the definition of a row permission or a column mask, the view cannot be dropped (SQLSTATE 42893).

VIEW HIERARCHY *root-view-name*
Identifies the typed view hierarchy that is to be dropped. The *root-view-name* must identify a typed view that is the root view in the typed view hierarchy (SQLSTATE 428DR). The typed view identified by *root-view-name* and all of its subviews are deleted from the database.

The definition of any view or trigger that is directly or indirectly dependent on any of the dropped views is marked inoperative. Any packages dependent on any view or trigger that is dropped or marked inoperative will be invalidated. Any reference columns for which a dropped view or view marked inoperative is defined as the scope of the reference become unscoped.

WORK ACTION SET *work-action-set-name*
Identifies the work action set that is to be dropped. The *work-action-set-name* must identify a work action set that exists at the current server (SQLSTATE 42704). All work actions that are contained by the *work-action-set-name* are also dropped.

WORK CLASS SET *work-class-set-name*
Identifies the work class set that is to be dropped. The *work-class-set-name* must identify a work class set that exists at the current server (SQLSTATE 42704). All work classes that are contained by the *work-class-set-name* are also dropped.

WORKLOAD *workload-name*
Identifies the workload that is to be dropped. This is a one-part name. The *workload-name* must identify a workload that exists at the current server (SQLSTATE 42704). SYSDEFAULTUSERWORKLOAD or SYSDEFAULTADMWORKLOAD cannot be dropped (SQLSTATE 42832). A workload must be disabled and must not have active workload occurrences associated with it before it can be dropped (SQLSTATE 5U023). To drop a workload with an associated threshold (SQLSTATE 5U031), you must drop the threshold first. The specified workload is deleted from the catalog.

WRAPPER *wrapper-name*
Identifies the wrapper to be dropped. The *wrapper-name* must identify a wrapper that is described in the catalog (SQLSTATE 42704). The wrapper is deleted.

All server definitions, user-defined function mappings, and user-defined data type mappings that are dependent on the wrapper are dropped. All user-defined function mappings, nicknames, user-defined data type mappings, and user mappings that are dependent on the dropped server definitions are also dropped. Any index specifications dependent on the dropped nicknames are dropped, and any views dependent on these nicknames are marked inoperative. All packages dependent on the dropped objects and inoperative views are invalidated. All federated procedures that are dependent on the dropped server definitions are also dropped.

XSROBJECT *xsrobject-name*
Identifies the XSR object to be dropped. The *xsrobject-name* must identify an XSR object that is described in the catalog (SQLSTATE 42704).

Check constraints that reference the XSR object are dropped. All triggers and views referencing the XSR object are marked inoperative. Packages having a dependency on a dropped XSR object are invalidated.

In a partitioned database environment, you can issue this statement against an XSR object by connecting to any partition.

Rules

Dependencies: Table 149 on page 1632 shows the dependencies that objects have on each other. Not all dependencies are explicitly recorded in the catalog. For example, there is no record of the constraints on which a package has dependencies. Four different types of dependencies are shown:

R
Restrict semantics. The underlying object cannot be dropped as long as the object that depends on it exists.

C
Cascade semantics. Dropping the underlying object causes the object that depends on it (the depending object) to be dropped as well. However, if the depending object cannot be dropped because it has a Restrict dependency on some other object, the drop of the underlying object will fail.

X
Inoperative semantics. Dropping the underlying object causes the object that depends on it to become inoperative. It remains inoperative until a user takes some explicit action.

A
Automatic invalidation and revalidation semantics. Dropping the underlying object causes the object that depends on it to become invalid. The database manager attempts to revalidate the invalid object.

A package used by a function or a method, or by a procedure that is called directly or indirectly from a function or method, will only be automatically revalidated if the routine is defined as MODIFIES SQL DATA. If the routine is not MODIFIES SQL DATA, an error is returned (SQLSTATE 56098).

In general, the database manager attempts to revalidate the invalid objects the next time the object is used. However, in situations when **auto_reval** is set to IMMEDIATE, the impacted dependent objects will be revalidated immediately after they become invalid. Those situations are:

- ALTER TABLE ... ALTER COLUMN

- ALTER TABLE ... DROP COLUMN
- ALTER TABLE ... RENAME COLUMN
- ALTER TYPE ... ADD ATTRIBUTE
- ALTER TYPE ... DROP ATTRIBUTE
- Any CREATE statement that specifies "OR REPLACE"

Some of the dependencies shown in Table 149 on page 1632 change to "A" (Automatic Invalidation/ Revalidation semantics) when the database configuration parameter **auto_reval** is set to IMMEDIATE or DEFERRED. Table 150 on page 1638 summarizes the dependent objects that are impacted. Objects listed in the "Impacted Dependent Objects" column will be invalidated when the corresponding statement listed in the "Statement" column is executed.

Some DROP statement parameters and objects are not shown in Table 149 on page 1632 because they would result in blank rows or columns:

- EVENT MONITOR, PACKAGE, PROCEDURE, SCHEMA, TYPE MAPPING, and USER MAPPING DROP statements do not have object dependencies.
- Alias, buffer pool, distribution key, privilege, and procedure object types do not have DROP statement dependencies.
- A DROP SERVER, DROP FUNCTION MAPPING, or DROP TYPE MAPPING statement in a given unit of work (UOW) cannot be processed under either of the following conditions:
  - The statement references a single data source, and the UOW already includes a SELECT statement that references a nickname for a table or view within this data source (SQLSTATE 55006).
  - The statement references a category of data sources (for example, all data sources of a specific type and version), and the UOW already includes a SELECT statement that references a nickname for a table or view within one of these data sources (SQLSTATE 55006).

Table 149. Dependencies

Statement	Object Type																										
	CONSTRAINT	FUNCTION	FUNCTION MAPPING	GLOBAL VARIABLE	INDEX	INDEX EXTENSION	MASK	METHOD	NICKNAME	DB PARTITION GROUP	PACKAGE[31]	PERMISSION	SERVER	SERVICE CLASS	TABLE	TABLE SPACE	THRESHOLD	TRIGGER	TYPE	TYPE MAPPING	USAGE LIST	USER MAPPING	VIEW	WORK ACTION SET	WORKLOAD	XSROBJECT
ALTER FUNCTION	-	-	-	-	-	-	-	-	-	-	A	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ALTER METHOD	-	-	-	-	-	-	-	-	-	-	A	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ALTER NICKNAME, altering the local name or the local type	R[33]	R	-	-	-	-	-	R	-	-	A	-	-	-	R	-	-	-	-	-	-	-	R	-	-	-
ALTER NICKNAME, altering a column option or a nickname option	-	-	-	-	-	-	-	-	-	-	A	-	-	-	R	-	-	-	-	-	-	-	-	-	-	-
ALTER NICKNAME, adding, altering, or dropping a constraint	-	-	-	-	-	-	-	-	-	-	A	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ALTER PROCEDURE	-	-	-	-	-	-	-	-	-	-	A	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ALTER SERVER	-	-	-	-	-	-	-	-	-	-	A	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 149. Dependencies (continued)

Statement	Object Type																										
	CONSTRAINT	FUNCTION	FUNCTION MAPPING	GLOBAL VARIABLE	INDEX	INDEX EXTENSION	MASK	METHOD	NICKNAME	DB PARTITION GROUP	PACKAGE[31]	PERMISSION	SERVER	SERVICE CLASS	TABLE	TABLE SPACE	THRESHOLD	TRIGGER	TYPE	TYPE MAPPING	USAGE LIST	USER MAPPING	VIEW	WORK ACTION	WORK ACTION SET	WORKLOAD	XSROBJECT
ALTER TABLE ALTER COLUMN	-	A	-	A	-	-	R	-	-	-	A	-	-	-	-	-	-	A	-	-	-	-	A	-	-	-	X 34
ALTER TABLE DROP COLUMN	C	C	-	C	C	-	R	-	-	-	-	R	-	-	-	-	-	C	-	-	-	-	C	-	-	-	X 34
ALTER TABLE DROP CONSTRAINT	C	-	-	-	-	-	-	-	-	-	A 1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ALTER TABLE DROP PARTITIONING KEY	-	-	-	-	-	-	-	-	-	R 20	A 1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ALTER TYPE ADD ATTRIBUTE	-	-	-	-	-	R	-	-	-	-	A 23	-	-	-	R 24	-	-	-	-	-	-	-	R 14	-	-	-	-
ALTER TYPE ALTER METHOD	-	-	-	-	-	-	-	-	-	-	A	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ALTER TYPE DROP ATTRIBUTE	-	-	-	-	-	R	-	-	-	-	A 23	-	-	-	R 24	-	-	-	-	-	-	-	R 14	-	-	-	-
ALTER TYPE ADD METHOD	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ALTER TYPE DROP METHOD	-	-	-	-	-	-	-	R 27	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CREATE METHOD	-	-	-	-	-	-	-	-	-	-	A 28	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CREATE TYPE	-	-	-	-	-	-	-	-	-	-	A 29	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
DROP ALIAS	-	R	-	R	-	-	R	-	-	-	A 3	R	-	-	C 3	-	-	X 3	-	-	-	-	X 3	-	-	-	-
DROP BUFFERPOOL	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	R	-	-	-	-	-	-	-	-	-	-	-
DROP DATABASE PARTITION GROUP	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	C	-	-	-	-	-	-	-	-	-	-	-
DROP FUNCTION	R	R 7	R	R	-	R	R	R 7	-	-	X	R	-	-	R	-	-	R	-	-	-	-	R	-	-	-	-
DROP FUNCTION MAPPING	-	-	-	-	-	-	-	-	-	-	A	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
DROP INDEX	R	-	-	-	-	-	-	-	-	-	A	-	-	-	-	-	-	-	-	-	C 37	-	R 17	-	-	-	-
DROP INDEX EXTENSION	-	R	-	R	R	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
DROP MASK	-	-	-	-	-	-	-	-	-	-	A 39	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
DROP METHOD	R	R 7	R	R	-	R	-	R	-	-	X, A 30	-	-	-	R	-	-	R	-	-	-	-	R	-	-	-	-
DROP NICKNAME	-	R	-	R	C	-	-	R	-	-	A	-	-	-	C 11	-	-	-	-	-	-	-	X 16	-	-	-	-

Table 149. Dependencies (continued)

Statement	Object Type																										
	CONSTRAINT	FUNCTION	FUNCTION MAPPING	GLOBAL VARIABLE	INDEX	INDEX EXTENSION	MASK	METHOD	NICKNAME	DB PARTITION GROUP	PACKAGE[31]	PERMISSION	SERVER	SERVICE CLASS	TABLE	TABLE SPACE	THRESHOLD	TRIGGER	TYPE	TYPE MAPPING	USAGE LIST	USER MAPPING	VIEW	WORK ACTION	WORK ACTION SET	WORKLOAD	XSROBJECT
DROP PERMISSION	-	-	-	-	-	-	-	-	-	-	A 40	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
DROP PROCEDURE	-	R 7	-	R	-	-	-	R 7	-	-	A	-	-	-	-	-	-	R	-	-	-	-	-	-	-	-	-
DROP SEQUENCE	-	R	-	-	-	-	-	R	-	-	A	-	-	-	-	-	-	R	-	-	-	-	-	-	-	-	-
DROP SERVER	-	C 21	C 19	-	-	-	-	-	C	-	A	-	-	-	-	-	-	-	-	C 19	-	C	-	-	-	-	-
DROP SERVICE CLASS	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	R 35	-	-	-	-	-	-	R 35	-	R 35	-
DROP STOGROUP	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	R	-	-	-	-	-	-	-	-	-	-	-
DROP TABLE[32]	C	R	-	R	C	-	R	-	-	-	A 9	R	-	-	R, C 11	-	-	X 16	-	-	C 37	-	X 16	-	-	-	X 34
DROP TABLE HIERARCHY	C	R	-	R	C	-	-	-	-	-	A 9	-	-	-	R, C 11	-	-	X 16	-	-	-	-	X 16	-	-	-	-
DROP TABLESPACE	-	-	-	-	C 6	-	-	-	-	-	-	-	-	-	C, R 6	-	-	-	-	-	-	-	-	-	-	-	-
DROP TRANSFORM	-	R	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
DROP TRIGGER	-	-	-	-	-	-	-	-	-	-	A 1	-	-	-	-	-	-	X 26	-	-	-	-	-	-	-	-	-
DROP TYPE	R 13	R 5	-	R	-	R	-	-	-	-	A 12	-	-	-	R 18	-	-	R 13	R 4	-	-	-	R 14	-	-	-	-
DROP VARIABLE	-	-	R	R	-	-	R	R	-	-	A	R	-	-	-	-	-	R	-	-	-	-	R	-	-	-	-
DROP VIEW	-	R	-	R	-	-	R	-	-	-	A 2	R	-	-	-	-	-	X 16	-	-	-	-	X 15	-	-	-	-
DROP VIEW HIERARCHY	-	R	-	R	-	-	-	-	-	-	A 2	-	-	-	-	-	-	X 16	-	-	-	-	X 16	-	-	-	-
DROP WORK CLASS SET	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	R 36	-	-
DROP WRAPPER	-	-	C	-	-	-	-	-	-	-	-	-	C	-	-	-	-	-	-	C	-	-	-	-	-	-	-
DROP XSROBJECT	C	-	-	-	-	-	-	-	-	-	A	-	-	-	-	-	-	X	-	-	-	-	X	-	-	-	-
REVOKE a privilege[10]	-	C, R 25	-	-	-	-	R 38	C, R 25	-	-	A 1	R 38	-	-	C, X 8	-	-	X	-	-	-	-	X 8	-	-	-	-

1

This dependency is implicit in depending on a table with these constraints, triggers, or a distribution key.

2

If a package has an INSERT, UPDATE, or DELETE statement acting upon a view, then the package has an insert, update or delete usage on the underlying base table of the view. In the case of UPDATE, the package has an update usage on each column of the underlying base table that is modified by the UPDATE.

If a package has a statement acting on a typed view, creating or dropping any view in the same view hierarchy will invalidate the package.

3

If a package, materialized query table, staging table, view, or trigger uses an alias, it becomes dependent both on the alias and the object that the alias references. If the alias is in a chain, a dependency is created on each alias in the chain.

Aliases themselves are not dependent on anything. It is possible for an alias to be defined on an object that does not exist.

4

A user-defined type T can depend on another user-defined type B, if T:

- names B as the data type of an attribute
- has an attribute of REF(B)
- has B as a supertype.

5

If the user-defined type is referenced as a function parameter type or return type, then the type will be dropped and its catalog data will be maintained due to the routine parameter dependency. A value 'X' in the VALID column of the SYSCAT.DATATYPES catalog view indicates this dropped type. Its catalog data will be deleted by a DROP FUNCTION statement if the DROP FUNCTION statement also dropped the last routine parameter dependency on this type, or will be deleted by a CREATE TYPE statement with the same schema name, module name, and type name. If the user-defined type is a structured type, any methods that are associated with the type are also dropped.

6

Dropping a table space or a list of table spaces causes all the tables that are completely contained within the given table space or list to be dropped. However, if a table spans table spaces (indexes, long columns, or data partitions in different table spaces) and those table spaces are not in the list being dropped, the table spaces cannot be dropped as long as the table exists.

7

A function can depend on another specific function if the depending function names the base function in a SOURCE clause. A function or method can also depend on another specific function or method if the depending routine is written in SQL and uses the base routine in its body. An external method, or an external function with a structured type parameter or returns type will also depend on one or more transform functions.

8

Only loss of SELECT privilege will cause a materialized query table to be dropped or a view to become inoperative. If the view that is made inoperative is included in a typed view hierarchy, all of its subviews also become inoperative.

9

If a package has an INSERT, UPDATE, or DELETE statement acting on table T, then the package has an insert, update or delete usage on T. In the case of UPDATE, the package has an update usage on each column of T that is modified by the UPDATE.

If a package has a statement acting on a typed table, creating or dropping any table in the same table hierarchy will invalidate the package.

10

Dependencies do not exist at the column level because privileges on columns cannot be revoked individually.

If a package, trigger or view includes the use of OUTER(*Z*) in the FROM clause, there is a dependency on the SELECT privilege on every subtable or subview of *Z*. Similarly, if a package, trigger, or view includes the use of DEREF(*Y*) where *Y* is a reference type with a target table or view *Z*, there is a dependency on the SELECT privilege on every subtable or subview of *Z*.

11

A materialized query table is dependent on the underlying tables or nicknames specified in the fullselect of the table definition.

Cascade semantics apply to dependent materialized query tables.

A subtable is dependent on its supertables up to the root table. A supertable cannot be dropped until all of its subtables are dropped.

A history table is dependent on the system-period temporal table with which it is associated. Cascade semantics apply to the history table when the system-period temporary table on which depends is dropped.

12

A package can depend on structured types as a result of using the TYPE predicate or the subtype-treatment expression (TREAT *expression* AS *data-type*). The package has a dependency on the subtypes of each structured type specified in the right side of the TYPE predicate, or the right side of the TREAT expression. Dropping or creating a structured type that alters the subtypes on which the package is dependent causes invalidation.

All packages that are dependent on methods defined in supertypes of the type being dropped, and that are eligible for overriding, are invalidated.

13

A check constraint or trigger is dependent on a type if the type is used anywhere in the constraint or trigger. There is no dependency on the subtypes of a structured type used in a TYPE predicate within a check constraint or trigger.

14

A view is dependent on a type if the type is used anywhere in the view definition (this includes the type of typed view). There is no dependency on the subtypes of a structured type used in a TYPE predicate within a view definition.

15

A subview is dependent on its superview up to the root view. A superview cannot be dropped until all its subviews are dropped. Refer to [16] for additional view dependencies.

16

A trigger or view is also dependent on the target table or target view of a dereference operation or DEREF function. A trigger or view with a FROM clause that includes OUTER(*Z*) is dependent on all the subtables or subviews of *Z* that existed at the time the trigger or view was created.

17

A typed view can depend on the existence of a unique index to ensure the uniqueness of the object identifier column.

18

A table may depend on a user defined data type (distinct or structured) because the type is:

- used as the type of a column
- used as the type of the table
- used as an attribute of the type of the table
- used as the target type of a reference type that is the type of a column of the table or an attribute of the type of the table
- directly or indirectly used by a type that is the column of the table.

19

Dropping a server cascades to drop the function mappings and type mappings created for that named server.

20

If the distribution key is defined on a table in a multiple partition database partition group, the distribution key is required.

21

If a dependent OLE DB table function has "R" dependent objects (see DROP FUNCTION), then the server cannot be dropped.

22

An SQL function or method can depend on the objects referenced by its body.

23

When an attribute A of type TA of *type-name* T is dropped, the following DROP statements are effectively executed:

```
Mutator method: DROP METHOD A (TA) FOR T
Observer method: DROP METHOD A () FOR T
ALTER TYPE T
  DROP METHOD A(TA)
  DROP METHOD A()
```

24

A table may depend on an attribute of a user-defined structured data type in the following cases:

1. The table is a typed table that is based on *type-name* or any of its subtypes.
2. The table has an existing column of a type that directly or indirectly refers to *type-name*.

25

A REVOKE of SELECT privilege on a table or view that is used in the body of an SQL function or method body causes an attempt to drop the function or method body, if the function or method body defined no longer has the SELECT privilege. If such a function or method body is used in a view, trigger, function, or method body, it cannot be dropped, and the REVOKE is restricted as a result. Otherwise, the REVOKE cascades and drops such functions.

26

A trigger depends on an INSTEAD OF trigger when it modifies the view on which the INSTEAD OF trigger is defined, and the INSTEAD OF trigger fires.

27

A method declaration of an original method that is overridden by other methods cannot be dropped (SQLSTATE 42893).

28

If the method of the method body being created is declared to override another method, all packages dependent on the overridden method, and on methods that override this method in supertypes of the method being created, are invalidated.

29

When a new subtype of an existing type is created, all packages dependent on methods that are defined in supertypes of the type being created, and that are eligible for overriding (for example, no mutators or observers), are invalidated.

30

If the specific method of the method body being dropped is declared to override another method, all packages dependent on the overridden method, and on methods that override this method in supertypes of the specific method being dropped, are invalidated.

31

Cached dynamic SQL has the same semantics as packages.

32

When a remote base table is dropped using the DROP TABLE statement, both the nickname and the remote base table are dropped.

33

A primary key or unique keys that are not referenced by a foreign key do not restrict the altering of a nickname local name or local type.

34

An XSROBJECT can become inoperative for decomposition as a result of changes to a table that is associated with the XML schema for decomposition. Changes that could impact decomposition are: dropping the table or dropping a column of the table, or changing a column of the table. The decomposition status of the XML schema can be reset by issuing an ALTER XSROBJECT statement to enable or disable decomposition for the XML schema.

35

- A service class cannot be dropped if any workload is mapped to it (SQLSTATE 5U031).
- A service subclass cannot be dropped if any work action is mapped to it (SQLSTATE 5U031).
- A service subclass cannot be explicitly dropped if it is the target of a threshold REMAP action (SQLSTATE 5U031).

Dropping a service superclass cascades to drop any thresholds, work action sets and service subclasses defined for the service class.

36

A work class set cannot be dropped until the work action set that is defined on it has been dropped.

37

Once the index or table is dropped, its usage list will be invalidated in the catalog. Revalidation will take place on the next activation of the list or it can be explicitly revalidated using the procedure ADMIN_REVALIDATE_DB_OBJECTS.

38

Revoking a privilege is restricted if it causes an object to be dropped or invalidated, and a permission or mask depends on it. For example, if you have a view which depends on a table, and a permission or mask that references the view, REVOKE SELECT on the table invalidates the view, but causes an error.

39

Packages are invalidated when a table on which the enabled permission is defined has row level access control activated on the table. Packages are not affected when dropping a permission that is disabled or is defined on a table with row access control deactivated.

40

Packages are invalidated when a table on which the enabled permission is defined has row level access control activated on the table. Packages are not affected when dropping a permission that is disabled or is defined on a table with row access control deactivated.

*Table 150. Dependent Objects Impacted by **auto_reval***

Statement	Impacted Dependent Objects
ALTER NICKNAME (altering the local name or the local type)	Anchor Type, Function, Method, Procedure, User Defined Type, Variable, View
ALTER TABLE ALTER COLUMN	Anchor Type, Function, Method, Procedure, Trigger[4], User Defined Type, Variable, View, XSROBJECT
ALTER TABLE DROP COLUMN[2]	Anchor Type, Function, Method, Index, Procedure, Trigger[4], User Defined Type, Variable, View, XSROBJECT
ALTER TABLE RENAME COLUMN[1, 3]	Anchor Type, Function, Method, Index, Procedure, Trigger[4], User Defined Type, Variable, View, XSROBJECT
ALTER TYPE ADD ATTRIBUTE	View
ALTER TYPE DROP ATTRIBUTE	View
DROP ALIAS	Anchor Type, Function, Method, Procedure, Trigger, User Defined Type, Variable, View

*Table 150. Dependent Objects Impacted by **auto_reval** (continued)*

Statement	Impacted Dependent Objects
DROP FUNCTION (ALTER MODULE DROP FUNCTION)	Function, Function Mapping, Index Extension, Method, Procedure, Trigger, Variable, View
DROP METHOD	Function, Function Mapping, Index Extension, Method, Procedure, Trigger, Variable, View
DROP NICKNAME	Anchor Type, Function, Method, Procedure, Trigger, User Defined Type, Variable, View
DROP PROCEDURE (ALTER MODULE DROP PROCEDURE)	Function, Method, Procedure, Trigger
DROP SEQUENCE	Function, Method, Procedure, Trigger, Variable, View
DROP TABLE	Anchor Type, Function, Method, Procedure, Trigger[4], User Defined Type, Variable, View, XSROBJECT
DROP TABLE HIERARCHY	Function, Method, Procedure, Trigger, Variable, View
DROP TRIGGER	Trigger
DROP TYPE (ALTER MODULE DROP TYPE)	Anchor Type, Cursor Type, Function, Method, Procedure, Index Extension, Trigger, User Defined Type, Variable, View
DROP VARIABLE (ALTER MODULE DROP VARIABLE)	Anchor Type, Function, Function Mapping, Method, Procedure, Trigger, User Defined Type, Variable, View
DROP VIEW	Anchor Type, Function, Method, Procedure, Trigger[4], User Defined Type, Variable, View
DROP VIEW HIERARCHY	Function, Procedure, Trigger, Variable, View
DROP XSROBJECT	Trigger, View
RENAME TABLE	Anchor Type, Function, Method, Procedure, Trigger[4], User Defined Type, Variable, View, XSROBJECT
REVOKE a privilege	Function, Method, Procedure, Trigger, Variable, View
CREATE OR REPLACE ALIAS[1]	Function, Trigger, Procedure, Variable, View
CREATE OR REPLACE VIEW[1]	Anchor Type, Function, Method, Procedure, Trigger[4], User Defined Type, Variable, View
CREATE OR REPLACE FUNCTION[1]	Function, Function Mapping, Index Extension, method, Procedure, Variable, View
CREATE OR REPLACE PROCEDURE[1]	Function, Method, Procedure, Trigger
CREATE OR REPLACE NICKNAME[1]	Function, method, Procedure, Variable, View
CREATE OR REPLACE SEQUENCE[1]	Function, Method, Procedure, Trigger, Variable, View
CREATE OR REPLACE VARIABLE[1]	Function, Method, Procedure, Trigger, User Defined Type, Variable, View
CREATE OR REPLACE TRIGGER[1]	Trigger

1

Immediate revalidation semantics apply for these statements (for the CREATE statements, only if OR REPLACE is specified) regardless of the setting of the **auto_reval** database configuration parameter.

2

The dependent objects listed will be revalidated the next time the object is used, except for the following objects, which will be revalidated immediately as part of the statement:

- ANCHOR TYPE
- CURSOR TYPE
- VIEW (where the select list consists only of SELECT *, and does not contain any explicitly defined view columns).

For an immediate view revalidation, the list of column names for the select list will be re-established during revalidation.

3

The dependent objects listed will be revalidated the next time the object is used except for the following objects, which will be revalidated immediately as part of the statement:

- User Defined Type
- VIEW (where the select list consists only of SELECT *, and does not contain any explicitly defined view columns).

For an immediate view revalidation, the list of column names for the select list will be re-established during revalidation.

4

If the dependency is because the trigger is defined on the table or view, then the inoperative semantics from Table 1 continue to apply. If the dependency is because the trigger body references the table or view, then automatic invalidation and revalidation semantics apply.

The DROP DATABASE PARTITION GROUP statement might fail (SQLSTATE 55071) if an add database partition server request is either pending or in progress. This statement might also fail (SQLSTATE 55077) if a new database partition server is added online to the instance and not all applications are aware of the new database partition server.

Notes

- It is valid to drop a user-defined function while it is in use. Also, a cursor can be open over a statement which contains a reference to a user-defined function, and while this cursor is open the function can be dropped without causing the cursor fetches to fail.
- If a package which depends on a user-defined function is executing, it is not possible for another authorization ID to drop the function until the package completes its current unit of work. At that point, the function is dropped and the package becomes inoperative. The next request for this package results in an error indicating that the package must be explicitly rebound.
- The removal of a function body (this is very different from dropping the function) can occur while an application which needs the function body is executing. This may or may not cause the statement to fail, depending on whether the function body still needs to be loaded into storage by the database manager on behalf of the statement.
- In addition to the dependencies recorded for any explicitly specified UDF, the following dependencies are recorded when transforms are implicitly required:
  1. When the structured type parameter or result of a function or method requires a transform, a dependency is recorded for the function or method on the required TO SQL or FROM SQL transform function.
  2. When an SQL statement included in a package requires a transform function, a dependency is recorded for the package on the designated TO SQL or FROM SQL transform function.

  Since these describe the only circumstances under which dependencies are recorded due to implicit invocation of transforms, no objects other than functions, methods, or packages can have a dependency on implicitly invoked transform functions. On the other hand, explicit calls to transform functions (in views and triggers, for example) do result in the usual dependencies of these other types of objects on

transform functions. As a result, a DROP TRANSFORM statement may also fail due to these "explicit" type dependencies of objects on the transform(s) being dropped (SQLSTATE 42893).

- Since the dependency catalogs do not distinguish between depending on a function as a transform versus depending on a function by explicit function call, it is suggested that explicit calls to transform functions are not written. In such an instance, the transform property on the function cannot be dropped, or packages will be marked inoperative, simply because they contain explicit invocations in an SQL expression.
- System created sequences for IDENTITY columns cannot be dropped using the DROP SEQUENCE statement.
- When a sequence is dropped, all privileges on the sequence are also dropped and any packages that refer to the sequence are invalidated.
- For relational nicknames, the DROP NICKNAME statement within a given unit of work (UOW) cannot be processed under either of the following conditions (SQLSTATE 55007):
  - A nickname referenced in this statement has a cursor open on it in the same UOW
  - Either an INSERT, DELETE, or UPDATE statement is already issued in the same UOW against the nickname that is referenced in this statement
- For non-relational nicknames, the DROP NICKNAME statement within a given unit of work (UOW) cannot be processed under any of the following conditions (SQLSTATE 55007):
  - A nickname referenced in this statement has a cursor open on it in the same UOW
  - A nickname referenced in this statement is already referenced by a SELECT statement in the same UOW
  - Either an INSERT, DELETE, or UPDATE statement has already been issued in the same UOW against the nickname that is referenced in this statement
- A DROP SERVER statement (SQLSTATE 55006), or a DROP FUNCTION MAPPING or DROP TYPE MAPPING statement (SQLSTATE 55007) within a given unit of work (UOW) cannot be processed under either of the following conditions:
  - The statement references a single data source, and the UOW already includes one of the following items:
    - A SELECT statement that references a nickname for a table or view within this data source
    - An open cursor on a nickname for a table or view within this data source
    - Either an INSERT, DELETE, or UPDATE statement issued against a nickname for a table or view within this data source
  - The statement references a category of data sources (for example, all data sources of a specific type and version), and the UOW already includes one of the following items:
    - A SELECT statement that references a nickname for a table or view within one of these data sources
    - An open cursor on a nickname for a table or view within one of these data sources
    - Either an INSERT, DELETE, or UPDATE statement issued against a nickname for a table or view within one of these data sources
- The DROP WORKLOAD statement does not take effect until it is committed, even for the connection that issues the statement.
- Only one of these statements can be issued by any application at a time, and only one of these statements is allowed within any one unit of work. Each statement must be followed by a COMMIT or a ROLLBACK statement before another one of these statements can be issued (SQLSTATE 5U021).
  - CREATE HISTOGRAM TEMPLATE, ALTER HISTOGRAM TEMPLATE, or DROP (HISTOGRAM TEMPLATE)
  - CREATE SERVICE CLASS, ALTER SERVICE CLASS, or DROP (SERVICE CLASS)
  - CREATE THRESHOLD, ALTER THRESHOLD, or DROP (THRESHOLD)
  - CREATE WORK ACTION, ALTER WORK ACTION, or DROP (WORK ACTION)
  - CREATE WORK CLASS, ALTER WORK CLASS, or DROP (WORK CLASS)

  - CREATE WORKLOAD, ALTER WORKLOAD, or DROP (WORKLOAD)
  - GRANT (Workload Privileges) or REVOKE (Workload Privileges)
- ***Soft invalidation:*** After the drop or change of a database object done by the following statements, active access to the dropped or changed object continues until the access is complete.
  - ALTER FUNCTION
  - ALTER MODULE ... DROP FUNCTION
  - ALTER MODULE ... DROP VARIABLE
  - ALTER TABLE ... DETACH PARTITION
  - ALTER VIEW
  - DROP ALIAS
  - DROP FUNCTION
  - DROP TRIGGER
  - DROP VARIABLE
  - DROP VIEW
  - All of the CREATE OR REPLACE statements except CREATE OR REPLACE SEQUENCE.

  This is the case when the database registry variable *DB2_DLL_SOFT_INVALID* is set to ON. When it is set to OFF, the drop or change of these objects will only complete after all active access to the object to be dropped or changed is complete.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - NODEGROUP can be specified in place of DATABASE PARTITION GROUP
  - DISTINCT TYPE *type-name* can be specified in place of TYPE *type-name*
  - DATA TYPE *type-name* can be specified in place of TYPE *type-name*
  - SYNONYM can be specified in place of ALIAS
  - PROGRAM can be specified in place of PACKAGE
- **Invalidation of packages and dynamically cached statements after dropping row permissions or column masks:** If row level access control is activated on the table, dropping an enabled row permission defined for that table invalidates all packages and dynamically cached statements that reference that same table. If column level access control is activated on the table, dropping an enabled column mask defined for that table invalidates all packages and dynamically cached statements that reference that same table. There is no invalidation for dropping disabled masks or permissions.
- **Circular dependency:** Circular dependency exists in the following example:

```
CREATE PERMISSION RP1 ON T1 FOR ROWS
     WHERE C1>(SELECT MAX(C1) FROM T2)
ENFORCED FOR ALL ACCESS
ENABLE;

CREATE PERMISSION RP2 ON T2 FOR ROWS
     WHERE C1>(SELECT MAX(C1) FROM T1)
ENFORCED FOR ALL ACCESS
ENABLE
```

  The DROP TABLE T1 and DROP TABLE T2 statements fail because **RP1** depends on **T2** and **RP2** depends on **T1**. The user with the SECADM authority should drop one of the row permissions first then issue the **DROP TABLE** statement.

Examples

1. Drop table TDEPT.

```
DROP TABLE TDEPT
```

2. Drop the view VDEPT.

```
DROP VIEW VDEPT
```

3. The authorization ID HEDGES attempts to drop an alias.

```
DROP ALIAS A1
```

   The alias HEDGES.A1 is removed from the catalogs.

4. Hedges attempts to drop an alias, but specifies T1 as the alias-name, where T1 is the name of an existing table (not the name of an alias).

```
DROP ALIAS T1
```

   This statement fails (SQLSTATE 42809).

5. Drop the BUSINESS_OPS database partition group. To drop the database partition group, the two table spaces (ACCOUNTING and PLANS) in the database partition group must first be dropped.

```
DROP TABLESPACE ACCOUNTING
DROP TABLESPACE PLANS
DROP DATABASE PARTITION GROUP BUSINESS_OPS
```

6. Pellow wants to drop the CENTER function, which he created in his PELLOW schema, using the signature to identify the function instance to be dropped.

```
DROP FUNCTION CENTER (INT,FLOAT)
```

7. McBride wants to drop the FOCUS92 function, which she created in the PELLOW schema, using the specific name to identify the function instance to be dropped.

```
DROP SPECIFIC FUNCTION PELLOW.FOCUS92
```

8. Drop the function ATOMIC_WEIGHT from the CHEM schema, where it is known that there is only one function with that name.

```
DROP FUNCTION CHEM.ATOMIC_WEIGHT
```

9. Drop the trigger SALARY_BONUS, which caused employees under a specified condition to receive a bonus to their salary.

```
DROP TRIGGER SALARY_BONUS
```

10. Drop the distinct data type named shoesize, if it is not currently in use.

```
DROP TYPE SHOESIZE
```

11. Drop the SMITHPAY event monitor.

```
DROP EVENT MONITOR SMITHPAY
```

12. Drop the schema from Example 2 under CREATE SCHEMA using RESTRICT. Notice that the table called PART must be dropped first.

```
DROP TABLE PART
DROP SCHEMA INVENTRY RESTRICT
```

13. Macdonald wants to drop the DESTROY procedure, which he created in the EIGLER schema, using the specific name found in the system catalog to identify the procedure to be dropped.

```
DROP SPECIFIC PROCEDURE  EIGLER.SQL100506102825100
```

14. Drop the procedure OSMOSIS from the BIOLOGY schema, where it is known that there is only one procedure with that name.

```
DROP PROCEDURE BIOLOGY.OSMOSIS
```

15. User SHAWN used one authorization ID to access the federated database and another to access the database at an Oracle data source called ORACLE1. A mapping was created between the two authorizations, but SHAWN no longer needs to access the data source. Drop the mapping.

```
DROP USER MAPPING FOR  SHAWN SERVER ORACLE1
```

16. An index of a data source table that a nickname references has been deleted. Drop the index specification that was created to let the optimizer know about this index.

```
DROP INDEX INDEXSPEC
```

17. Drop the MYSTRUCT1 transform group.

```
DROP TRANSFORM MYSTRUCT1 FOR POLYGON
```

18. Drop the method BONUS for the EMP data type in the PERSONNEL schema.

```
DROP METHOD BONUS (SALARY DECIMAL(10,2)) FOR PERSONNEL.EMP
```

19. Drop the sequence ORG_SEQ, with restrictions.

```
DROP SEQUENCE ORG_SEQ
```

20. A remote table EMPLOYEE was created in a federated system using transparent DDL. Access to the table is no longer needed. Drop the remote table EMPLOYEE.

```
DROP TABLE EMPLOYEE
```

21. Drop the function mapping BONUS_CALC and reinstate the default function mapping (if one exists).

```
DROP FUNCTION MAPPING BONUS_CALC
```

22. Drop the security label component LEVEL.

```
DROP SECURITY LABEL COMPONENT LEVEL
```

23. Drop the security label EMPLOYEESECLABEL of the security policy DATA_ACCESS.

```
DROP SECURITY LABEL DATA_ACCESS.EMPLOYEESECLABEL
```

24. Drop the security policy DATA_ACCESS.

```
DROP SECURITY POLICY DATA_ACCESS
```

25. Drop the security label component GROUPS.

```
DROP SECURITY LABEL COMPONENT GROUPS
```

26. Drop the XML schema EMPLOYEE located in the SQL schema HR.

```
DROP XSROBJECT HR.EMPLOYEE
```

27. Drop service subclass DOGSALES under service superclass PETSALES.

```
DROP SERVICE CLASS DOGSALES UNDER PETSALES
```

28. Drop service superclass PETSALES, which has no user-defined service subclasses. The default subclass for service class PETSALES is automatically dropped.

```
DROP SERVICE CLASS PETSALES
```

29. DROP permission P1.

```
DROP PERMISSION P1
```

30. DROP mask M1.

```
DROP MASK M1
```

31. Drop a storage group named TEST_SG.

```
DROP STOGROUP TEST_SG
```

32. Drop the usage list MON_PAYROLL

```
DROP USAGE LIST MON_PAYROLL
```

END DECLARE SECTION

The END DECLARE SECTION statement marks the end of a host variable declare section.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It must not be specified in REXX.

Authorization

None required.

Syntax

Description

The END DECLARE SECTION statement can be coded in the application program wherever declarations can appear according to the rules of the host language. It indicates the end of a host variable declaration section. A host variable section starts with a BEGIN DECLARE SECTION statement.

The BEGIN DECLARE SECTION and the END DECLARE SECTION statements must be paired and may not be nested.

Host variable declarations can be specified by using the SQL INCLUDE statement. Otherwise, a host variable declaration section must not contain any statements other than host variable declarations.

Host variables referenced in SQL statements must be declared in a host variable declare section in all host languages, other than REXX. Furthermore, the declaration of each variable must appear before the first reference to the variable.

Variables declared outside a declare section should not have the same name as variables declared within a declare section.

EXECUTE

The EXECUTE statement executes a prepared SQL statement.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

For each global variable used as an *expression* in the USING clause or in the expression for an *array-index*, the privileges held by the authorization ID of the statement must include one of the following authorities:

- READ privilege on the global variable that is not defined in a module
- EXECUTE privilege on the module of the global variable that is defined in a module
- EXECUTEIN privilege on the schema containing the module of the global variable that is defined in a module
- DATAACCESS authority on the schema containing the module of the global variable that is defined in a module

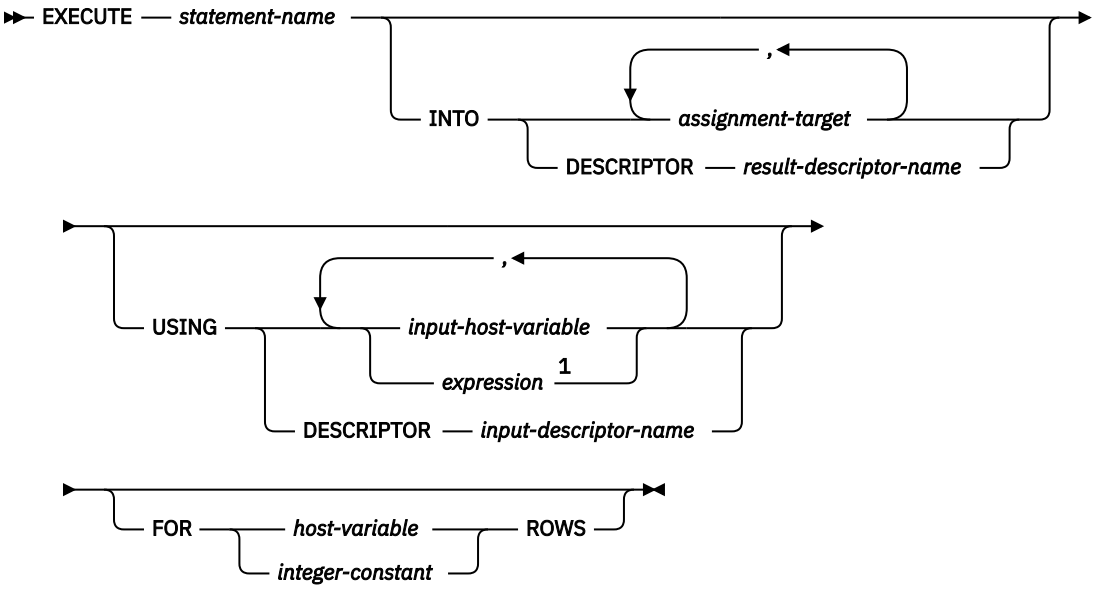
For each global variable used as an *assignment-target*, the privileges held by the authorization ID of the statement must include one of the following authorities:

- WRITE privilege on the global variable that is not defined in a module
- EXECUTE privilege on the module of the global variable that is defined in a module
- EXECUTEIN privilege on the schema containing the module of the global variable that is defined in a module
- DATAACCESS authority on the schema containing the module of the global variable that is defined in a module

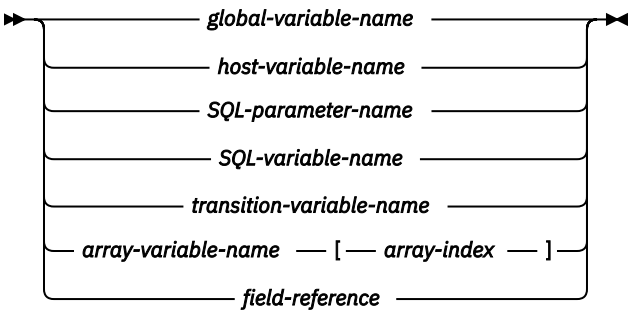
For statements where authorization checking is performed at statement execution time (DDL, GRANT, and REVOKE statements), the privileges held by the authorization ID of the statement must include those required to execute the SQL statement specified by the PREPARE statement. The authorization ID of the statement might be affected by the DYNAMICRULES bind option.

For statements where authorization checking is performed at statement preparation time (DML), no further authorization checking is performed on the SQL statement specified by the PREPARE statement.

Syntax



assignment-target



Notes:

[1] An expression other than *host-variable* can only be used when the EXECUTE statement is used within a compound SQL (compiled) statement.

Description

statement-name
Identifies the prepared statement to be executed. The *statement-name* must identify a statement that was previously prepared, and the prepared statement cannot be a SELECT statement.

INTO
Introduces a list of targets which are used to receive values from output parameter markers in the prepared statement. Each assignment to a target is made in sequence through the list. If an error occurs on any assignment, the value is not assigned to the target, and no more values are assigned to targets. Any values that have already been assigned to targets remain assigned.

For a dynamic CALL statement, parameter markers appearing in OUT and INOUT arguments to the procedure are output parameter markers. If any output parameter markers appear in the statement, the INTO clause must be specified (SQLSTATE 07007).

assignment-target
Identifies one or more targets for the assignment of output values. The first value in the result row is assigned to the first target in the list, the second value to the second target, and so on.

If the data type of an *assignment-target* is a row type, then there must be exactly one *assignment-target* specified (SQLSTATE 428HR), the number of columns must match the number of fields in the row type, and the data types of the columns of the fetched row must be assignable to the corresponding fields of the row type (SQLSTATE 42821).

If the data type of an *assignment-target* is an array element, then there must be exactly one *assignment-target* specified.

global-variable-name
Identifies the global variable that is the assignment target.

host-variable-name
Identifies the host variable that is the assignment target. For LOB output values, the target can be a regular host variable (if it is large enough), a LOB locator variable, or a LOB file reference variable.

SQL-parameter-name
Identifies the routine parameter that is the assignment target.

SQL-variable-name
Identifies the SQL variable that is the assignment target. SQL variables must be declared before they are used.

transition-variable-name
Identifies the column to be updated in the transition row. A *transition-variable-name* must identify a column in the subject table of a trigger, optionally qualified by a correlation name that identifies the new value.

array-variable-name
Identifies an SQL variable, SQL parameter, or global variable of an array type.

array-index
An expression that specifies which element in the array will be the target of the assignment. For an ordinary array, the *array-index* expression must be assignable to INTEGER (SQLSTATE 428H1) and cannot be the null value. Its value must be between 1 and the maximum cardinality defined for the array (SQLSTATE 2202E). For an associative array, the *array-index* expression must be assignable to the index data type of the associative array (SQLSTATE 428H1) and cannot be the null value.

field-reference
Identifies the field within a row type value that is the assignment target. The *field-reference* must be specified as a qualified *field-name* where the qualifier identifies the row value in which the field is defined.

DESCRIPTOR *result-descriptor-name*
Identifies an output SQLDA that must contain a valid description of host variables.

Before the EXECUTE statement is processed, the user must set the following fields in the input SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA
- SQLD to indicate the number of variables used in the SQLDA when processing the statement
- SQLVAR occurrences to indicate the attributes of the variables.

The SQLDA must have enough storage to contain all SQLVAR occurrences. Therefore, the value in SQLDABC must be greater than or equal to 16 + SQLN*(N), where N is the length of an SQLVAR occurrence.

If LOB or structured data type output data must be accommodated, there must be two SQLVAR entries for every output parameter marker.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.

USING
Introduces a list of variables or expressions for which values are substituted for the input parameter markers in the prepared statement.

For a dynamic CALL statement, parameter markers appearing in IN and INOUT arguments to the procedure are input parameter markers. For all other dynamic statements, all the parameter markers are input parameter markers. If any input parameter markers appear in the statement, the USING clause must be specified (SQLSTATE 07004).

***input-host-variable*, ...**
Identifies a host variable that is declared in the program in accordance with the rules for declaring host variables. The number of variables must be the same as the number of input parameter markers in the prepared statement. The *n*th variable corresponds to the *n*th parameter marker in the prepared statement. Locator variables and file reference variables, where appropriate, can be provided as the source of values for parameter markers.

expression
Identifies an expression to be used as the input for the corresponding input parameter marker in the prepared statement. An expression other than a *host-variable* can only be specified when the EXECUTE statement is issued within a compound SQL (compiled) statement.

DESCRIPTOR *input-descriptor-name*
Identifies an input SQLDA that must contain a valid description of host variables.

Before the EXECUTE statement is processed, the user must set the following fields in the input SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA

- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA
- SQLD to indicate the number of variables used in the SQLDA when processing the statement
- SQLVAR occurrences to indicate the attributes of the variables.

The SQLDA must have enough storage to contain all SQLVAR occurrences. Therefore, the value in SQLDABC must be greater than or equal to 16 + SQLN*(N), where N is the length of an SQLVAR occurrence.

If LOB or structured data type input data must be accommodated, there must be two SQLVAR entries for every parameter marker.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.

FOR host-variable or integer-constant ROWS

Specifies the number of rows of source data. The values for the insert or merge operation are specified in the USING clause.

host-variable or integer-constant is assigned to an integral value k. If host-variable is specified, it must be of type integer or short and must not include an indicator variable. k must be in the range 2 to 32767. If FOR host-variable or integer-constant ROWS is not provided, the SQL will be executed with array of size 1.

Notes

- Before the prepared statement is executed, each input parameter marker is effectively replaced by the value of its corresponding variable or expression. For a typed parameter marker, the attributes of the target variable or expression are those specified by the CAST specification. For an untyped parameter marker, the attributes of the target variable or expression are determined according to the context of the parameter marker.

  Let V denote an input variable or expression that corresponds to parameter marker P. The value of V is assigned to the target variable for P in accordance with the rules for assigning a value to a column. Thus:

  – V must be compatible with the target.
  – If V is a string, its length must not be greater than the length attribute of the target.
  – If V is a number, the absolute value of its integral part must not be greater than the maximum absolute value of the integral part of the target.
  – If the attributes of V are not identical to the attributes of the target, the value is converted to conform to the attributes of the target.

  When the prepared statement is executed, the value used in place of P is the value of the target variable for P or the result of the target expression for P. For example, if V is CHAR(6) and the target is CHAR(8), the value used in place of P is the value of V padded with two blanks.
- For a dynamic CALL statement, after the prepared statement is executed, the returned value of each OUT and INOUT argument is assigned to the assignment target corresponding to the output parameter marker used for the argument. For a typed parameter marker, the attributes of the target variable are those specified by the CAST specification. For an untyped parameter marker, the attributes of the target variable are those specified by the definition of the parameter of the procedure.

  Let V denote an output assignment target that corresponds to parameter marker P, which is used for argument A of a procedure. The value of A is assigned to V in accordance with the rules for retrieving a value from a column. Thus:

  – V must be compatible with A.
  – If V is a string, its length must not be less than the length of A, or the value of A will be truncated.
  – If V is a number, the maximum absolute value of its integral part must not be less than the absolute value of the integral part of A.
  – If the attributes of V are not identical to the attributes of A, the value of A is converted to conform to the attributes of V.

- ***Dynamic SQL statement caching:*** The information required to execute dynamic and static SQL statements is placed in the database package cache when static SQL statements are first referenced or when dynamic SQL statements are first prepared. This information stays in the package cache until it becomes invalid, the cache space is required for another statement, or the database is shut down.

  When an SQL statement is executed or prepared, the package information relevant to the application issuing the request is loaded from the system catalog into the package cache. The actual executable section for the individual SQL statement is also placed into the cache: static SQL sections are read in from the system catalog and placed in the package cache when the statement is first referenced; dynamic SQL sections are placed directly in the cache after they have been created. Dynamic SQL sections can be created by an explicit statement, such as PREPARE or EXECUTE IMMEDIATE. Once created, sections for dynamic SQL statements may be recreated by an implicit prepare of the statement by the system if the original section has been deleted for space management reasons, or has become invalid due to changes in the environment.

  Each SQL statement is cached at the database level and can be shared among applications. Static SQL statements are shared among applications using the same package; dynamic SQL statements are shared among applications using the same compilation environment, and the exact same statement text. The text of each SQL statement issued by an application is cached locally within the application for use if an implicit prepare is required. Each PREPARE statement in the application program can cache one statement. All EXECUTE IMMEDIATE statements in an application program share the same space, and only one cached statement exists for all these EXECUTE IMMEDIATE statements at a time. If the same PREPARE or any EXECUTE IMMEDIATE statement is issued multiple times with a different SQL statement each time, only the last statement will be cached for reuse. The optimal use of the cache is to issue a number of different PREPARE statements once at the start of the application, and then to issue an EXECUTE or OPEN statement as required.

  When dynamic SQL statements are cached, a statement can be reused over multiple units of work without needing to prepare the statement again, unless the SQL statements prepared in a package are bound with the `KEEPDYNAMIC NO` option. The system recompiles the statement if necessary when environment changes occur.

  The following events are examples of environment or data object changes that can cause cached dynamic statements to be implicitly prepared on the next PREPARE, EXECUTE, EXECUTE IMMEDIATE, or OPEN request:

  - ALTER FUNCTION
  - ALTER METHOD
  - ALTER NICKNAME
  - ALTER PROCEDURE
  - ALTER SERVER
  - ALTER TABLE
  - ALTER TABLESPACE
  - ALTER TYPE
  - CREATE FUNCTION
  - CREATE FUNCTION MAPPING
  - CREATE INDEX
  - CREATE METHOD
  - CREATE PROCEDURE
  - CREATE TABLE
  - CREATE TEMPORARY TABLESPACE
  - CREATE TRIGGER
  - CREATE TYPE
  - DROP (all objects)

- RUNSTATS on any table or index
- Any action that causes a view to become inoperative
- UPDATE of statistics in any system catalog table
- SET CURRENT DEGREE
- SET PATH
- SET QUERY OPTIMIZATION
- SET SCHEMA
- SET SERVER OPTION

The following list outlines the behavior that can be expected from cached dynamic SQL statements:

- *PREPARE Requests:* Subsequent preparations of the same statement do not incur the cost of compiling the statement if the section is still valid. The cost and cardinality estimates for the current cached section are returned. These values might differ from the values returned from any previous PREPARE for the same SQL statement. You do not need to issue a PREPARE statement subsequent to a COMMIT or ROLLBACK statement, unless the statement is associated with a package that was bound with `KEEPDYNAMIC NO`.
- *EXECUTE Requests:* EXECUTE statements may occasionally incur the cost of implicitly preparing the statement if it has become invalid since the original PREPARE. If a section is implicitly prepared, it will use the current environment and not the environment of the original PREPARE statement.
- *EXECUTE IMMEDIATE Requests:* Subsequent EXECUTE IMMEDIATE statements for the same statement will not incur the cost of compiling the statement if the section is still valid.
- *OPEN Requests:* OPEN requests for dynamically defined cursors may occasionally incur the cost of implicitly preparing the statement if it has become invalid since the original PREPARE statement. If a section is implicitly prepared, it will use the current environment and not the environment of the original PREPARE statement.
- *FETCH Requests:* No behavior changes should be expected.
- *ROLLBACK:* Only those dynamic SQL statements prepared or implicitly prepared during the unit of work affected by the rollback operation are invalidated. Inactive dynamic SQL statements associated with a package bound with `KEEPDYNAMIC NO` are removed from the application SQL context after a ROLLBACK opeation and must be explicitly prepared again before the application can execute them. Dynamic SQL statements are still cached at the database level, so a subsequent PREPARE request does not incur the cost of compiling the statement if the section is still valid.
- *COMMIT:* Dynamic SQL statements are not be invalidated, but any acquired locks are be freed. Cursors not defined with the `WITH HOLD` option are closed and their locks freed. Open cusors defined with the `WITH HOLD` option hold onto their package and section locks to protect the active section both during and after commit processing. Dynamic SQL statements bound with the `KEEPDYNAMIC NO` option are not in a prepared state after a transaction boundary and must be explicitly prepared again before the application can execute them. SELECT statements prepared for an open cursor defined with the `WITH HOLD` option remain in a prepared state until a transaction boundary is hit where the cursor is closed. Inactive dynamic SQL statements associated with a package bound with `KEEPDYNAMIC NO` are removed from the application SQL context after a commit operation and must be explicitly prepared again before the application can execute them.

If an error occurs during an implicit prepare, an error will be returned for the request causing the implicit prepare (SQLSTATE 56098).

- For Embedded SQL applications, the `db2dsdriver.cfg` keyword **Anonyblksqlexec** and Db2 registry variable **DB2_ANONYMOUS_ESQL_EXECUTION_BLOCK** allow you to use INPUT and OUTPUT parameters in both the USING and INTO clauses, to get correct values in output.

  For example, when the variable is set to TRUE or 1 (or the keyword set to 1), you can specify INPUT and OUTPUT parameters in both the USING and INTO clauses:

```
EXEC SQL EXECUTE db2strm1 INTO
:h_name_in INDICATOR:h_name_in_ind,
```

```
:h_name_out INDICATOR:h_name_out_ind USING
:h_name_in, :h_name_out;
```

If the **DB2_ANONYMOUS_ESQL_EXECUTION_BLOCK** variable is not set, or set to FALSE or 0 (or the keyword set to 0), you can specify only INPUT parameters in the USING clause and only OUTPUT parameters in the INTO clause.

Attention: The **Anonyblksqlexec** keyword and **DB2_ANONYMOUS_ESQL_EXECUTION_BLOCK** registry variable are available in Db2 11.5.6 and later versions.

Examples

Example 1: In this C example, an INSERT statement with parameter markers is prepared and executed. Host variables h1 - h4 correspond to the format of TDEPT.

```
strcpy (s,"INSERT INTO TDEPT VALUES(?,?,?,?)");
EXEC SQL PREPARE DEPT_INSERT FROM :s;
  .
  .
(Check for successful execution and put values into :h1, :h2, :h3, :h4)
  .
  .
EXEC SQL EXECUTE DEPT_INSERT USING :h1, :h2,
:h3, :h4;
```

Example 2: This EXECUTE statement uses an SQLDA.

```
EXECUTE S3 USING DESCRIPTOR :sqlda3
```

Example 3: Given a procedure to award an employee a bonus:

```
CREATE PROCEDURE GIVE_BONUS (IN EMPNO INTEGER,
                             IN DEPTNO INTEGER,
                             OUT CHECK INTEGER,
                             INOUT BONUS DEC(6,0))
...
```

Dynamically call the procedure from a C application. The procedure takes the following host variables as input:

- *employee*, the ID number of the employee
- *dept*, the department number
- *bonus*, the bonus to be awarded to the employee

The procedure returns the following values to the host variables:

- *check_no*, the ID number from the check
- *bonus*, the actual bonus amount (after any adjustments)

```
strcpy (s, "CALL GIVE_BONUS(?, ?, ?, ?)");
EXEC SQL PREPARE DO_BONUS FROM :s;
  .
  .
/* Check for successful execution and put values into
   :employee, :dept, and :bonus */
  .
  .
EXEC SQL EXECUTE DO_BONUS INTO :check_no, :bonus
                          USING :employee, :dept, :bonus;
  .
  .
/* Check for successful execution and process the
   values returned in :check_no and :bonus */
```

EXECUTE IMMEDIATE

The EXECUTE IMMEDIATE statement prepares an executable form of an SQL statement from a character string form of the statement, and executes the SQL statement.

EXECUTE IMMEDIATE combines the basic functions of the PREPARE and EXECUTE statements. It can be used to prepare and execute SQL statements that contain neither host variables nor parameter markers.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

The authorization rules are those defined for the specified SQL statement.

The authorization ID of the statement might be affected by the DYNAMICRULES bind option.

Syntax

Description

expression
An expression returning the statement string to be executed. The expression must return a character-string type that is less than the maximum statement size of 2 097 152 bytes. Note that a CLOB(2097152) can contain a maximum size statement, but a VARCHAR cannot.

The statement string must be one of the following SQL statements:

- ALTER
- CALL
- COMMENT
- COMMIT
- Compound SQL (compiled)
- Compound SQL (inlined)
- CREATE
- DECLARE GLOBAL TEMPORARY TABLE
- DELETE
- DROP
- EXPLAIN
- FLUSH EVENT MONITOR
- FLUSH PACKAGE CACHE
- GRANT
- INSERT
- LOCK TABLE
- MERGE
- REFRESH TABLE
- RELEASE SAVEPOINT
- RENAME
- REVOKE

- ROLLBACK
- SAVEPOINT
- SET COMPILATION ENVIRONMENT
- SET CURRENT DECFLOAT ROUNDING MODE
- SET CURRENT DEFAULT TRANSFORM GROUP
- SET CURRENT DEGREE
- SET CURRENT EXPLAIN MODE
- SET CURRENT EXPLAIN SNAPSHOT
- SET CURRENT FEDERATED ASYNCHRONY
- SET CURRENT IMPLICIT XMLPARSE OPTION
- SET CURRENT ISOLATION
- SET CURRENT LOCALE LC_MESSAGES
- SET CURRENT LOCALE LC_TIME
- SET CURRENT LOCK TIMEOUT
- SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
- SET CURRENT MDC ROLLOUT MODE
- SET CURRENT OPTIMIZATION PROFILE
- SET CURRENT QUERY OPTIMIZATION
- SET CURRENT REFRESH AGE
- SET CURRENT TEMPORAL BUSINESS_TIME
- SET CURRENT TEMPORAL SYSTEM_TIME
- SET ENCRYPTION PASSWORD
- SET EVENT MONITOR STATE (only if DYNAMICRULES run behavior is in effect for the package)
- SET INTEGRITY
- SET PASSTHRU
- SET PATH
- SET ROLE (only if DYNAMICRULES run behavior is in effect for the package)
- SET SCHEMA
- SET SERVER OPTION
- SET SESSION AUTHORIZATION
- SET SQL_CCFLAGS
- SET USAGE LIST STATE (only if DYNAMICRULES run behavior is in effect for the package)
- SET variable
- TRANSFER OWNERSHIP (only if DYNAMICRULES run behavior is in effect for the package)
- TRUNCATE (only if DYNAMICRULES run behavior is in effect for the package)
- UPDATE

The statement string must not include parameter markers or references to host variables, and must not begin with EXEC SQL. It must not contain a statement terminator, with the exception of compound SQL statements which can contain semi-colons (;) to separate statements within the compound block. A compound SQL statement is used within some CREATE and ALTER statements which, therefore, can also contain semi-colons.

When an EXECUTE IMMEDIATE statement is executed, the specified statement string is parsed and checked for errors. If the SQL statement is invalid, it is not executed, and the error condition that prevents its execution is reported in the SQLCA. If the SQL statement is valid, but an error occurs during its execution, that error condition is reported in the SQLCA.

Notes

- Statement caching affects the behavior of an EXECUTE IMMEDIATE statement.

Example

Use C program statements to move an SQL statement to the host variable *qstring* (char[80]), and prepare and execute whatever SQL statement is in the host variable qstring.

```
if ( strcmp(accounts,"BIG") == 0 )
  strcpy (qstring,"INSERT INTO WORK_TABLE SELECT *
    FROM EMP_ACT WHERE ACTNO < 100");
else
  strcpy (qstring,"INSERT INTO WORK_TABLE SELECT *
    FROM EMP_ACT WHERE ACTNO >= 100");
  .
  .
  .
EXEC SQL  EXECUTE IMMEDIATE :qstring;
```

EXPLAIN

The EXPLAIN statement captures information about the access plan chosen for the supplied explainable statement and places this information into the explain tables.

An *explainable statement* can either be a valid XQuery statement or one of the following SQL statements: CALL, Compound SQL (Dynamic), DELETE, INSERT, MERGE, REFRESH, SELECT, SELECT INTO, SET INTEGRITY, UPDATE, VALUES, or VALUES INTO.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

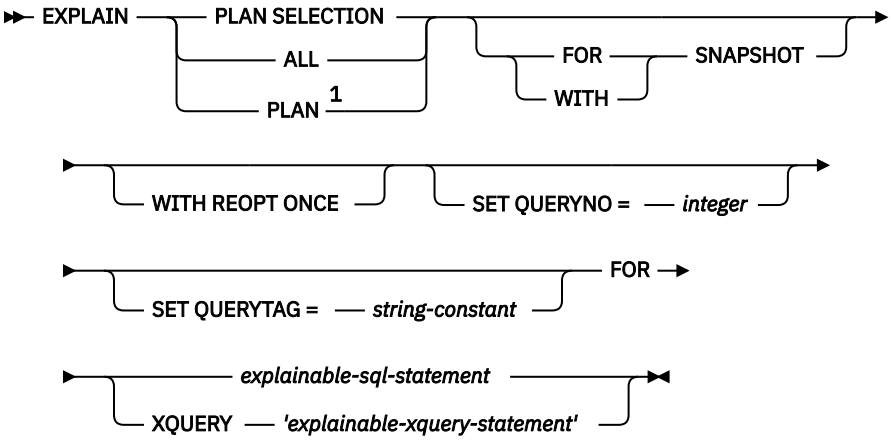
The statement to be explained is not executed.

Authorization

The authorization ID of the statement must hold at least one of the following authorizations:

- DATAACCESS authority which allows an INSERT, UPDATE, DELETE, or SELECT statement.
- DATAACCESS authority on the schema containing the explain tables
- INSERT privilege on the explain tables and at least one of the following authorizations:
  - All the privileges that are necessary to execute the explainable statement that is specified in the EXPLAIN statement (for example, if a DELETE statement is used as the explainable statement, the authorization rules for the DELETE statement are applied when the DELETE statement is explained)
  - EXPLAIN authority
  - SQLADM authority
  - DBADM authority

Syntax



Notes:

[1] The PLAN option is supported only for syntax toleration of existing Db2 for z/OS EXPLAIN statements. There is no PLAN table. Specifying PLAN is equivalent to specifying PLAN SELECTION.

Description

PLAN SELECTION
Indicates that the information from the plan selection phase of query compilation is to be inserted into the explain tables.

ALL
Specifying ALL is equivalent to specifying PLAN SELECTION.

PLAN
The PLAN option provides syntax toleration for existing database applications from other systems. Specifying PLAN is equivalent to specifying PLAN SELECTION.

FOR SNAPSHOT
This clause indicates that only an explain snapshot is to be taken and placed into the SNAPSHOT column of the EXPLAIN_STATEMENT table. No other explain information is captured other than that present in the EXPLAIN_INSTANCE and EXPLAIN_STATEMENT tables.

WITH SNAPSHOT
This clause indicates that, in addition to the regular explain information, an explain snapshot is to be taken.

The default behavior of the EXPLAIN statement is to only gather regular explain information and not the explain snapshot.

default (neither FOR SNAPSHOT nor WITH SNAPSHOT specified)
Puts explain information into the explain tables.

WITH REOPT ONCE
This clause indicates that the specified *explainable statement* is to be reoptimized using the values for host variables, parameter markers, special registers, or global variables that were previously used to reoptimize this statement with REOPT ONCE. The explain tables will be populated with the new access plan. If the user has DBADM authority, or the database registry variable DB2_VIEW_REOPT_VALUES is set to YES, the EXPLAIN_PREDICATE table will also be populated with the values if they are used to reoptimize the statement.

SET QUERYNO = *integer*
Associates *integer*, via the QUERYNO column in the EXPLAIN_STATEMENT table, with the *explainable statement*. The integer value supplied must be a positive value.

If this clause is not specified for a dynamic EXPLAIN statement, a default value of one (1) is assigned. For a static EXPLAIN statement, the default value assigned is the statement number assigned by the precompiler.

SET QUERYTAG = *string-constant*

Associates *string-constant*, via the QUERYTAG column in the EXPLAIN_STATEMENT table, with the *explainable statement*. *string-constant* can be any character string up to 20 bytes in length. If the value supplied is less than 20 bytes in length, the value is padded on the right with blanks to the required length.

If this clause is not specified for an EXPLAIN statement, blanks are used as the default value.

FOR *explainable-sql-statement*

Specifies the SQL statement to be explained. This statement can be any valid CALL, Compound SQL (Dynamic), DELETE, INSERT, MERGE, REFRESH, SELECT, SELECT INTO, SET INTEGRITY, UPDATE, VALUES, or VALUES INTO SQL statement. If the EXPLAIN statement is embedded in a program, the *explainable-sql-statement* can contain references to host variables (these variables must be defined in the program). Similarly, if EXPLAIN is being dynamically prepared, the *explainable-sql-statement* can contain parameter markers.

The *explainable-sql-statement* must be a valid SQL statement that could be prepared and executed independently of the EXPLAIN statement. It cannot be a statement name or host variable. SQL statements referring to cursors defined through CLP are not valid for use with this statement.

To explain dynamic SQL within an application, the entire EXPLAIN statement must be dynamically prepared.

FOR XQUERY *'explainable-xquery-statement'*

Specifies the XQUERY statement to be explained. This statement can be any valid XQUERY statement.

If the EXPLAIN statement is embedded in a program, the *'explainable-xquery-statement'* can contain references to host variables, provided that the host variables are not used in the top level XQUERY statement, but are passed in through an XMLQUERY function, by an XMLEXISTS predicate, or by an XMLTABLE function. The host variables must be defined in the program.

Similarly, if EXPLAIN is being dynamically prepared, the *'explainable-xquery-statement'* can contain parameter markers, provided that the same restrictions as for passing host variables are followed.

Alternatively, the Db2 XQUERY function db2-fn:sqlquery can be used to embed SQL statements with references to host variables and parameter markers.

The *'explainable-xquery-statement'* must be a valid XQUERY statement that could be prepared and executed independently of the EXPLAIN statement. Query statements referring to cursors defined through CLP are not valid for use with this statement.

Notes

- The Explain facility uses the following IDs as the schema when qualifying explain tables that it is populating:
  - The session authorization ID for dynamic SQL
  - The statement authorization ID for static SQL

  The schema can be associated with a set of explain tables, or aliases that point to a set of explain tables under a different schema. If no explain tables are found under the schema, the Explain facility checks for explain tables under the SYSTOOLS schema and attempts to use those tables.
- The following table shows the interaction of the snapshot keywords and the explain information.

Keyword Specified	Capture Explain Information?
none	Yes
FOR SNAPSHOT	No
WITH SNAPSHOT	Yes

If neither the FOR SNAPSHOT nor the WITH SNAPSHOT clause is specified, an explain snapshot is not taken.

- The explain tables must be created by the user before invocation of the EXPLAIN statement. The information generated by this statement is stored in the explain tables, in the schema that is designated at the time the statement is compiled.
- If any errors occur during the compilation of the *explainable statement* supplied, then no information is stored in the explain tables.
- The access plan generated for the *explainable statement* is not saved and thus, cannot be invoked at a later time. The explain information for the *explainable statement* is inserted when the EXPLAIN statement itself is compiled.
- For a static EXPLAIN query statement, the information is inserted into the explain tables at bind time and during an explicit rebind. During precompilation, the static EXPLAIN statements are commented out in the modified application source file. At bind time, the EXPLAIN statements are stored in the SYSCAT.STATEMENTS catalog. When the package is run, the EXPLAIN statement is not executed. Note that the section numbers for all statements in the application will be sequential and will include the EXPLAIN statements. An alternative to using a static EXPLAIN statement is to use a combination of the EXPLAIN and EXPLSNAP BIND or PREP options. Static EXPLAIN statements can be used to cause the explain tables to be populated for one specific static query statement out of many; simply prefix the target statement with the appropriate EXPLAIN statement syntax and bind the application without using either of the explain BIND or PREP options. The EXPLAIN statement can also be used when it is advantageous to set the QUERYNO or QUERYTAG field at the time of the actual explain invocation.
- Static EXPLAIN statements in an SQL procedure are evaluated when the procedure is compiled.
- For an incremental bind EXPLAIN query statement, the explain tables are populated when the EXPLAIN statement is submitted for compilation. When the package is run, the EXPLAIN statement performs no processing (though the statement will be successful). When populating the explain tables, the explain table qualifier and authorization ID used during population will be those of the package owner. The EXPLAIN statement can also be used when it is advantageous to set the QUERYNO or QUERYTAG field at the time of the actual explain invocation.
- For dynamic EXPLAIN statements, the explain tables are populated at the time the EXPLAIN statement is submitted for compilation. An EXPLAIN statement can be prepared with the PREPARE statement but, if executed, will perform no processing (though the statement will be successful). An alternative to issuing dynamic EXPLAIN statements is to use a combination of the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special registers to explain dynamic query statements. The EXPLAIN statement should be used when it is advantageous to set the QUERYNO or QUERYTAG field at the time of the actual EXPLAIN invocation.
- If the REOPT bind option is set to ONCE, and either the CURRENT EXPLAIN MODE or the CURRENT EXPLAIN SNAPSHOT special register is set to REOPT, the execution of static and dynamic query statements containing host variables, special registers, parameter markers, or global variables will cause explain information to be captured for the statement only when the statement is reoptimized. Alternatively, if the REOPT bind option is set to ALWAYS, explain information will be captured every time these statements are executed.

Examples

- *Example 1:* Explain a simple SELECT statement and tag with QUERYNO = 13.

```
EXPLAIN PLAN SET QUERYNO = 13
  FOR SELECT C1
  FROM T1
```

- *Example 2:* Explain a simple SELECT statement and tag with QUERYTAG = 'TEST13'.

```
EXPLAIN PLAN SELECTION SET QUERYTAG = 'TEST13'
  FOR SELECT C1
  FROM T1
```

- *Example 3:* Explain a simple SELECT statement and tag with QUERYNO = 13 and QUERYTAG = 'TEST13'.

```
EXPLAIN PLAN SELECTION SET QUERYNO = 13 SET QUERYTAG = 'TEST13'
  FOR SELECT C1
  FROM T1
```

- *Example 4:* Attempt to get explain information when explain tables do not exist.

```
EXPLAIN ALL FOR SELECT C1
  FROM T1
```

  This statement will fail because the explain tables have not been defined (SQLSTATE 42704).

- *Example 5:* The following statement will succeed if it is found in the package cache and has already been compiled using REOPT ONCE.

```
EXPLAIN ALL WITH REOPT ONCE FOR SELECT C1
  FROM T1
  WHERE C1 = :<host variable>
```

- *Example 6:* The following example uses the db2-fn:xmlcolumn function, which takes the case- sensitive name of an XML column as an argument and returns an XML sequence that is the concatenation of XML column values.

  Consider a table called BUSINESS.CUSTOMER with an XML column called INFO. A simple XQuery that returns all documents from the INFO column is :

```
EXPLAIN PLAN SELECTION
   FOR XQUERY 'db2-fn:xmlcolumn ("BUSINESS.CUSTOMER.INFO")'
```

  If a column value is null, then the resulting return sequence for that row will be empty.

FETCH

The FETCH statement positions a cursor on the next row of its result table and assigns the values of that row to target variables.

Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared. When invoked using the command line processor, the syntax following *cursor-name* is optional and different from the SQL syntax.

For more information, refer to "Using command line SQL statements and XQuery statements" in *Command Reference*.

Authorization

For each global variable used as a *cursor-variable-name* or in the expression for an *array-index*, the privileges held by the authorization ID of the statement must include one of the following:

- READ privilege on the global variable that is not defined in a module
- EXECUTE privilege on the module of the global variable that is defined in a module
- EXECUTEIN privilege on the schema containing the module of the global variable that is defined in a module
- DATAACCESS authority on the schema containing the module of the global variable that is defined in a module

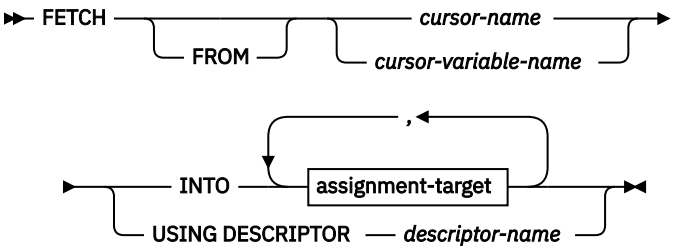
For each global variable used as an *assignment-target*, the privileges held by the authorization ID of the statement must include one of the following:

- WRITE privilege on the global variable that is not defined in a module

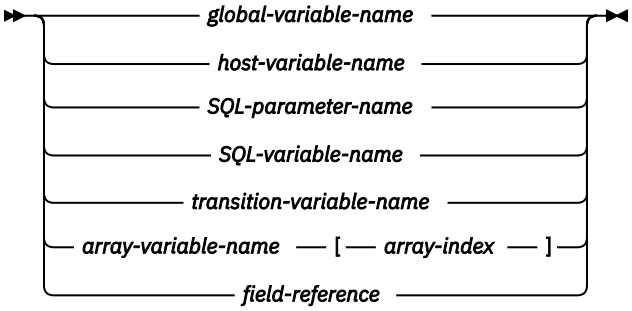
- EXECUTE privilege on the module of the global variable that is defined in a module
- EXECUTEIN privilege on the schema containing the module of the global variable that is defined in a module
- DATAACCESS authority on the schema containing the module of the global variable that is defined in a module

For the authorization required to use a cursor, see "DECLARE CURSOR".

Syntax



assignment-target



Description

cursor-variable-name

Identifies the cursor to be used in the fetch operation. The cursor-variable-name must identify a cursor variable that is in scope. When the FETCH statement is executed, the underlying cursor of the *cursor-variable-name* must be in the open state. A FETCH statement using a *cursor-variable-name* can only be used within a compound SQL (compiled) statement.

INTO *assignment-target*

Identifies one or more targets for the assignment of output values. The first value in the result row is assigned to the first target in the list, the second value to the second target, and so on. Each assignment to an *assignment-target* is made in sequence through the list. If an error occurs on any assignment, the value is not assigned to the target, and no more values are assigned to targets. Any values that have already been assigned to targets remain assigned.

When the data type of every *assignment-target* is not a row type, then the value 'W' is assigned to the SQLWARN3 field of the SQLCA if the number of *assignment-targets* is less than the number of result column values.

If the data type of an *assignment-target* is a row type, then there must be exactly one *assignment-target* specified (SQLSTATE 428HR), the number of columns must match the number of fields in the row type, and the data types of the columns of the fetched row must be assignable to the corresponding fields of the row type (SQLSTATE 42821).

If the data type of an *assignment-target* is an array element, then there must be exactly one *assignment-target* specified.

global-variable-name
Identifies the global variable that is the assignment target.

host-variable-name
Identifies the host variable that is the assignment target. For LOB output values, the target can be a regular host variable (if it is large enough), a LOB locator variable, or a LOB file reference variable.

SQL-parameter-name
Identifies the parameter that is the assignment target.

SQL-variable-name
Identifies the SQL variable that is the assignment target. SQL variables must be declared before they are used.

transition-variable-name
Identifies the column to be updated in the transition row. A *transition-variable-name* must identify a column in the subject table of a trigger, optionally qualified by a correlation name that identifies the new value.

array-variable-name
Identifies an SQL variable, SQL parameter, or global variable of an array type.

[array-index]
An expression that specifies which element in the array will be the target of the assignment. For an ordinary array, the *array-index* expression must be assignable to INTEGER (SQLSTATE 428H1) and cannot be the null value. Its value must be between 1 and the maximum cardinality defined for the array (SQLSTATE 2202E). For an associative array, the *array-index* expression must be assignable to the index data type of the associative array (SQLSTATE 428H1) and cannot be the null value.

field-reference
Identifies the field within a row type value that is the assignment target. The *field-reference* must be specified as a qualified *field-name* where the qualifier identifies the row value in which the field is defined.

USING DESCRIPTOR *descriptor-name*
Identifies an SQLDA that must contain a valid description of zero or more host variables.

Before the FETCH statement is processed, the user must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA.
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA.
- SQLD to indicate the number of variables used in the SQLDA when processing the statement.
- SQLVAR occurrences to indicate the attributes of the variables.

The SQLDA must have enough storage to contain all SQLVAR occurrences. Therefore, the value in SQLDABC must be greater than or equal to 16 + SQLN*(N), where N is the length of an SQLVAR occurrence.

If LOB or structured type result columns need to be accommodated, there must be two SQLVAR entries for every select-list item (or column of the result table).

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.

The *n*th variable described in the SQLDA corresponds to the *n*th column of the result table of the cursor. The data type of each variable must be compatible with its corresponding column.

Each assignment to a variable is made according to specific rules. If the number of variables is less than the number of values in the row, the SQLWARN3 field of the SQLDA is set to 'W'. Note that there is no warning if there are more variables than the number of result columns. If an assignment error occurs, the value is not assigned to the variable, and no more values are assigned to variables. Any values that have already been assigned to variables remain assigned.

Notes

- ***Cursor position:*** An open cursor has three possible positions:
  - Before a row
  - On a row
  - After the last row.

  A cursor can only be on a row as a result of a FETCH statement. If the cursor is currently positioned on or after the last row of the result table:
  - SQLCODE is set to +100, and SQLSTATE is set to '02000'.
  - The cursor is positioned after the last row.
  - Values are not assigned to assignment targets.

  If the cursor is currently positioned before a row, it will be repositioned on that row, and values will be assigned to targets as specified by the INTO or USING clause.

  If the cursor is currently positioned on a row other than the last row, it will be repositioned on the next row and values of that row will be assigned to targets as specified by the INTO or USING clause.

  If a cursor is on a row, that row is called the current row of the cursor. A cursor referenced in an UPDATE or DELETE statement must be positioned on a row.

  It is possible for an error to occur that makes the state of the cursor unpredictable.
- When retrieving into LOB locators in situations where it is not necessary to retain the locator across FETCH statements, it is good practice to issue a FREE LOCATOR statement before issuing the next FETCH statement, as locator resources are limited.
- It is possible that a warning may not be returned on a FETCH. It is also possible that the returned warning applies to a previously fetched row. This occurs as a result of optimizations such as the use of system temporary tables or pushdown operators.
- Statement caching affects the behavior of an EXECUTE IMMEDIATE statement.
- Db2 CLI supports additional fetching capabilities. For instance, when a cursor's result table is read-only, the SQLFetchScroll() function can be used to position the cursor at any spot within that result table.
- For an updatable cursor, a lock is obtained on a row when it is fetched.
- If the cursor definition contains an SQL data change statement or invokes a routine that modifies SQL data, an error during the fetch operation does not cause the modified rows to be rolled back, even if the error results in the cursor being closed.

Examples

- *Example 1:* In this C example, the FETCH statement fetches the results of the SELECT statement into the program variables dnum, dname, and mnum. When no more rows remain to be fetched, the not found condition is returned.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO FROM TDEPT
    WHERE ADMRDEPT = 'A00';

EXEC SQL OPEN C1;

while (SQLCODE==0) {
    EXEC SQL FETCH C1 INTO :dnum, :dname, :mnum;
}

EXEC SQL CLOSE C1;
```

- *Example 2:* This FETCH statement uses an SQLDA.

```
FETCH CURS USING DESCRIPTOR :sqlda3
```

FLUSH BUFFERPOOLS

The FLUSH BUFFERPOOLS statement writes the dirty pages from all the local buffer pools for a particular database synchronously to disk.

In Db2 pureScale environments, the dirty pages in the group buffer pool are also written synchronously to disk.

This statement is not under transaction control.

The FLUSH BUFFERPOOLS statement can be used in the following ways:

- To reduce the recovery window of a database in the event of a failure
- To reduce the size of logs written to a backup image before database operations such as online backups
- To minimize the recovery time of a split-mirror database

Invocation

The statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include SQLADM, DBADM, SYSMAINT, SYSCTRL, or SYSADM authority.

Syntax

Description

ALL
Flushes the dirty pages from all the buffer pools (local and group).

Notes

- **Dirty pages processing:** Only the dirty pages that are in the buffer pools when the statement begins processing are written to disk. Any dirty pages that are added to the buffer pools before the statement finishes processing are not written to disk.
- **Syntax alternatives:** BUFFERPOOL can be specified in place of BUFFERPOOLS.

FLUSH EVENT MONITOR

The FLUSH EVENT MONITOR statement writes current database monitor values for all active monitor types associated with event monitor *event-monitor-name* to the event monitor I/O target.

A partial event record is available at any time for event monitors that have low record generation frequency (such as a database event monitor). Such records are noted in the event monitor log with a *partial record* identifier.

When an event monitor is flushed, its active internal buffers are written to the event monitor output object.

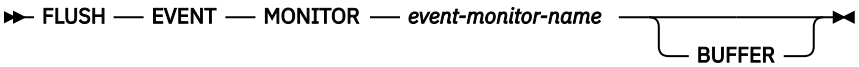
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include SQLADM or DBADM authority.

Syntax



Description

event-monitor-name
Name of the event monitor. This is a one-part name. It is an ordinary identifier.

BUFFER
Indicates that the event monitor buffers are to be written out. If BUFFER is specified, then a partial record is not generated. Only the data already present in the event monitor buffers are written out.

Notes

- Flushing out the event monitor will not cause the event monitor values to be reset. This means that the event monitor record that would have been generated if no flush was performed, will still be generated when the normal monitor event is triggered.
- The FLUSH EVENT MONITOR statement does not cause events to be generated and written for the UNIT OF WORK event monitor.

FLUSH FEDERATED CACHE

The FLUSH FEDERATED CACHE statement flushes the federated cache, allowing fresh metadata to be obtained the next time an SQL statement is issued against the remote table or view using a federated three part name.

When an SQL statement is issued against a remote table or view using a federated three part name, if the remote table or view is being referenced for the first time, the metadata and statistics for the remote object are retrieved and stored in a federated cache.

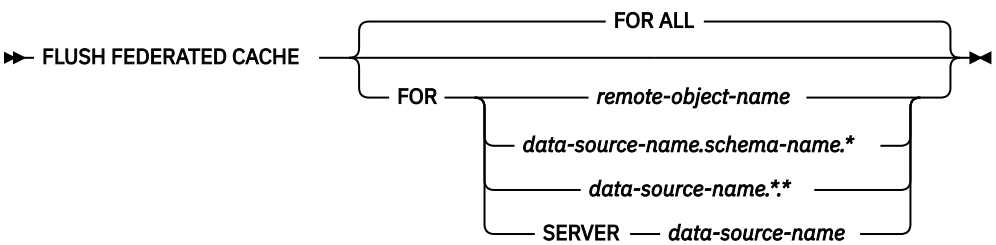
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include either SQLADM or DBADM authority.

Syntax



Description

FOR ALL
Flushes the federated cache information for all objects from all data sources. This is the default.

FOR *remote-object-name*
Flushes the federated cache information for a specific remote table or view.

FOR *data-source-name.schema-name.**
Flushes the federated cache information for all objects in the schema identified by *schema-name* from the specific data source identified by *data-source-name*.

FOR *data-source-name.*.**
Flushes the federated cache information for all objects from the specific data source identified by *data-source-name*.

FOR SERVER *data-source-name*
Flushes the federated cache information for all objects from the specific data source identified by *data-source-name*.

Notes

- **Package invalidation:** Flushing the federated cache causes packages with a dependency on the three-part name to be invalidated. This action could have a performance impact since the invalidated packages need to be recompiled whenever statements from the package are executed.
- **View invalidation:** Flushing the federated cache will not cause the views depending on the three part name to be invalidated. The next time the view is used, it will implicitly revalidate the view. If there are changes to the remote object, it is possible that the statement using the view could return an error.

Examples

- *Example 1*: Flush the federated cache information for the remote-table-name *t1* in the remote-schema-name *rschema* on the data source *rudb*.

```
FLUSH FEDERATED CACHE FOR rudb.rschema.t1
```

- *Example 2*: Flush the federated cache information for all objects in the remote-schema-name *rschema* on the data source *rudb*.

```
FLUSH FEDERATED CACHE FOR rudb.rschema.*
```

- *Example 3*: Flush the federated cache information for all objects from the data source *rudb*.

```
FLUSH FEDERATED CACHE FOR rudb.*.*
```

An alternative to this syntax is as follows:

```
FLUSH FEDERATED CACHE FOR SERVER rudb
```

FLUSH OPTIMIZATION PROFILE CACHE

Multiple statements can be compiled using the same optimization profile.

To make optimization profile processing more efficient, the optimization profile is processed the first time it is used to optimize a statement, and the output is stored in the optimization profile cache. Subsequent references to the optimization profile use the processed version in the optimization profile cache.

An optimization profile should be removed from the optimization profile cache when the version stored in SYSTOOLS.OPT_PROFILE has been updated. When the old version is removed from the cache, the new version will be used upon optimization of subsequent statements that use the optimization profile.

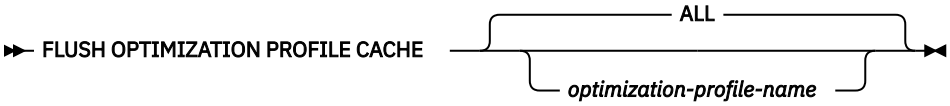
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include either SQLADM or DBADM authority (SQLSTATE 42502).

Syntax



Description

optimization-profile-name
Specifies the name of the optimization profile to be flushed from the optimization profile cache. If the name specified is unqualified, the value of the CURRENT DEFAULT SCHEMA register is used as the implicit qualifier.

ALL
Specifies that all profiles on all active database partitions be flushed from the optimization profile cache.

Notes

- The FLUSH OPTIMIZATION PROFILE CACHE statement removes all or a single optimization profile from the optimization profile cache. It also causes the logical invalidation of any cached dynamic SQL statements that were prepared with that optimization profile.
- New access plans for any invalidated dynamic plans are regenerated when the next request for the same SQL statement is made.
- Packages that reference an optimization profile removed from the optimization profile cache by this statement must be explicitly bound again to allow new access plans to be generated.

Examples

- *Example 1:* The optimization profile "Rick"."Foo" is flushed from the optimization profile cache.

```
SET CURRENT SCHEMA = '"Rick"'
FLUSH OPTIMIZATION PROFILE CACHE "Foo"
```

- *Example 2:* The optimization profile JOHN.ALL is removed from the optimization profile cache.

```
SET CURRENT SCHEMA = '"Rick"'
FLUSH OPTIMIZATION PROFILE CACHE JOHN.ALL
```

Messages

- No errors are issued if the optimization profile cache is empty or if the specified optimization profiles (specified explicitly or implicitly) do not exist in the optimization profile cache.

FLUSH PACKAGE CACHE

The FLUSH PACKAGE CACHE statement invalidates cached dynamic SQL statements in the package cache. This invalidation causes the next request for any SQL statement that matches an invalidated cached dynamic SQL statement to be compiled instead of reused from the package cache.

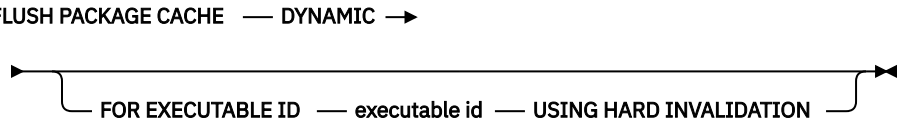
Invocation

This statement can be embedded in an application program or issued by using dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

The privileges that are held by the authorization ID of the statement must include SQLADM or DBADM authority.

Syntax



FOR EXECUTABLE ID executable id
An input argument of type VARCHAR(32) FOR BIT DATA that contains the executable ID used to identify the section to be removed from the package cache. The executable ID cannot represent a static section (4274L). If the executable ID does not map to a currently cached entry, no action will be taken and an error will be returned (4274L).

USING HARD INVALIDATION
Specifies that, if the identified package cache entries to be invalidated are currently being used, the FLUSH PACKAGE CACHE statement will wait until the entry is no longer being used before completing the invalidation.

Notes

- This statement affects all cached dynamic SQL entries in the package cache on all active database partitions.
- As cached dynamic SQL statements are invalidated, the package cache memory that is used for the cached entry is freed if the entry is not in use when the FLUSH PACKAGE CACHE statement runs. Entries are not evicted during the FLUSH PACKAGE CACHE statement if the FOR EXECUTABLE ID clause is used. If the entry still remains in the cache after the statement is executed, the VALID element for that entry will indicate that it is invalid and subsequent requests for that statement will cause a new compilation.
- Any cached dynamic SQL statement currently in use is allowed to continue to exist in the package cache until it is no longer needed by the current user. The next new user of the same statement will force an implicit prepare of the statement, and the new user will run the new version of the cached dynamic SQL statement.

Examples

To mark all cached dynamic SQL entries invalid to force them to be prepared again:

```
FLUSH PACKAGE CACHE DYNAMIC
```

To force a specific cached entry out of the cache, find the target executable ID:

```
select section_type, executable_id, substr(stmt_text, 1, 30) as stmt_text from table
(mon_get_pkg_cache_stmt(null, null, null, -1)) where stmt_text like 'VALUES (17+30)%'

SECTION_TYPE EXECUTABLE_ID
```

```
STMT_TEXT
------------ ------------------------------------------------------------------
------------------------------
D            x'00000001000000000000000000000001000000000002201902201551126783 01' VALUES
(17+30)

  1 record(s) selected.
```

And then issue the command to force it out:

```
flush package cache dynamic for executable id
x'00000001000000000000000000000001000000000002201902201551126783 01' using hard invalidation
DB20000I  The SQL command completed successfully.
```

FLUSH AUTHENTICATION CACHE

The FLUSH AUTHENTICATION CACHE statement flushes the Db2 authentication cache of all entries, allowing fresh entries to be cached. Running this statement will perform the flush on all database members in which the cache is present.
When the Db2 authentication cache is active, each new authentication request made with a username and password is compared against existing, valid entries in the cache. If a match is found, the authentication request is deemed to be successful and processing continues. If a match is not found, the authentication request is processed and, if successful, the result is placed as a new entry in the cache.

Invocation

This statement can be issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include either SECADM or DBADM authority.

Syntax

Description

FOR ALL
Flushes the authentication cache information for all users. This is the default.

Example

- Flush the authentication cache information for all entries with the following command:

```
FLUSH AUTHENTICATION CACHE
```

FOR

The FOR statement executes a statement or group of statements for each row of a table.

Invocation

This statement can be embedded in an:

- SQL procedure definition
- Compound SQL (compiled) statement

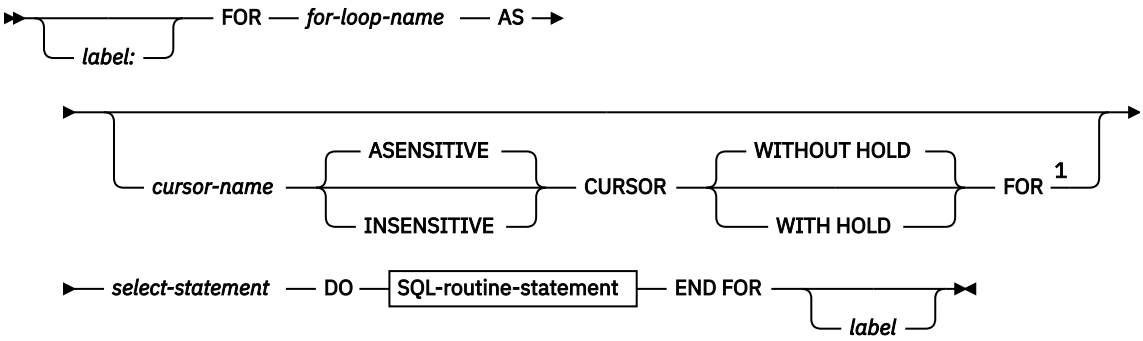
- Compound SQL (inlined) statement

The compound statements can be embedded in an SQL procedure definition, SQL function definition, or SQL trigger definition. It is not an executable statement and cannot be dynamically prepared.

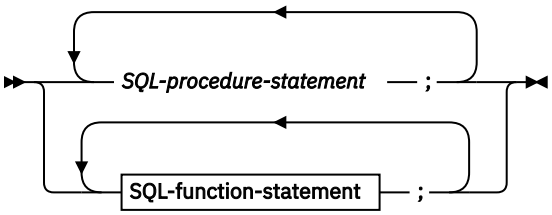
Authorization

No privileges are required to invoke the FOR statement. However, the authorization ID of the statement must hold the necessary privileges to invoke the SQL statements that are embedded in the FOR statement. For the authorization required to use a cursor, see "DECLARE CURSOR".

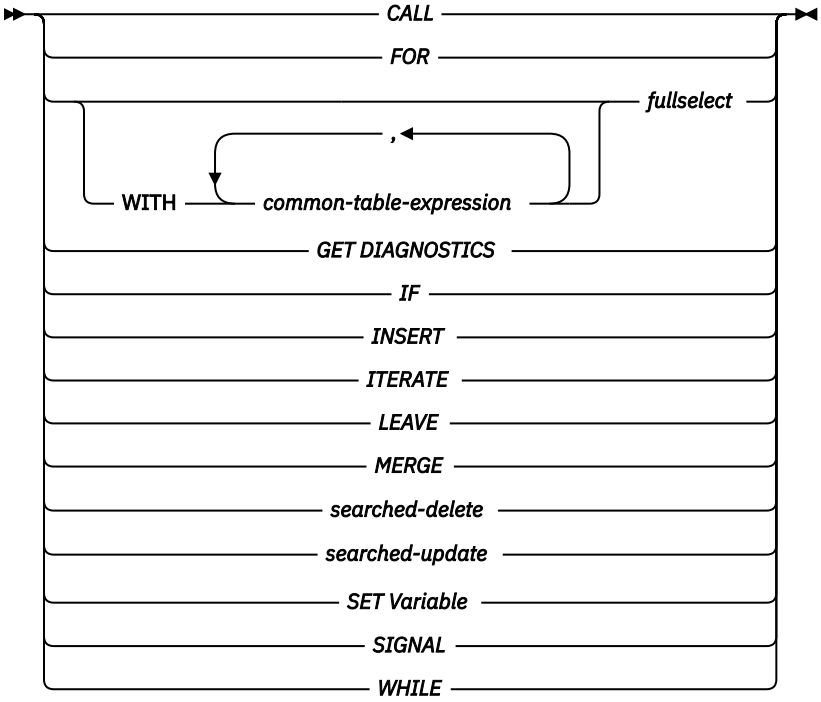
Syntax



SQL-routine-statement



SQL-function-statement



Notes:

[1] This option can only be used in the context of an SQL procedure or a compound SQL (compiled) statement.

Description

label
Specifies the label for the FOR statement. If the beginning label is specified, that label can be used in LEAVE and ITERATE statements. If the ending label is specified, it must be the same as the beginning label.

for-loop-name
Specifies a label for the implicit compound statement generated to implement the FOR statement. It follows the rules for the label of a compound statement except that it cannot be used with an ITERATE or LEAVE statement within the FOR statement. The *for-loop-name* is used to qualify the column names returned by the specified *select-statement*.

cursor-name
Names the cursor that is used to select rows from the result table of the SELECT statement. If not specified, the database manager generates a unique cursor name. For a description of ASENSITIVE, INSENSITIVE, WITHOUT HOLD, or WITH HOLD, see "DECLARE CURSOR".

select-statement
Specifies the SELECT statement of the cursor. All columns in the select list must have a name and there cannot be two columns with the same name.

In a trigger, function, method, or compound SQL (inlined) statement, the *select-statement* must consist of only a *fullselect* with optional common table expressions.

SQL-procedure-statement
Specifies one or more statements to be invoked for each row of the table. *SQL-procedure-statement* is only applicable when in the context of an SQL procedure or within a compound SQL (compiled) statement. See *SQL-procedure-statement* in "Compound SQL (compiled)" statement.

SQL-function-statement
Specifies one or more statements to be invoked for each row of the table. A searched-update, searched-delete, or INSERT operation on nicknames is not supported. *SQL-function-statement* is only applicable when in the context of an SQL function or SQL method.

Rules

- The select list must consist of unique column names and the objects specified in the *select-statement* must exist when the procedure is created, or the object must be created in a previous SQL procedure statement.
- The cursor specified in a for-statement cannot be referenced outside the for-statement and cannot be specified in an OPEN, FETCH, or CLOSE statement.

Example

In the following example, the for-statement is used to iterate over the entire `employee` table. For each row in the table, the SQL variable `fullname` is set to the last name of the employee, followed by a comma, the first name, a blank space, and the middle initial. Each value for `fullname` is inserted into table `tnames`.

```
BEGIN ATOMIC
  DECLARE fullname CHAR(40);
  FOR vl AS
    SELECT firstnme, midinit, lastname FROM employee
      DO
      SET fullname = lastname CONCAT ','
        CONCAT firstnme CONCAT ' ' CONCAT midinit;
      INSERT INTO tnames VALUES (fullname);
  END FOR;
END
```

FREE LOCATOR

The FREE LOCATOR statement removes the association between a large object locator variable and its value.

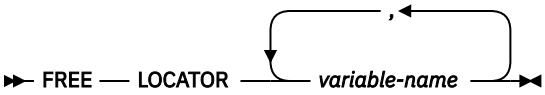
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

LOCATOR *variable-name, ...*

Identifies one or more large object locator variables that must be declared in accordance with the rules for declaring locator variables.

The locator-variable must currently have a locator assigned to it. That is, a locator must have been assigned during this unit of work (by a CALL, FETCH, SELECT INTO, or VALUES INTO statement) and must not subsequently have been freed (by a FREE LOCATOR statement); otherwise, an error is returned (SQLSTATE 0F001).

If more than one locator is specified, all locators that can be freed will be freed, regardless of errors detected in other locators in the list.

Example

In a COBOL program, free the BLOB locator variables TKN-VIDEO and TKN-BUF and the CLOB locator variable LIFE-STORY-LOCATOR.

```
EXEC SQL
FREE LOCATOR :TKN-VIDEO, :TKN-BUF, :LIFE-STORY-LOCATOR
END-EXEC.
```

GET DIAGNOSTICS

The GET DIAGNOSTICS statement is used to obtain current execution environment information including information about the previous SQL statement (other than a GET DIAGNOSTICS statement) that was executed. Some of the information available through the GET DIAGNOSTICS statement is also available in the SQLCA.

Invocation

This statement can be embedded in an:

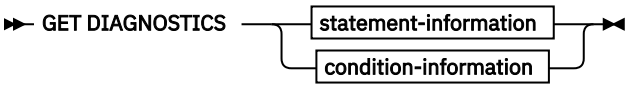
- SQL procedure definition
- Compound SQL (compiled) statement
- Compound SQL (inlined) statement

The compound statements can be embedded in an SQL procedure definition, SQL function definition, or SQL trigger definition. It is not an executable statement and cannot be dynamically prepared.

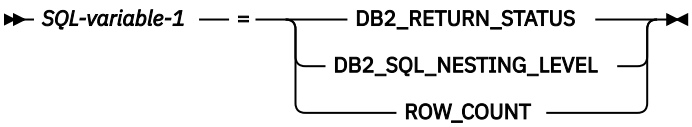
Authorization

None required.

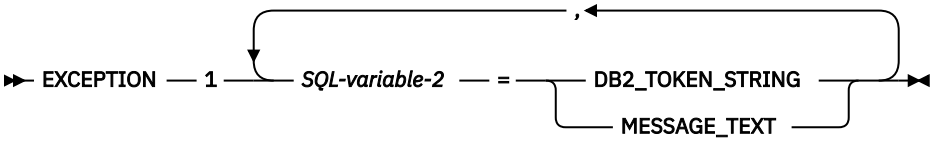
Syntax



statement-information



condition-information



Description

statement-information

Returns information about the last SQL statement executed.

SQL-variable-1

Identifies the variable that is the assignment target. The variable must not be a global variable. SQL variables can be defined in a compound statement. The data type of the variable must be compatible with the data type as specified in Table 151 on page 1673.

DB2_RETURN_STATUS

Identifies the status value returned from the procedure associated with the previously executed SQL statement, provided that the statement was a CALL statement invoking a procedure that returns a status. If the previous statement is not such a statement, then the value returned has no meaning and could be any integer.

DB2_SQL_NESTING_LEVEL

Identifies the current level of nesting or recursion in effect when the GET DIAGNOSTICS statement was executed. Each level of nesting corresponds to a nested or recursive invocation of a compiled SQL function, compiled SQL procedure, compiled trigger, or dynamically prepared compound SQL (compiled) statement. If the GET DIAGNOSTICS statement is executed outside of a level of nesting, the value zero is returned. This option can be specified only in the context of a compiled SQL function, compiled SQL procedure, compiled trigger, or compound SQL (compiled) statement (SQLSTATE 42601).

ROW_COUNT

Identifies the number of rows associated with the previous SQL statement. If the previous SQL statement is a DELETE, INSERT, or UPDATE statement, ROW_COUNT identifies the number of rows that qualified for the operation. If the previous statement is a PREPARE statement, ROW_COUNT identifies the *estimated* number of result rows in the prepared statement.

condition-information

Specifies that the error or warning information for the previously executed SQL statement is to be returned. If information about an error is needed, the GET DIAGNOSTICS statement must be the first statement specified in the handler that will handle the error. If information about a warning is needed, and if the handler will get control of the warning condition, the GET DIAGNOSTICS statement must be the first statement specified in that handler. If the handler will *not* get control of the warning condition, the GET DIAGNOSTICS statement must be the next statement executed. This option can only be specified in the context of an SQL Procedure (SQLSTATE 42601).

SQL-variable-2
Identifies the variable that is the assignment target. The variable must not be a global variable. SQL variables can be defined in a compound statement. The data type of the variable must be compatible with the data type as specified in Table 151 on page 1673.

DB2_TOKEN_STRING
Identifies any error or warning message tokens returned from the previously executed SQL statement. If the statement completed with an SQLCODE of zero, or if the SQLCODE had no tokens, an empty string is returned for a VARCHAR variable or blanks are returned for a CHAR variable.

MESSAGE_TEXT
Identifies any error or warning message text returned from the previously executed SQL statement. The message text is returned in the language of the database server where the statement is processed. If the statement completed with an SQLCODE of zero, an empty string is returned for a VARCHAR variable or blanks are returned for a CHAR variable.

Notes

- The GET DIAGNOSTICS statement does not change the contents of the diagnostics area (SQLCA). If an SQLSTATE or SQLCODE special variable is declared in the SQL procedure, these are set to the SQLSTATE or SQLCODE returned from issuing the GET DIAGNOSTICS statement.
- **Data types for items**: The following table shows the SQL data type for each diagnostic item. When a diagnostic item is assigned to a variable, the data type of the variable must be compatible with the data type of the requested diagnostic item.

Table 151. Data types for GET DIAGNOSTICS items

Type of information	Item	Data type
Statement information	DB2_RETURN_STATUS	INTEGER
Statement information	DB2_SQL_NESTING_LEVEL	INTEGER
Statement information	ROW_COUNT	DECIMAL(31,0)
Condition information	DB2_TOKEN_STRING	VARCHAR(1000)
Condition information	MESSAGE_TEXT	VARCHAR(32672)

- ***Syntax alternatives***: The following are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - RETURN_STATUS can be specified in place of DB2_RETURN_STATUS.

Examples

- *Example 1*: In an SQL procedure, execute a GET DIAGNOSTICS statement to determine how many rows were updated.

```
CREATE PROCEDURE sqlprocg (IN deptnbr VARCHAR(3))
   LANGUAGE SQL
   BEGIN
      DECLARE SQLSTATE CHAR(5);
      DECLARE rcount INTEGER;
      UPDATE CORPDATA.PROJECT
         SET PRSTAFF = PRSTAFF + 1.5
         WHERE DEPTNO = deptnbr;
      GET DIAGNOSTICS rcount = ROW_COUNT;
-- At this point, rcount contains the number of rows that were updated.
...
   END
```

- *Example 2*: Within an SQL procedure, handle the returned status value from the invocation of a procedure called TRYIT that could either explicitly RETURN a positive value indicating a user failure, or

encounter SQL errors that would result in a negative return status value. If the procedure is successful, it returns a value of zero.

```
CREATE PROCEDURE TESTIT ()
   LANGUAGE SQL
   A1:BEGIN
      DECLARE RETVAL INTEGER DEFAULT 0;
      ...
      CALL TRYIT;
      GET DIAGNOSTICS RETVAL = DB2_RETURN_STATUS;
      IF RETVAL <> 0 THEN
         ...
         LEAVE A1;
      ELSE
         ...
      END IF;
   END A1
```

GOTO

The GOTO statement is used to branch to a user-defined label within an SQL procedure.

Invocation

This statement can only be embedded in an SQL procedure. It is not an executable statement and cannot be dynamically prepared.

Authorization

None required.

Syntax

Description

label

Specifies a labelled statement where processing is to continue. The labelled statement and the GOTO statement must be in the same scope:

- If the GOTO statement is defined in a FOR statement, *label* must be defined inside the same FOR statement, excluding a nested FOR statement or nested compound statement
- If the GOTO statement is defined in a compound statement, *label* must be defined inside the same compound statement, excluding a nested FOR statement or nested compound statement
- If the GOTO statement is defined in a handler, *label* must be defined in the same handler, following the other scope rules
- If the GOTO statement is defined outside of a handler, *label* must not be defined within a handler.

If *label* is not defined within a scope that the GOTO statement can reach, an error is returned (SQLSTATE 42736).

Notes

- It is recommended that the GOTO statement be used sparingly. This statement interferes with normal processing sequences, thus making a routine more difficult to read and maintain. Before using a GOTO statement, determine whether another statement, such as IF or LEAVE, can be used in place, to eliminate the need for a GOTO statement.

Example

In the following compound statement, the parameters *rating* and *v_empno* are passed into the procedure, which then returns the output parameter *return_parm* as a date duration. If the employee's time in service with the company is less than 6 months, the GOTO statement transfers control to the end of the procedure, and *new_salary* is left unchanged.

```
CREATE PROCEDURE adjust_salary
  (IN v_empno CHAR(6),
  IN rating INTEGER,
  OUT return_parm DECIMAL (8,2))
  MODIFIES SQL DATA
  LANGUAGE SQL
  BEGIN
    DECLARE new_salary DECIMAL (9,2);
    DECLARE service DECIMAL (8,2);
      SELECT SALARY, CURRENT_DATE - HIREDATE
        INTO new_salary, service
        FROM EMPLOYEE
        WHERE EMPNO = v_empno;
      IF service < 600
        THEN GOTO EXIT;
      END IF;
      IF rating = 1
        THEN SET new_salary = new_salary + (new_salary * .10);
      ELSEIF rating = 2
        THEN SET new_salary = new_salary + (new_salary * .05);
      END IF;
      UPDATE EMPLOYEE
        SET SALARY = new_salary
        WHERE EMPNO = v_empno;
      EXIT: SET return_parm = service;
END
```

GRANT (database authorities)

This form of the GRANT statement grants authorities that apply to the entire database (rather than privileges that apply to specific objects within the database).

Invocation

This statement can be embedded in an application program or issued by using dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

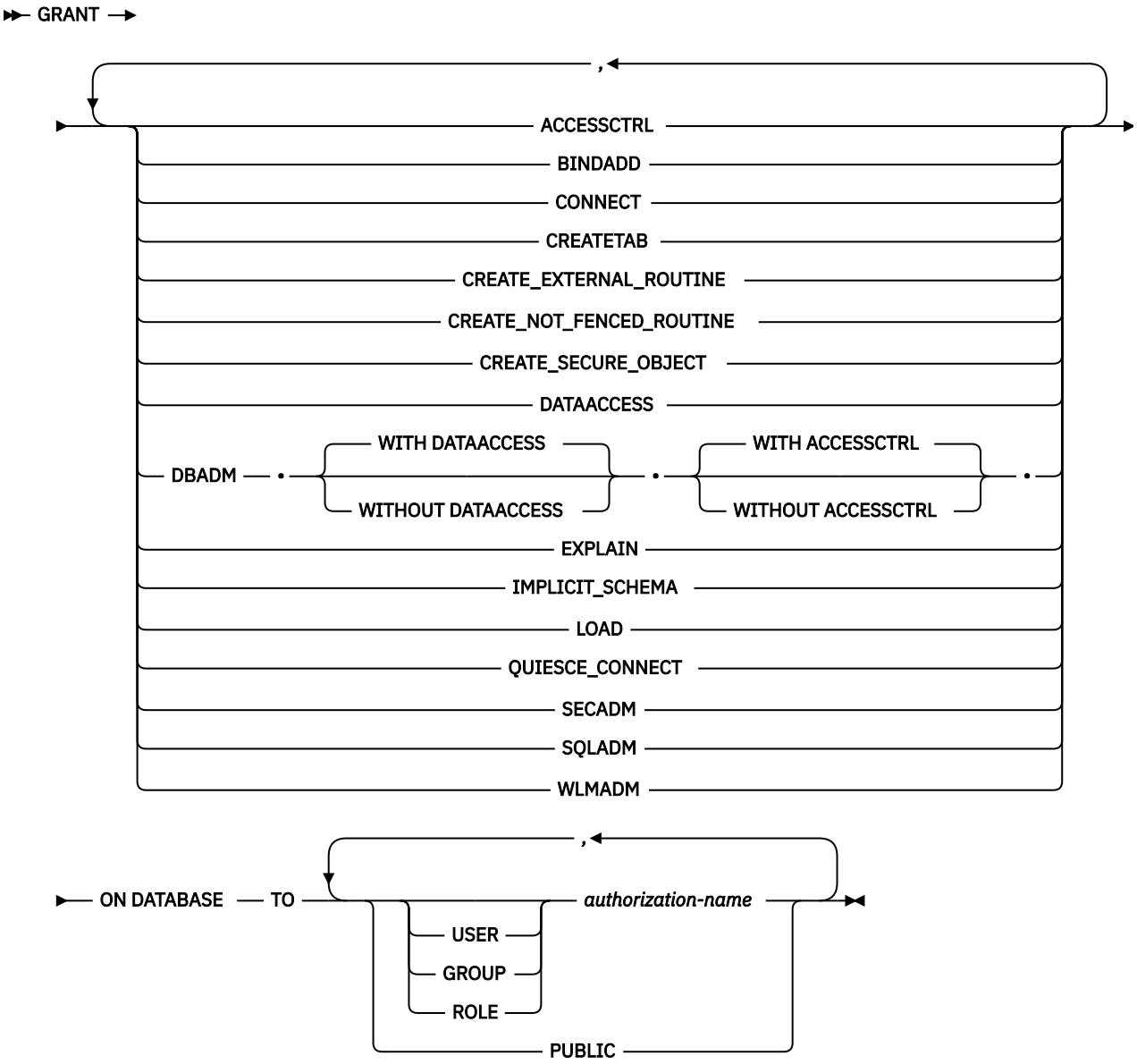
Authorization

To grant ACCESSCTRL, CREATE_SECURE_OBJECT, DATAACCESS, DBADM, or SECADM authority, SECADM authority is needed.

Note: In Db2 11.5.7 and later, to grant CREATE_EXTERNAL_ROUTINE authority, SYSADM authority is needed. If the DB2_ALTERNATE_AUTHZ_BEHAVIOUR registry variable is set and contains the value EXTERNAL_ROUTINE_DBAUTH, then SYSADM, SECADM, or ACCESSCTRL authority is needed.
Also, in Db2 11.5.7 and later, to grant CREATE_NOT_FENCED_ROUTINE authority, SYSADM authority is needed. If the DB2_ALTERNATE_AUTHZ_BEHAVIOUR registry variable is set and contains the value NOT_FENCED_ROUTINE_DBAUTH, then SYSADM, SECADM, or ACCESSCTRL authority is needed.

To grant other authorities ACCESSCTRL or SECADM authority is needed.

Syntax



Description

ACCESSCTRL

Grants the access control authority. The ACCESSCTRL authority allows the holder to:

- Grant and revoke the following database authorities: BINDADD, CONNECT, CREATETAB, CREATE_EXTERNAL_ROUTINE, CREATE_NOT_FENCED_ROUTINE, EXPLAIN, IMPLICIT_SCHEMA, LOAD, QUIESCE_CONNECT, SQLADM, WLMADM
- Grant and revoke all object level privileges.

The ACCESSCTRL authority cannot be granted to PUBLIC (SQLSTATE 42508).

BINDADD

Grants the authority to create packages. The creator of a package automatically has the CONTROL privilege on that package and retains this privilege even if the BINDADD authority is later revoked.

CONNECT

Grants the authority to access the database.

CREATETAB
Grants the authority to create base tables. The creator of a base table automatically has the CONTROL privilege on that table. The creator retains this privilege even if the CREATETAB authority is later revoked.

No explicit authority is needed for view creation. A view can be created at any time if the authorization ID of the statement that is used to create the view has either CONTROL or SELECT privilege on each base table of the view.

CREATE_EXTERNAL_ROUTINE
Grants the authority to register external routines. Care must be taken that routines so registered do not have adverse side effects. (For more information, see the description of the THREADSAFE clause in “CREATE PROCEDURE (external) ” on page 1292).

After an external routine has been registered, it continues to exist, even if CREATE_EXTERNAL_ROUTINE is later revoked.

CREATE_NOT_FENCED_ROUTINE
Grants the authority to register routines that run in the database manager's process. Care must be taken that routines so registered do not have adverse side effects. (For more information, see the description of the FENCED clause on the “CREATE PROCEDURE (external) ” on page 1292).

After a routine is registered as not fenced, it continues to run in this manner, even if CREATE_NOT_FENCED_ROUTINE is later revoked.

CREATE_EXTERNAL_ROUTINE is automatically granted to an *authorization-name* that is granted CREATE_NOT_FENCED_ROUTINE authority.

CREATE_SECURE_OBJECT
Grants the authority to create secure triggers and secure functions. Grants the authority to alter the secure attribute of such objects as well.

DATAACCESS
Grants the authority to access data. The DATAACCESS authority allows the holder to:

- Select, insert, update, delete, and load data.
- Run any package.
- Run any routine (except audit routines).

The DATAACCESS authority cannot be granted to PUBLIC (SQLSTATE 42508).

DBADM
Grants the database administrator authority. A database administrator holds nearly all privileges on nearly all objects in the database. The only exceptions are those privileges that are part of the access control, data access, and security administrator authorities. DBADM cannot be granted to PUBLIC.

EXPLAIN
Grants the authority to explain statements. The EXPLAIN authority allows the holder to explain, prepare, and describe dynamic and static SQL statements without requiring access to data.

IMPLICIT_SCHEMA
Grants the authority to implicitly create a schema.

LOAD
Grants the authority to load in this database. This authority gives a user the right to use the LOAD utility in this database. DATAACCESS and DBADM also have this authority by default. However, if a user only has LOAD authority (not DATAACCESS), the user is also needs to have table-level privileges. In addition to LOAD privilege, the user needs to have:

- INSERT privilege on the table for LOAD with mode INSERT, TERMINATE (to terminate a previous LOAD INSERT), or RESTART (to restart a previous LOAD INSERT).
- INSERT and DELETE privilege on the table for LOAD with mode REPLACE, TERMINATE (to terminate a previous LOAD REPLACE), or RESTART (to restart a previous LOAD REPLACE).
- INSERT privilege on the exception table, if such a table is used as part of LOAD.

QUIESCE_CONNECT
Grants the authority to access the database while it is quiesced.

SECADM
Grants the security administrator authority. The authority allows the holder to:

- Create and drop security objects such as audit policies, roles, security labels, security label components, security policies, and trusted contexts.
- Grant and revoke authorities, exemptions, privileges, roles, and security labels.
- Grant and revoke the SETSESSIONUSER privilege.
- Run TRANSFER OWNERSHIP on objects that are owned by others.

The SECADM authority cannot be granted to PUBLIC (SQLSTATE 42508).

SQLADM
Grants the authority to manage SQL statement execution. The SQLADM authority allows the holder to:

- Create, drop, flush, and set event monitors.
- Explain, prepare, and describe dynamic and static SQL statements without requiring access to data.
- Flush optimization profile cache
- Flush package cache
- Execute the runstats utility.
- Create, alter, drop, and set usage lists.

WLMADM
Grants the authority to manage workloads. The WLMADM authority allows the holder to:

- Create, drop, and alter service classes, thresholds, work action sets, work class sets, or workloads.

TO
Specifies to whom the authorities are granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name. The role name must exist at the current server (SQLSTATE 42704).

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Grants the authorities to a set of users (authorization IDs).

Rules

- For each *authorization-name* specified, if neither USER, GROUP, or ROLE is specified, then:
  - If the security plug-in in effect for the instance cannot determine the status of the *authorization-name*, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as ROLE in the database, and as either GROUP or USER according to the security plug-in in effect, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as both USER and GROUP, an error is returned (SQLSTATE 56092).

- If the *authorization-name* is defined according to the security plug-in in effect as USER only, or if it is undefined, USER is assumed.
- If the *authorization-name* is defined according to the security plug-in in effect as GROUP only, GROUP is assumed.
- If the *authorization-name* is defined in the database as ROLE only, ROLE is assumed.

Notes

- ACCESSCTRL, CREATE_SECURE_OBJECT, DATAACCESS, DBADM, or SECADM authorities cannot be granted to the special group PUBLIC. Therefore, granting ACCESSCTRL, CREATE_SECURE_OBJECT, DBADM, DATAACCESS, or SECADM authority to a role *role-name* fails if *role-name* is granted to PUBLIC either directly or indirectly (SQLSTATE 42508).
  - Role *role-name* is granted directly to PUBLIC if the following statement has been issued:

    ```
    GRANT ROLE role-name TO PUBLIC
    ```

  - Role *role-name* is granted indirectly to PUBLIC if the following statements have been issued:

    ```
    GRANT ROLE role-name TO ROLE role-name2
    GRANT ROLE role-name2 TO PUBLIC
    ```

- ***Syntax alternatives***: The following are supported for compatibility with previous versions of Db2 and with other database products.
  - CREATE_NOT_FENCED can be specified in place of CREATE_NOT_FENCED_ROUTINE.
  - SYSTEM can be specified in place of DATABASE.
- ***Privileges granted to a group***: A privilege that is granted to a group is not used for authorization checking on:
  - Static DML statements in a package
  - A base table while processing a CREATE VIEW statement
  - A base table while processing a CREATE TABLE statement for a materialized query table
  - Create SQL routine
  - Create trigger
- WITH GRANT OPTION is ignored when granting database authorities.

Examples

- *Example 1:* Give the users WINKEN, BLINKEN, and NOD the authority to connect to the database.

  ```
  GRANT CONNECT ON DATABASE TO USER WINKEN, USER BLINKEN, USER NOD
  ```

- *Example 2:* Grant BINDADD authority on the database to a group named D024. Both a group and a user called D024 exist in the system.

  ```
  GRANT BINDADD ON DATABASE TO GROUP D024
  ```

  Observe that, the GROUP keyword must be specified; otherwise, an error will occur since both a user and a group named D024 exist. Any member of the D024 group will be allowed to bind packages in the database, but the D024 user will not be allowed (unless this user is also a member of the group D024, had been granted BINDADD authority previously, or BINDADD authority had been granted to another group of which D024 was a member).

- *Example 3:* Give user Walid security administrator authority.

  ```
  GRANT SECADM ON DATABASE TO USER Walid
  ```

- *Example 4:* A user with SECADM authority grants the CREATE_SECURE_OBJECT authority to user Haytham.

```
GRANT CREATE_SECURE_OBJECT ON DATABASE TO USER HAYTHAM
```

GRANT (exemption)

This form of the GRANT statement grants to a user, group, or role an exemption on an access rule for a specified label-based access control (LBAC) security policy.

When the user holding the exemption accesses data in a table protected by that security policy the indicated rule will not be enforced when deciding if they can access the data.

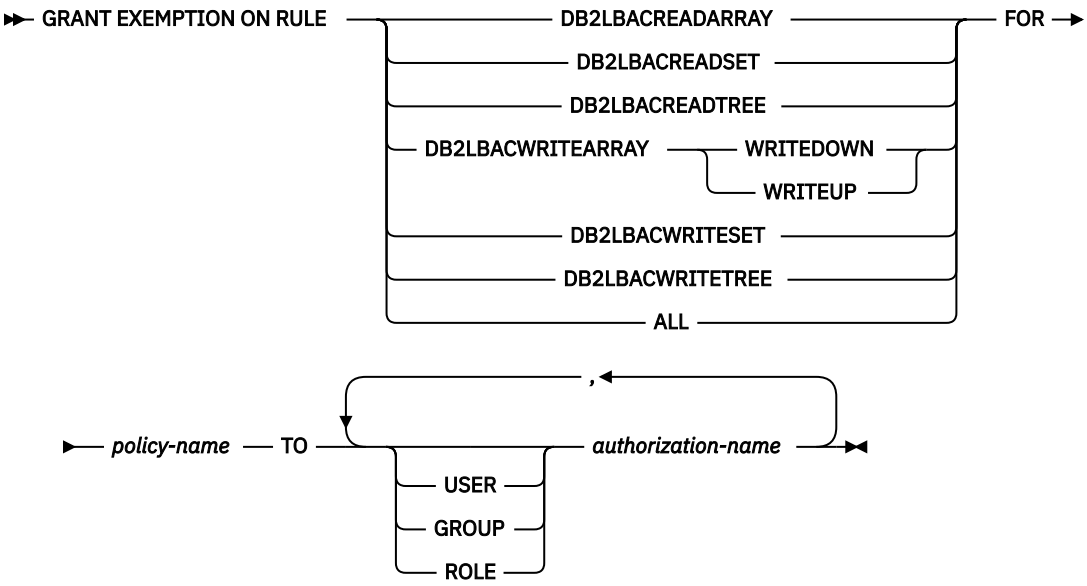
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include SECADM authority.

Syntax



Description

EXEMPTION ON RULE
Grants an exemption on an access rule.

DB2LBACREADARRAY
Grants an exemption on the predefined DB2LBACREADARRAY rule.

DB2LBACREADSET
Grants an exemption on the predefined DB2LBACREADSET rule.

DB2LBACREADTREE
Grants an exemption on the predefined DB2LBACREADTREE rule.

DB2LBACWRITEARRAY
Grants an exemption on the predefined DB2LBACWRITEARRAY rule.

WRITEDOWN
Specifies that the exemption only applies to write down.

WRITEUP
Specifies that the exemption only applies to write up.

DB2LBACWRITESET
Grants an exemption on the predefined DB2LBACWRITESET rule.

DB2LBACWRITETREE
Grants an exemption on the predefined DB2LBACWRITETREE rule.

ALL
Grants an exemption on all of the predefined rules.

FOR *policy-name*
Identifies the security policy for which the exemption is being granted. The exemption will only be effective for tables that are protected by this security policy. The name must identify a security policy already described in the catalog (SQLSTATE 42704).

TO
Specifies to whom the exemption is granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name. The role name must exist at the current server (SQLSTATE 42704).

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

Rules

- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:
  - If the security plug-in in effect for the instance cannot determine the status of the *authorization-name*, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as ROLE in the database, and as either GROUP or USER according to the security plug-in in effect, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as both USER and GROUP, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as USER only, or if it is undefined, USER is assumed.
  - If the *authorization-name* is defined according to the security plug-in in effect as GROUP only, GROUP is assumed.
  - If the *authorization-name* is defined in the database as ROLE only, ROLE is assumed.
- If the security policy is not defined to consider access through groups or roles, any exemption granted to a group or role is ignored when access is attempted.

Notes

- By default when a security policy is created, only exemptions granted to an individual user are considered. To have groups or roles considered for the security policy, you must issue the ALTER SECURITY POLICY statement and specify USE GROUP AUTHORIZATION or USE ROLE AUTHORIZATION as applicable.

Examples

- *Example 1:* Grant an exemption on access rule DB2LBACREADSET for security policy DATA_ACCESS to user WALID.

```
GRANT EXEMPTION ON RULE DB2LBACREADSET FOR DATA_ACCESS TO USER WALID
```

- *Example 2:* Grant an exemption on access rule DB2LBACWRITEARRAY with the WRITEDOWN option for security policy DATA_ACCESS to user BOBBY.

```
GRANT EXEMPTION ON RULE DB2LBACWRITEARRAY WRITEDOWN
  FOR DATA_ACCESS TO USER BOBBY
```

- *Example 3:* Grant an exemption on access rule DB2LBACWRITEARRAY with the WRITEUP option for security policy DATA_ACCESS to user BOBBY.

```
GRANT EXEMPTION ON RULE DB2LBACWRITEARRAY WRITEUP
  FOR DATA_ACCESS TO USER BOBBY
```

GRANT (global variable privileges)

This form of the GRANT statement grants one or more privileges on a created global variable.

Invocation

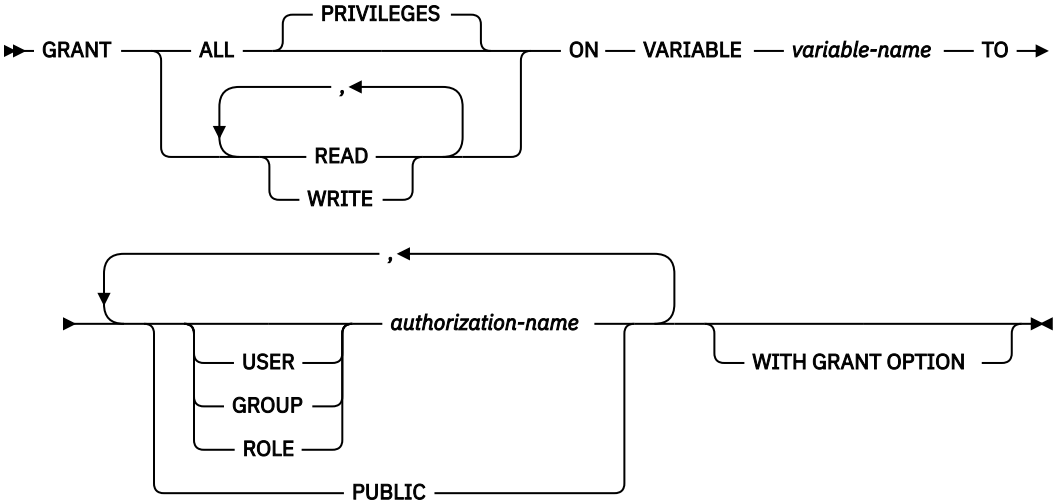
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- The WITH GRANT OPTION for each identified privilege on the global variable
- ACCESSCTRL authority on the schema containing the global variable
- ACCESSCTRL or SECADM authority

Syntax



Description

ALL PRIVILEGES
Grants all privileges on the specified global variable.

READ
Grants the privilege to read the value of the specified global variable.

WRITE
Grants the privilege to assign a value to the specified global variable.

ON VARIABLE *variable-name*
Identifies the global variable on which one or more privileges are to be granted. The *variable-name*, including an implicit or explicit qualifier, must identify a global variable that exists at the current server and is not a module variable (SQLSTATE 42704).

TO
Specifies to whom the privileges are granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group.

ROLE
Specifies that the *authorization-name* identifies an existing role at the current server (SQLSTATE 42704).

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles. The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Grants the specified privileges to a set of users (authorization IDs).

WITH GRANT OPTION
Allows the specified *authorization-name* to grant the privileges to others. If the WITH GRANT OPTION clause is omitted, the specified *authorization-name* cannot grant the privileges to others unless that authority has been received from some other source.

Rules

- For each *authorization-name* specified, if none of the keywords USER, GROUP, or ROLE is specified:
  - If the security plug-in in effect for the instance cannot determine the status of the *authorization-name*, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as ROLE in the database and as either GROUP or USER in the operating system, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as both USER and GROUP according to the security plug-in in effect, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as USER only according to the security plug-in in effect, or if it is undefined, USER is assumed.
  - If the *authorization-name* is defined as GROUP only according to the security plug-in in effect, GROUP is assumed.
  - If the *authorization-name* is defined in the database as ROLE only, ROLE is assumed.

Notes

- ***Privileges granted to a group***: A privilege that is granted to a group is not used for authorization checking on:
  - Static DML statements in a package

- A base table while processing a CREATE VIEW statement
- A base table while processing a CREATE TABLE statement for a materialized query table
- Create SQL routine
- Create trigger

Example

Grant the READ and WRITE privilege on global variable MYSCHEMA.MYJOB_PRINTER to user ZUBIRI.

```
GRANT READ, WRITE ON VARIABLE MYSCHEMA.MYJOB_PRINTER TO ZUBIRI
```

GRANT (index privileges)

This form of the GRANT statement grants the CONTROL privilege on indexes.

Invocation

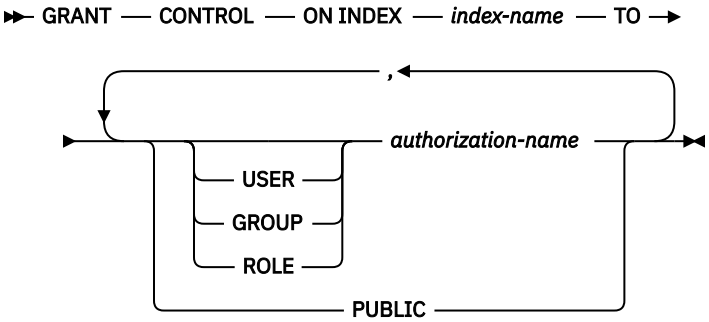
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The authorization ID privileges of the statement on the schema containing the index must include one of the following:

- ACCESSCTRL authority
- SECADM authority
- Schema ACCESSCTRL authority

Syntax



Description

CONTROL
Grants the privilege to drop the index. This is the CONTROL authority for indexes, which is automatically granted to creators of indexes.

ON INDEX *index-name*
Identifies the index for which the CONTROL privilege is to be granted.

TO
Specifies to whom the privileges are granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name. The role name must exist at the current server (SQLSTATE 42704).

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Grants the privileges to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership".

Rules

- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:
  - If the security plug-in in effect for the instance cannot determine the status of the *authorization-name*, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as ROLE in the database, and as either GROUP or USER according to the security plug-in in effect, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as both USER and GROUP, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as USER only, or if it is undefined, USER is assumed.
  - If the *authorization-name* is defined according to the security plug-in in effect as GROUP only, GROUP is assumed.
  - If the *authorization-name* is defined in the database as ROLE only, ROLE is assumed.

Notes

- ***Privileges granted to a group***: A privilege that is granted to a group is not used for authorization checking on:
  - Static DML statements in a package
  - A base table while processing a CREATE VIEW statement
  - A base table while processing a CREATE TABLE statement for a materialized query table
  - Create SQL routine
  - Create trigger
- WITH GRANT OPTION is ignored when granting index privileges

Example

Grant CONTROL privilege on the DEPTIDX index to the user whose ID is KIESLER:

```
GRANT CONTROL ON INDEX DEPTIDX TO USER KIESLER
```

GRANT (module privileges)

This form of the GRANT statement grants privileges on a module.

Invocation

This statement can be embedded in an application program or issued by using dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges that are held by the authorization ID of the statement must include at least one of the following authorities:

- The WITH GRANT OPTION for EXECUTE on the module.
- ACCESSCTRL authority on the schema that contains the module.
- ACCESSCTRL or SECADM authority.

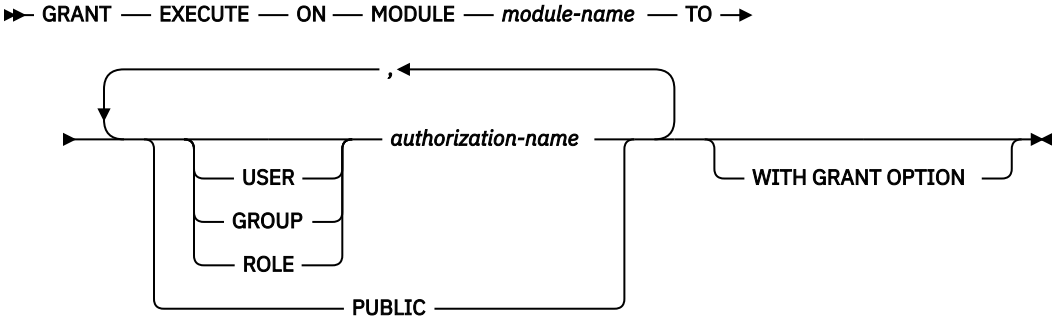
Note: In Db2 11.5.7 and later, the needed authorities are different if the module is SYSIBMADM.UTL_DIR. In this case, the authorities that are held by the authorization ID of the statement must include at least one of the following options:

- The WITH GRANT OPTION for EXECUTE on the module.
- SYSADM authority.

If the module on which the privilege is granted is SYSIBMADM.UTL_DIR, and the DB2_ALTERNATE_AUTHZ_BEHAVIOUR registry variable is set to UTL_DIR_DBAUTH, then the privileges that are held by the authorization ID of the statement are different. In this case, the needed privileges must include at least one of the following options:

- The WITH GRANT OPTION for EXECUTE on the module
- ACCESSCTRL authority on the schema that contains the module.
- SYSADM, ACCESSCTRL, or SECADM authority

Syntax



Description

EXECUTE

Grants the privilege to reference published module objects and run the following operations:

- Run any published routines defined in the module.
- Read from and write to any published global variables defined in the module.
- Reference any published user-defined types defined in the module.
- Reference any published conditions defined in the module.

ON MODULE ***module-name***
Identifies the module on which the privilege is granted. The module-name must identify a module that exists at the current server (SQLSTATE 42704).

TO
Indicates to whom the privilege is granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name. The role name must exist at the current server (SQLSTATE 42704).

authorization-name,...
Lists one or more authorization IDs.

PUBLIC
Grants the privilege to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership".

WITH GRANT OPTION
Allows the specified *authorization-name*s to grant the EXECUTE privilege to other users. If WITH GRANT OPTION is omitted, the specified *authorization-name*s cannot grant the EXECUTE privilege to others unless they receive that authority from some other source.

Notes

Privileges that are granted to a group

- A privilege that is granted to a group is not used for authorization checking on any of the following items:
  - Static DML statements in a package.
  - A base table that is actively processing a CREATE VIEW statement.
  - A base table that is actively processing a CREATE TABLE statement for a materialized query table.
  - Create SQL routine.
  - Create trigger.

Example

Grant the EXECUTE privilege on module MYMODA to user JONES:

```
GRANT EXECUTE
  ON MODULE MYMODA
    TO JONES
```

GRANT (package privileges)

This form of the GRANT statement grants privileges on a package.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

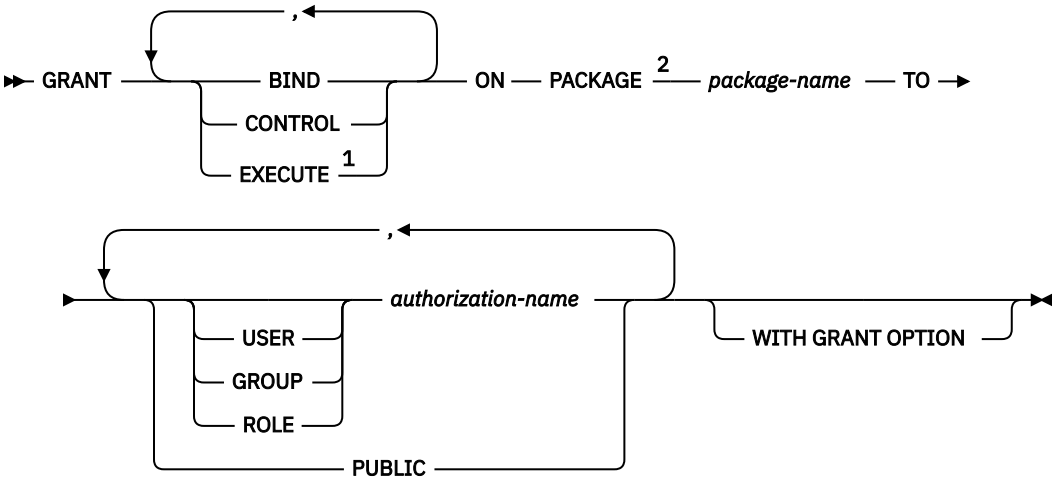
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- CONTROL privilege on the referenced package
- The WITH GRANT OPTION for each identified privilege on *package-name*
- ACCESSCTRL on the schema containing the package
- ACCESSCTRL or SECADM authority

One of the following authorities is required to grant the CONTROL privilege:

- ACCESSCTRL
- SECADM
- ACCESSCTRL authority on the schema containing the package

Syntax



Notes:

[1] RUN can be used as a synonym for EXECUTE.
[2] PROGRAM can be used as a synonym for PACKAGE.

Description

BIND

Grants the privilege to bind a package. The BIND privilege allows a user to re-issue the BIND command against that package, or to issue the REBIND command. It also allows a user to create a new version of an existing package.

In addition to the BIND privilege, a user must hold the necessary privileges on each table referenced by static DML statements contained in a program. This is necessary, because authorization on static DML statements is checked at bind time.

CONTROL

Grants the privilege to rebind, drop, or execute the package, and extend package privileges to other users. The CONTROL privilege for packages is automatically granted to creators of packages. A package owner is the package binder, or the ID specified with the OWNER option at bind/precompile time.

BIND and EXECUTE are automatically granted to an *authorization-name* that is granted CONTROL privilege.

CONTROL grants the ability to grant the previously mentioned privileges (except for CONTROL) to others.

EXECUTE

Grants the privilege to execute the package.

ON PACKAGE *package-name*
Specifies the name of the package on which privileges are to be granted. The granting of a package privilege applies to all versions of the package (that is, to all packages that share the same package name and package schema).

TO
Specifies to whom the privileges are granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name. The role name must exist at the current server (SQLSTATE 42704).

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Grants the privileges to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership".

WITH GRANT OPTION
Allows the specified *authorization-name* to GRANT the privileges to others.

If the specified privileges include CONTROL, the WITH GRANT OPTION applies to all of the applicable privileges except for CONTROL (SQLSTATE 01516).

Rules

- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:
  - If the security plug-in in effect for the instance cannot determine the status of the *authorization-name*, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as ROLE in the database, and as either GROUP or USER according to the security plug-in in effect, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as both USER and GROUP, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as USER only, or if it is undefined, USER is assumed.
  - If the *authorization-name* is defined according to the security plug-in in effect as GROUP only, GROUP is assumed.
  - If the *authorization-name* is defined in the database as ROLE only, ROLE is assumed.

Notes

- Package privileges apply to all versions of a package (that is, all packages that share the same package ID and package schema). It is not possible to restrict access to only one version. Because CONTROL privilege is implicitly granted to the binder of a package, if two different users bind two versions of a package, then both users will implicitly be granted access to each other's package.
- ***Privileges granted to a group***: A privilege that is granted to a group is not used for authorization checking on:
  - Static DML statements in a package
  - A base table while processing a CREATE VIEW statement

- A base table while processing a CREATE TABLE statement for a materialized query table
- Create SQL routine
- Create trigger

Examples

- *Example 1:* Grant the EXECUTE privilege on PACKAGE CORPDATA.PKGA to PUBLIC.

```
GRANT EXECUTE
  ON PACKAGE CORPDATA.PKGA
  TO PUBLIC
```

- *Example 2:* GRANT EXECUTE privilege on package CORPDATA.PKGA to a user named EMPLOYEE. There is neither a group nor a user called EMPLOYEE.

```
GRANT EXECUTE ON PACKAGE
  CORPDATA.PKGA TO EMPLOYEE
```

or

```
GRANT EXECUTE ON PACKAGE
  CORPDATA.PKGA TO USER EMPLOYEE
```

GRANT (role)

This form of the GRANT statement grants roles to users, groups, or to other roles.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

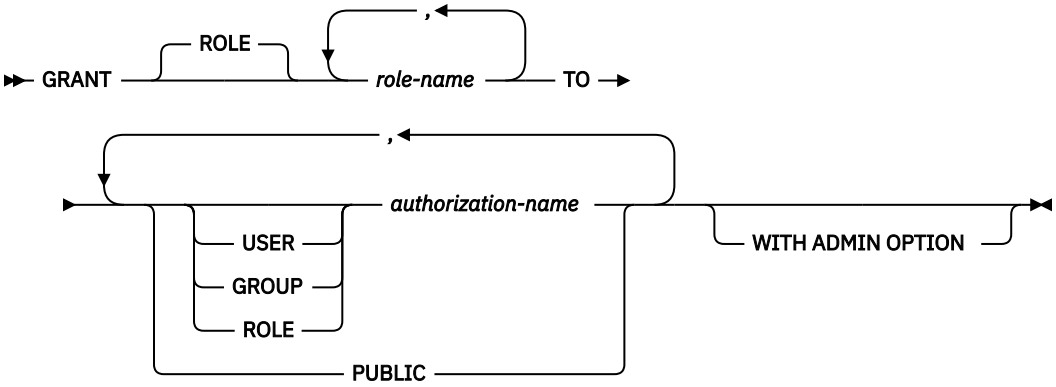
Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- The WITH ADMIN OPTION on the role
- SECADM authority

SECADM authority is required to grant the WITH ADMIN OPTION to an *authorization-name*.

Syntax



Description

ROLE *role-name*,...
Identifies one or more roles to be granted. Each *role-name* must identify an existing role at the current server (SQLSTATE 42704).

TO
Specifies to whom the role is granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group.

ROLE
Specifies that the *authorization-name* identifies an existing role at the current server (SQLSTATE 42704).

***authorization-name*,...**
Lists the authorization IDs of one or more users, groups, or roles. The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Grants the specified roles to a set of users (authorization IDs).

WITH ADMIN OPTION
Allows the specified *authorization-name* to grant or revoke the *role-name* to or from others, or to associate a comment with the role. It does not allow the specified *authorization-name* to drop the role.

Rules

- For each *authorization-name* specified, if none of the keywords USER, GROUP, or ROLE is specified:
  - If the security plug-in in effect for the instance cannot determine the status of the *authorization-name*, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as ROLE in the database and as either GROUP or USER in the operating system, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as both USER and GROUP according to the security plug-in in effect, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as USER only according to the security plug-in in effect, or if it is undefined, USER is assumed.
  - If the *authorization-name* is defined as GROUP only according to the security plug-in in effect, GROUP is assumed.
  - If the *authorization-name* is defined in the database as ROLE only, ROLE is assumed.
- Hierarchies of roles can be built by granting one role to another role. However, cycles are not allowed (SQLSTATE 428GF). For example, if role R1 is granted to another role R2, then role R2 (or some other role R*n* that contains R2) cannot be granted back to R1, because this would produce a cycle.
- The USER, GROUP, or ROLE names must not begin with the characters 'SYS' and must not be 'ACCESSCTRL', 'DATAACCESS', 'DBADM', 'NONE', 'NULL', 'PUBLIC', 'SECADM', 'SQLADM', 'SCHEMAADM', or 'WLMADM' (SQLSTATE 42939).

Notes

- When role R1 is granted to another role R2, then R2 contains R1.
- DBADM authority cannot be granted to PUBLIC. Therefore:
  - Granting role R1 to PUBLIC fails (SQLSTATE 42508) if role R1 holds DBADM authority either directly or indirectly.

- Role R1 holds DBADM authority directly if the following statement has been issued:

```
GRANT DBADM ON DATABASE TO ROLE R1
```

- Role R1 holds DBADM authority indirectly if the following statements have been issued:

```
GRANT DBADM ON DATABASE TO ROLE R2
GRANT ROLE R2 TO ROLE R1
```

– Granting role R1, which holds DBADM authority, to role R2 fails (SQLSTATE 42508) if role R2 is granted to PUBLIC either directly or indirectly.

- Role R2 is granted to PUBLIC directly if the following statement has been issued:

```
GRANT ROLE R2 TO PUBLIC
```

- Role R2 is granted to PUBLIC indirectly if the following statements have been issued:

```
GRANT ROLE R2 TO ROLE R3
GRANT ROLE R3 TO PUBLIC
```

- No schema-level authority (SCHEMAADM, schema ACCESSCTRL, schema DATAACCESS, schema LOAD) can be granted to PUBLIC.
- ***Privileges granted to a group***: A privilege that is granted to a group is not used for authorization checking on:
  – Static DML statements in a package
  – A base table while processing a CREATE VIEW statement
  – A base table while processing a CREATE TABLE statement for a materialized query table
  – Create SQL routine
  – Create trigger

Examples

- *Example 1:* Grant role INTERN to role DOCTOR and role DOCTOR to role SPECIALIST.

```
GRANT ROLE INTERN TO ROLE DOCTOR
GRANT ROLE DOCTOR TO ROLE SPECIALIST
```

- *Example 2:* Grant role INTERN to PUBLIC.

```
GRANT ROLE INTERN TO PUBLIC
```

- *Example 3:* Grant role SPECIALIST to user BOB and group TORONTO.

```
GRANT ROLE SPECIALIST TO USER BOB, GROUP TORONTO
```

GRANT (routine privileges)

This form of the GRANT statement grants privileges on a routine (function, method, or procedure) that is not defined in a module.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

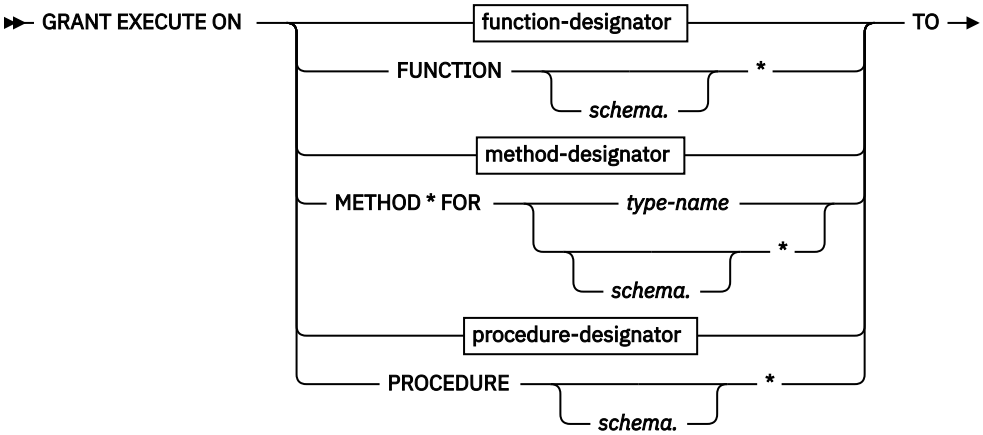
- The WITH GRANT OPTION for EXECUTE on the routine
- ACCESSCTRL authority on the schema containing the routine
- ACCESSCTRL or SECADM authority

To grant all routine EXECUTE privileges in the schema or type, the privileges held by the authorization ID of the statement must include at least one of the following authorities:

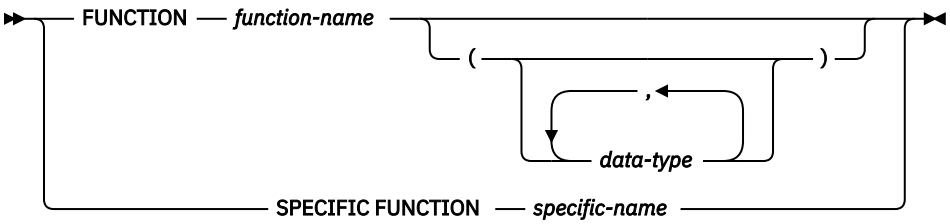
- The WITH GRANT OPTION for EXECUTE on all existing and future routines (of the specified type) in the specified schema
- ACCESSCTRL authority on the schema containing all the routines
- ACCESSCTRL or SECADM authority

SECADM authority is required to grant EXECUTE privilege on audit routines and the SET_MAINT_MODE_RECORD_NO_TEMPORALHISTORY procedure. EXECUTE privilege WITH GRANT OPTION cannot be granted for these routines (SQLSTATE 42501). EXECUTE privilege cannot be granted to PUBLIC on the SET_MAINT_MODE_RECORD_NO_TEMPORALHISTORY procedure (SQLSTATE 42501).

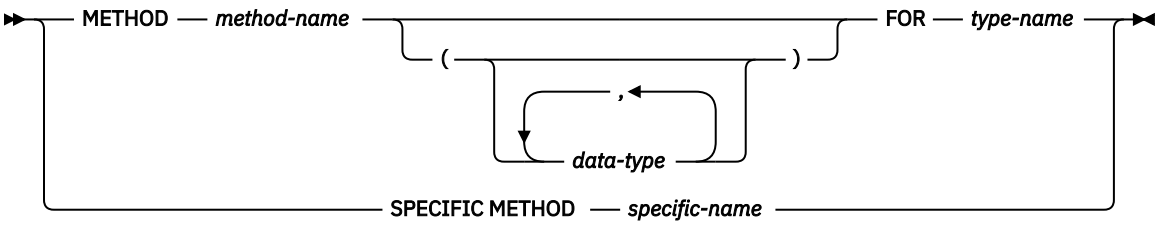
Syntax



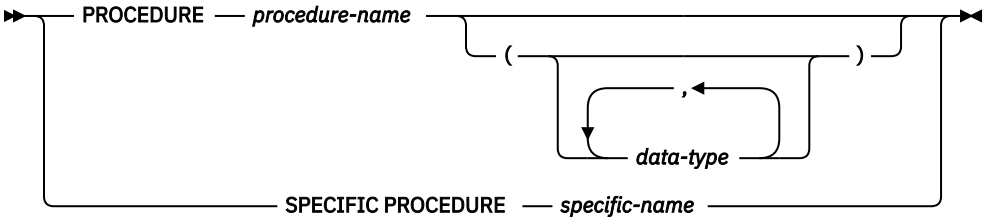
function-designator



method-designator



procedure-designator



Description

EXECUTE
Grants the privilege to run the identified user-defined function, method, or procedure.

function-designator
Uniquely identifies the function on which the privilege is granted. For more information, see "Function, method, and procedure designators" on page 745.

FUNCTION *schema*.*
Identifies all the functions in the schema, including any functions that may be created in the future. In dynamic SQL statements, if a schema is not specified, the schema in the CURRENT SCHEMA special register will be used. In static SQL statements, if a schema is not specified, the schema in the QUALIFIER precompile/bind option will be used.

method-designator
Uniquely identifies the method on which the privilege is granted. For more information, see "Function, method, and procedure designators" on page 745.

METHOD *
Identifies all the methods for the type *type-name*, including any methods that may be created in the future.

FOR *type-name*
Names the type in which the specified method is found. The name must identify a type already described in the catalog (SQLSTATE 42704). In dynamic SQL statements, the value of the CURRENT SCHEMA special register is used as a qualifier for an unqualified type name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified type names. An asterisk (*) can be used in place of *type-name* to identify all types in the schema, including any types that may be created in the future.

procedure-designator
Uniquely identifies the procedure on which the privilege is granted. For more information, see "Function, method, and procedure designators" on page 745.

PROCEDURE *schema*.*
Identifies all the procedures in the schema, including any procedures that may be created in the future. In dynamic SQL statements, if a schema is not specified, the schema in the CURRENT SCHEMA special register will be used. In static SQL statements, if a schema is not specified, the schema in the QUALIFIER precompile/bind option will be used.

TO
Specifies to whom the EXECUTE privilege is granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name. The role name must exist at the current server (SQLSTATE 42704).

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

PUBLIC
Grants the EXECUTE privilege to a set of users (authorization IDs).

WITH GRANT OPTION
Allows the specified *authorization-name*s to GRANT the EXECUTE privilege to others.

If the WITH GRANT OPTION is omitted, the specified *authorization-name* can only grant the EXECUTE privilege to others if they:

- have SYSADM or DBADM authority or
- received the ability to grant the EXECUTE privilege from some other source.

Rules

- It is not possible to grant the EXECUTE privilege on a function or method defined with schema 'SYSIBM' or 'SYSFUN' (SQLSTATE 42832).
- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:
  - If the security plug-in in effect for the instance cannot determine the status of the *authorization-name*, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as ROLE in the database, and as either GROUP or USER according to the security plug-in in effect, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as both USER and GROUP, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as USER only, or if it is undefined, USER is assumed.
  - If the *authorization-name* is defined according to the security plug-in in effect as GROUP only, GROUP is assumed.
  - If the *authorization-name* is defined in the database as ROLE only, ROLE is assumed.
- In general, the GRANT statement will process the granting of privileges that the authorization ID of the statement is allowed to grant, returning a warning (SQLSTATE 01007) if one or more privileges was not granted. If the package used for processing the statement was precompiled with LANGLEVEL set to SQL92E or MIA, and no privileges were granted, a warning is returned (SQLSTATE 01007). If the grantor has no privileges on the object of the grant operation, an error is returned (SQLSTATE 42501).

Notes

- Privileges for a routine defined in a module are granted at the module level using the GRANT (module privileges) statement. The EXECUTE privilege on the module allows access to all objects in the module.
- ***Privileges granted to a group***: A privilege that is granted to a group is not used for authorization checking on:
  - Static DML statements in a package
  - A base table while processing a CREATE VIEW statement
  - A base table while processing a CREATE TABLE statement for a materialized query table
  - Create SQL routine

– Create trigger

Examples

- *Example 1:* Grant the EXECUTE privilege on function CALC_SALARY to user JONES. Assume that there is only one function in the schema with function name CALC_SALARY.

```
GRANT EXECUTE ON FUNCTION CALC_SALARY TO JONES
```

- *Example 2:* Grant the EXECUTE privilege on procedure VACATION_ACCR to all users at the current server.

```
GRANT EXECUTE ON PROCEDURE VACATION_ACCR TO PUBLIC
```

- *Example 3:* Grant the EXECUTE privilege on function DEPT_TOTALS to the administrative assistant and give the assistant the ability to grant the EXECUTE privilege on this function to others. The function has the specific name DEPT85_TOT. Assume that the schema has more than one function named DEPT_TOTALS.

```
GRANT EXECUTE ON SPECIFIC FUNCTION DEPT85_TOT
  TO ADMIN_A WITH GRANT OPTION
```

- *Example 4:* Grant the EXECUTE privilege on function NEW_DEPT_HIRES to HR (Human Resources). The function has two input parameters of type INTEGER and CHAR(10), respectively. Assume that the schema has more than one function named NEW_DEPT_HIRES.

```
GRANT EXECUTE ON FUNCTION NEW_DEPT_HIRES (INTEGER, CHAR(10)) TO HR
```

- *Example 5:* Grant the EXECUTE privilege on method SET_SALARY of type EMPLOYEE to user JONES.

```
GRANT EXECUTE ON METHOD SET_SALARY FOR EMPLOYEE TO JONES
```

GRANT (schema privileges and authorities)

This form of the GRANT statement grants privileges and authorities on a schema.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

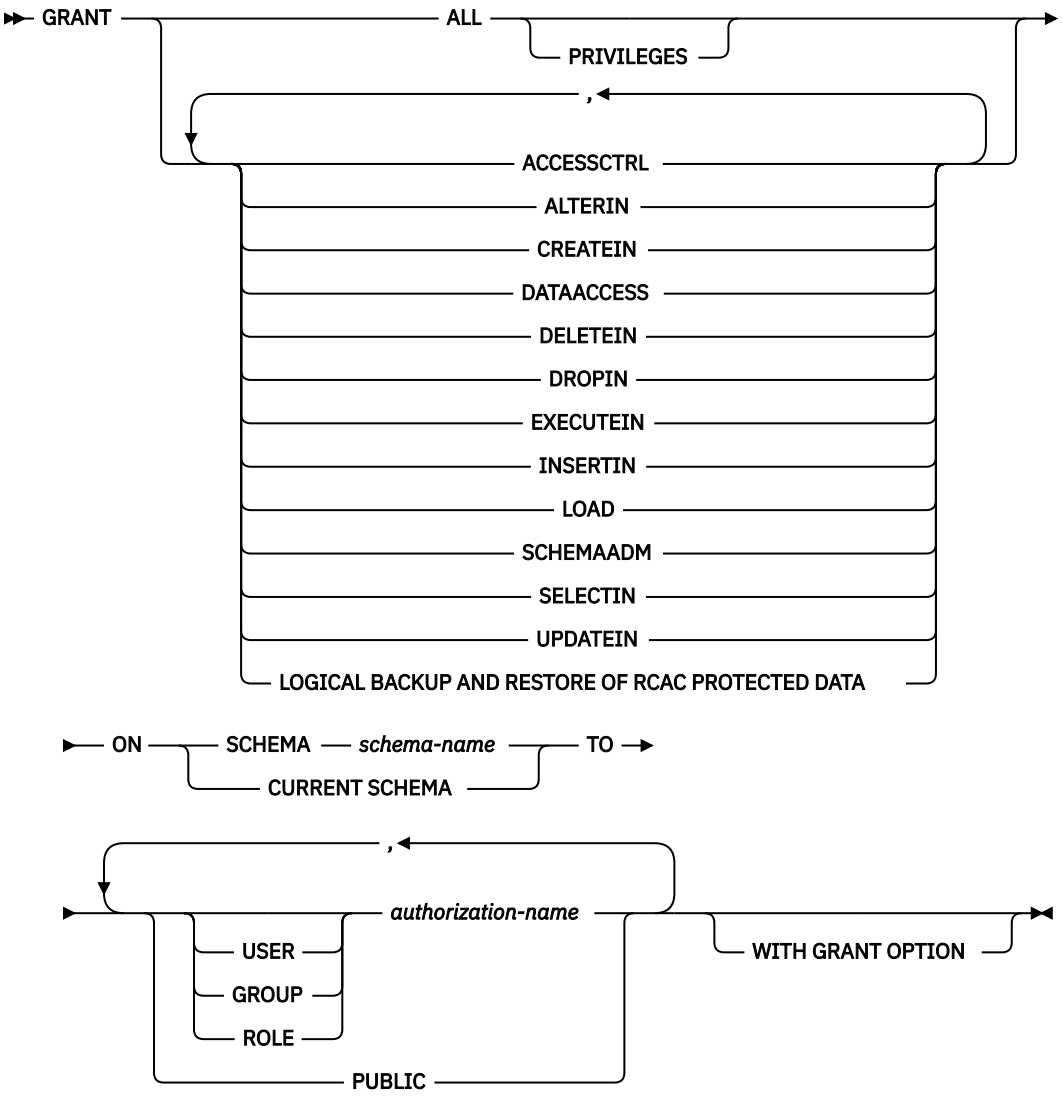
- The WITH GRANT OPTION for each identified privilege on *schema-name*
- ACCESSCTRL authority on *schema-name*
- ACCESSCTRL or SECADM authority

Exceptions:

- Only an authorization ID with ACCESSCTRL or SECADM can grant the following privileges on schema names starting with SYS:
  - SELECTIN privilege on SYSCAT, SYSFUN, SYSSTAT or any schema names starting with SYSIBM (SQLSTATE 42501).
  - SELECTIN, CREATEIN and DROPIN privileges on SYSPROC, SYSPUBLIC or SYSTOOLS schemas. Granting CREATEIN privilege allows the user to create a public alias. Granting DROPIN privilege allows the user to drop any public alias.

- No user can grant any other privileges or authorities on schema names starting with SYS (SQLSTATE 42501).
- Only a user with SECADM or database ACCESSCTRL authority can grant schema ACCESSCTRL authority.
- No schema authorities (SCHEMAADM, ACCESSCTRL, DATAACCESS, and LOAD) can be granted to PUBLIC directly or indirectly

Syntax



Description

ALL or ALL PRIVILEGES

Grants all of the following schema privileges on the schema that is named in the ON clause:

- ALTERIN
- CREATEIN
- DELETEIN
- DROPIN
- EXECUTEIN
- INSERTIN
- SELECTIN

- UPDATEIN

If ALL is not specified, one or more of the keywords in the list of privileges must be specified.

ACCESSCTRL
Grants the access control authority on the schema. The schema ACCESSCTRL authority allows the holder to:

- Grant and revoke the following privileges:
  - READ, WRITE on global variables defined in the schema
  - BIND privileges on packages defined in the schema
  - CONTROL privileges on packages and modules defined in the schema
  - USAGE privilege on XSR objects defined in the schema
- Grant and revoke all schema level privileges and authorities except for schema ACCESSCTRL itself.

The schema ACCESSCTRL authority cannot be granted to PUBLIC (SQLSTATE 42508).

For more information, see Schema access control authority (ACCESSCTRL).

ALTERIN
Grants the privilege to alter or comment on all objects in the schema. The owner of an explicitly created schema automatically receives ALTERIN privilege.

CREATEIN
Grants the privilege to create objects in the schema. Other authorities or privileges required to create the object (such as CREATETAB) are still required. The owner of an explicitly created schema automatically receives CREATEIN privilege. An implicitly created schema has CREATEIN privilege automatically granted to PUBLIC.

DATAACCESS
Grants the authority to access data in the schema. The schema DATAACCESS authority allows the holder to do the following:

- Select, insert, update, delete, and load data from tables or views defined in the schema
- Execute any package defined in the schema
- Execute any routine, except audit routines, defined in the schema

The schema DATAACCESS authority cannot be granted to PUBLIC (SQLSTATE 42508).

For more information, see Schema data access authority (DATAACCESS).

DELETEIN

Grants the privilege to delete data in the table objects in the schema. The owner of the schema (explicitly or implicitly created) does not automatically receive DELETEIN privilege.

DROPIN
Grants the privilege to drop all objects in the schema. The owner of an explicitly created schema automatically receives DROPIN privilege.

EXECUTEIN
Grants the privilege to execute all existing and future user-defined functions, methods, procedures, packages, or modules defined in the schema. The owner of the schema (explicitly or implicitly created) does not automatically receive EXECUTEIN privilege.

INSERTIN

Grants the privilege to insert rows and to run the IMPORT utility on all existing and future tables or views defined in the schema. The owner of the schema (explicitly or implicitly created) does not automatically receive INSERTIN privilege.

LOAD
Grants the authority to load in this schema. This authority gives a user the right to use the LOAD utility in this schema. SCHEMAADM has this authority by default. However, if a user only has schema

LOAD authority (not schema DATAACCESS), the user is also required to have table-level privileges. In addition to schema LOAD privilege, the user is required to have:

- INSERT privilege on the table or INSERTIN privilege on the schema of the table for LOAD with mode INSERT, TERMINATE (to terminate a previous LOAD INSERT), or RESTART (to restart a previous LOAD INSERT)
- INSERT and DELETE privilege on the table or INSERTIN and DELETEIN privilege on the schema of the table for LOAD with mode REPLACE, TERMINATE (to terminate a previous LOAD REPLACE), or RESTART (to restart a previous LOAD REPLACE)
- INSERT privilege on the exception table or INSERTIN privilege on the schema of the exception table, if such a table is used as part of LOAD

Schema LOAD authority cannot be granted to PUBLIC (SQLSTATE 42508).

For more information, see Schema load authority (LOAD).

SCHEMAADM

Grants the schema administrator authority. A schema administrator holds nearly all privileges on nearly all objects in the schema. The only exceptions are those privileges that are part of the access control, and schema data access.

SCHEMAADM authority cannot be granted to PUBLIC (SQLSTATE 42508).

For more information, see Schema administration authority (SCHEMAADM).

SELECTIN

Grants the privilege to select from all existing and future tables or views defined in the schema. The owner of the schema (explicitly or implicitly created) does not automatically receive SELECTIN privilege.

UPDATEIN

Grants the privilege to use the UPDATE statement on all existing and future tables or updatable views defined in the schema. The owner of the schema (explicitly or implicitly created) does not automatically receive UPDATEIN privilege.

LOGICAL BACKUP AND RESTORE OF RCAC PROTECTED DATA

Grants users the privilege to allow a schema-level **db_backup** and **db_restore** scripts to access RCAC protected data.

The schema **LOGICAL BACKUP AND RESTORE OF RCAC PROTECTED DATA** authority cannot be granted to PUBLIC (SQLSTATE 42508).

ON

SCHEMA *schema-name*

Specifies the name of the schema on which the authorities are to be granted. Authorities cannot be granted on any schema beginning with the SYS prefix (SQLSTATE 42501).

CURRENT SCHEMA

Specifies that the authorities will be granted on the schema described by the DB2® special register CURRENT SCHEMA. Authorities cannot be granted on any schema beginning with the SYS prefix (SQLSTATE 42501).

TO

Specifies to whom the privileges are granted.

USER

Specifies that the *authorization-name* identifies a user.

GROUP

Specifies that the *authorization-name* identifies a group name.

ROLE

Specifies that the *authorization-name* identifies a role name. The role name must exist at the current server (SQLSTATE 42704).

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Grants the privileges to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership".

WITH GRANT OPTION
Allows the specified *authorization-name*s to GRANT the privileges to others.

Rules

- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:
  - If the security plug-in in effect for the instance cannot determine the status of the *authorization-name*, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as ROLE in the database, and as either GROUP or USER according to the security plug-in in effect, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as both USER and GROUP, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as USER only, or if it is undefined, USER is assumed.
  - If the *authorization-name* is defined according to the security plug-in in effect as GROUP only, GROUP is assumed.
  - If the *authorization-name* is defined in the database as ROLE only, ROLE is assumed.
- In general, the GRANT statement will process the granting of privileges that the authorization ID of the statement is allowed to grant, returning a warning (SQLSTATE 01007) if one or more privileges was not granted. If no privileges were granted, an error is returned (SQLSTATE 42501). (If the package used for processing the statement was precompiled with LANGLEVEL set to SQL92E for MIA, a warning is returned (SQLSTATE 01007), unless the grantor has no privileges on the object of the grant operation.)

Notes

- **Grant on SYSPUBLIC:** You can grant privileges on the reserved schema SYSPUBLIC. Granting CREATEIN privilege allows you to create a public alias, and granting DROPIN privilege allows you to drop a public alias. Granting SELECTIN allows you to select from tables defined in the schema.
- The following authorities cannot be granted to the special group PUBLIC:
  - SCHEMAADM on the schema
  - ACCESSCTRL on the schema
  - DATAACCESS on the schema
  - LOAD on the schema

  Granting any of these authorities to a role that is granted to PUBLIC, either directly or indirectly, will fail (SQLSTATE 42508).
  - Role role-name is granted directly to PUBLIC if the following statement has been issued:

    ```
    GRANT ROLE role-name TO PUBLIC
    ```

  - Role role-name is granted indirectly to PUBLIC if the following statements have been issued:

    ```
    GRANT ROLE role-name TO ROLE role-name2
    ```

    ```
    GRANT ROLE role-name2 TO PUBLIC
    ```

- ***Privileges granted to a group***: An authority that is granted to a group is not used for authorization checking on:
  - Static DML statements in a package
  - A base table while processing a CREATE VIEW statement
  - A base table while processing a CREATE TABLE statement for a materialized query table
  - Create SQL routine
  - Create trigger
- WITH GRANT OPTION is ignored when granting schema authorities (SCHEMAADM, ACCESSCTRL, DATAACCESS, LOAD)

Examples

- *Example 1:* Grant user JSINGLETON to the ability to create objects in schema CORPDATA.

```
GRANT CREATEIN ON SCHEMA CORPDATA TO JSINGLETON
```

- *Example 2:* Grant user IHAKES the ability to create and drop objects in schema CORPDATA.

```
GRANT CREATEIN, DROPIN ON SCHEMA CORPDATA TO IHAKES
```

GRANT (security label)

This form of the GRANT statement grants a label-based access control (LBAC) security label to a user, group, or role for read access, write access, or for both read and write access.

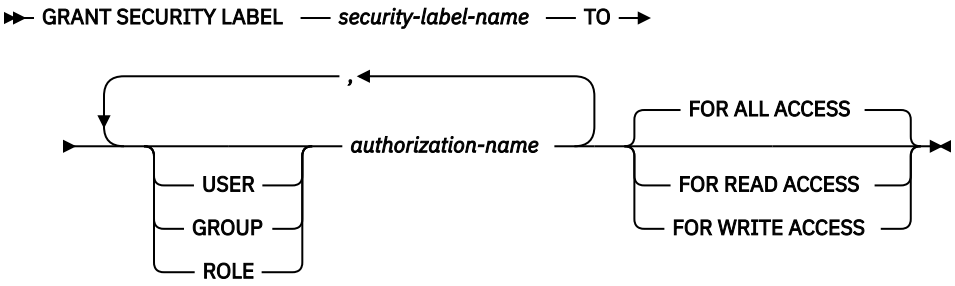
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include SECADM authority.

Syntax



Description

SECURITY LABEL *security-label-name*
Grants the security label *security-label-name*. The name must be qualified with a security policy (SQLSTATE 42704) and must identify a security label that exists at the current server (SQLSTATE 42704).

TO
Specifies to whom the specified security label is granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name. The role name must exist at the current server (SQLSTATE 42704).

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

FOR ALL ACCESS
Indicates that the security label is to be granted for both read access and write access.

FOR READ ACCESS
Indicates that the security label is to be granted for read access only.

FOR WRITE ACCESS
Indicates that the security label is to be granted for write access only.

Rules

- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:
  - If the security plug-in in effect for the instance cannot determine the status of the *authorization-name*, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as ROLE in the database, and as either GROUP or USER according to the security plug-in in effect, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as both USER and GROUP, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as USER only, or if it is undefined, USER is assumed.
  - If the *authorization-name* is defined according to the security plug-in in effect as GROUP only, GROUP is assumed.
  - If the *authorization-name* is defined in the database as ROLE only, ROLE is assumed.
- For any given security policy, an *authorization-name* can be granted at most one security label from that policy for read access and one for write access. If the grantee already holds a security label for the type of access (read or write) indicated and that is part of the security policy that qualifies *security-label-name*, an error is returned (SQLSTATE 428GR).
- If the security policy is not defined to consider access through groups or roles, any security label granted to a group or role is ignored when access is attempted.
- If an *authorization-name* holds different security labels for read access and write access, the security labels must meet the following criteria (SQLSTATE 428GQ):
  - If any component in the security labels is of type ARRAY then the value for that component must be the same in both security labels.
  - If any component in the security labels is of type SET then every element in the value for that component in the write security label must also be part of the value for that component in the read security label.
  - If any component in the security labels is of type TREE then every element in the value for that component in the write security label must be the same as or a descendent of one of the elements in the value for that same component in the read security label.

Notes

- By default when a security policy is created, only security labels granted to an individual user are considered. To have groups or roles considered for the security policy, you must issue the ALTER

SECURITY POLICY statement and specify USE GROUP AUTHORIZATION or USE ROLE AUTHORIZATION as applicable.

Example

The following statement grants two security labels to user GUYLAINE. The security label EMPLOYEESECLABELREAD is granted for read access and the security label EMPLOYEESECLABELWRITE is granted for write access. Both security labels are part of the security policy DATA_ACCESS.

```
GRANT SECURITY LABEL DATA_ACCESS.EMPLOYEESECLABELREAD
  TO USER GUYLAINE FOR READ ACCESS

GRANT SECURITY LABEL DATA_ACCESS.EMPLOYEESECLABELWRITE
  TO USER GUYLAINE FOR WRITE ACCESS
```

The same user is now granted the security label BEGINNER for both read and write access. This does not cause an error, because BEGINNER is part of the security policy CLASSPOLICY, and the security labels already held are part of the security policy DATA_ACCESS.

```
GRANT SECURITY LABEL CLASSPOLICY.BEGINNER
  TO USER GUYLAINE FOR ALL ACCESS
```

GRANT (sequence privileges)

This form of the GRANT statement grants privileges on a sequence.

Invocation

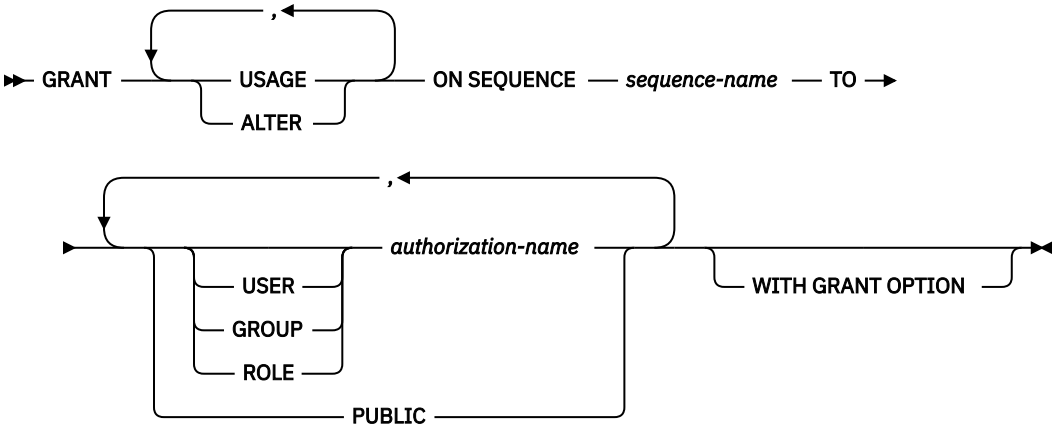
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- The WITH GRANT OPTION for each identified privilege on *sequence-name*
- ACCESSCTRL authority on the schema containing the sequence-name
- ACCESSCTRL or SECADM authority

Syntax



Description

USAGE
Grants the privilege to reference a sequence using *nextval-expression* or *prevval-expression*.

ALTER
Grants the privilege to alter sequence properties using the ALTER SEQUENCE statement.

ON SEQUENCE *sequence-name*
Identifies the sequence on which the specified privileges are to be granted. The sequence name, including an implicit or explicit schema qualifier, must uniquely identify an existing sequence at the current server. If no sequence by this name exists, an error (SQLSTATE 42704) is returned.

TO
Specifies to whom the specified privileges are granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name. The role name must exist at the current server (SQLSTATE 42704).

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

PUBLIC
Grants the specified privileges to a set of users (authorization IDs).

WITH GRANT OPTION
Allows the specified *authorization-name* to grant the specified privileges to others.

If the WITH GRANT OPTION is omitted, the specified *authorization-name* can only grant the specified privileges to others if they:

- have SYSADM or DBADM authority or
- received the ability to grant the specified privileges from some other source.

Rules

- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:
  - If the security plug-in in effect for the instance cannot determine the status of the *authorization-name*, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as ROLE in the database, and as either GROUP or USER according to the security plug-in in effect, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as both USER and GROUP, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as USER only, or if it is undefined, USER is assumed.
  - If the *authorization-name* is defined according to the security plug-in in effect as GROUP only, GROUP is assumed.
  - If the *authorization-name* is defined in the database as ROLE only, ROLE is assumed.
- In general, the GRANT statement will process the granting of privileges that the authorization ID of the statement is allowed to grant, returning a warning (SQLSTATE 01007) if one or more privileges is not granted. If no privileges are granted, an error is returned (SQLSTATE 42501). (If the package used for processing the statement was precompiled with LANGLEVEL set to SQL92E or MIA, a warning is returned (SQLSTATE 01007), unless the grantor has no privileges on the object of the grant operation.)

Notes

- ***Privileges granted to a group***: A privilege that is granted to a group is not used for authorization checking on:
  - Static DML statements in a package
  - A base table while processing a CREATE VIEW statement
  - A base table while processing a CREATE TABLE statement for a materialized query table
  - Create SQL routine
  - Create trigger

Examples

- *Example 1:* Grant any user the USAGE privilege on a sequence called ORG_SEQ.

```
GRANT USAGE ON SEQUENCE ORG_SEQ TO PUBLIC
```

- *Example 2:* Grant user BOBBY the ability to alter a sequence called GENERATE_ID, and to grant this privilege to others.

```
GRANT ALTER ON SEQUENCE GENERATE_ID TO BOBBY WITH GRANT OPTION
```

GRANT (server privileges)

This form of the GRANT statement grants the privilege to access and use a specified data source in pass-through mode.

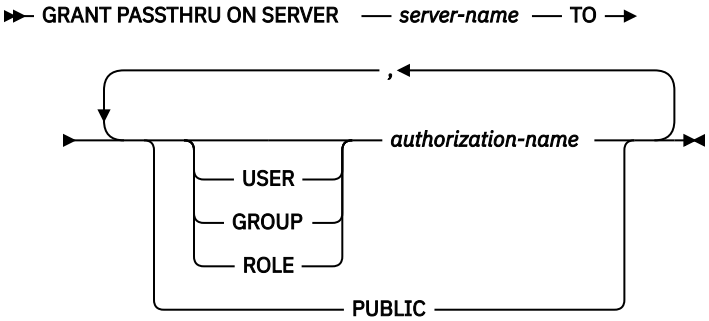
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include ACCESSCTRL or SECADM authority.

Syntax



Description

server-name
Names the data source for which the privilege to use in pass-through mode is being granted. *server-name* must identify a data source that is described in the catalog.

TO

Specifies to whom the privilege is granted.

USER

Specifies that the *authorization-name* identifies a user.

GROUP

Specifies that the *authorization-name* identifies a group name.

ROLE

Specifies that the *authorization-name* identifies a role name. The role name must exist at the current server (SQLSTATE 42704).

authorization-name,...

Lists the authorization IDs of one or more users, groups, or roles.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC

Grants to a set of users (authorization IDs) the privilege to pass through to *server-name*. For more information, see "Authorization, privileges and object ownership".

Rules

- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:
  - If the security plug-in in effect for the instance cannot determine the status of the *authorization-name*, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as ROLE in the database, and as either GROUP or USER according to the security plug-in in effect, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as both USER and GROUP, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as USER only, or if it is undefined, USER is assumed.
  - If the *authorization-name* is defined according to the security plug-in in effect as GROUP only, GROUP is assumed.
  - If the *authorization-name* is defined in the database as ROLE only, ROLE is assumed.

Examples

- *Example 1:* Give R. Smith and J. Jones the privilege to pass through to data source SERVALL. Their authorization IDs are RSMITH and JJONES.

```
GRANT PASSTHRU ON SERVER SERVALL
  TO USER RSMITH,
  USER JJONES
```

- *Example 2:* Grant the privilege to pass through to data source EASTWING to a group whose authorization ID is D024. There is a user whose authorization ID is also D024.

```
GRANT PASSTHRU ON SERVER EASTWING TO GROUP D024
```

The GROUP keyword must be specified; otherwise, an error will occur because D024 is a user's ID as well as the specified group's ID (SQLSTATE 56092). Any member of group D024 will be allowed to pass through to EASTWING. Therefore, if user D024 belongs to the group, this user will be able to pass through to EASTWING.

GRANT (SETSESSIONUSER privilege)

This form of the GRANT statement grants the SETSESSIONUSER privilege to one or more authorization IDs. The privilege allows the holder to use the SET SESSION AUTHORIZATION statement to set the session authorization to one of a set of specified authorization IDs.

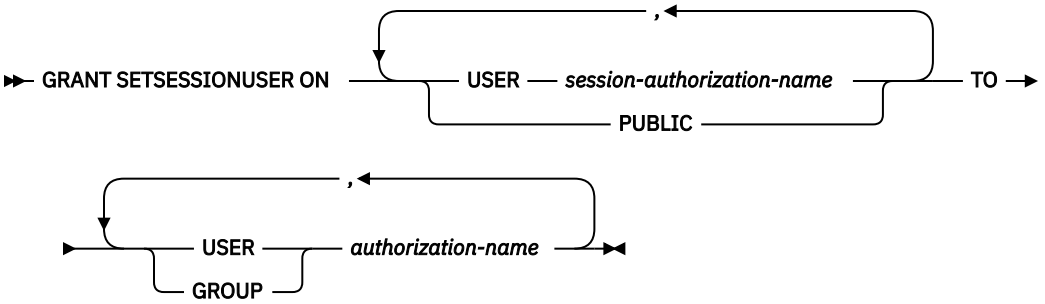
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include SECADM authority.

Syntax



Description

SETSESSIONUSER ON
Grants the privilege to assume the identity of a new authorization ID.

USER *session-authorization-name*
Specifies the authorization ID that the *authorization-name* will be able to assume, using the SET SESSION AUTHORIZATION statement. The *session-authorization-name* must identify a user, not a group.

PUBLIC
Specifies that the grantee will be able to assume any valid authorization ID, using the SET SESSION AUTHORIZATION statement.

TO
Specifies to whom the privilege is granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group.

authorization-name,...
Lists the authorization IDs of one or more users or groups.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

Rules

- For each *authorization-name* specified, if neither USER nor GROUP is specified, then:

- If the security plug-in in effect for the instance cannot determine the status of the *authorization-name*, an error is returned (SQLSTATE 56092).
- If the *authorization-name* is defined according to the security plug-in in effect as both USER and GROUP, an error is returned (SQLSTATE 56092).
- If the *authorization-name* is defined according to the security plug-in in effect as USER only, or if it is undefined, USER is assumed.
- If the *authorization-name* is defined according to the security plug-in in effect as GROUP only, GROUP is assumed.

Notes

- ***Privileges granted to a group***: A privilege that is granted to a group is not used for authorization checking on:
  - Static DML statements in a package
  - A base table while processing a CREATE VIEW statement
  - A base table while processing a CREATE TABLE statement for a materialized query table
  - Create SQL routine
  - Create trigger

Examples

- *Example 1:* The following statement grants user PAUL the ability to set the session authorization to user WALID and therefore to execute statements as WALID.

```
GRANT SETSESSIONUSER ON USER WALID
  TO USER PAUL
```

- *Example 2:* The following statement grants user GUYLAINE the ability to set the session authorization to user BOBBY. It also grants her the ability to set the session authorization to users RICK and KEVIN.

```
GRANT SETSESSIONUSER ON USER BOBBY, USER RICK, USER KEVIN
  TO USER GUYLAINE
```

- *Example 3:* The following statement grants user WALID and everyone in the groups ADMINS and ACCTG the ability to set the session authorization to any user.

```
GRANT SETSESSIONUSER ON PUBLIC TO USER WALID, GROUP ADMINS, ACCTG
```

GRANT (table space privileges)

This form of the GRANT statement grants privileges on a table space.

Invocation

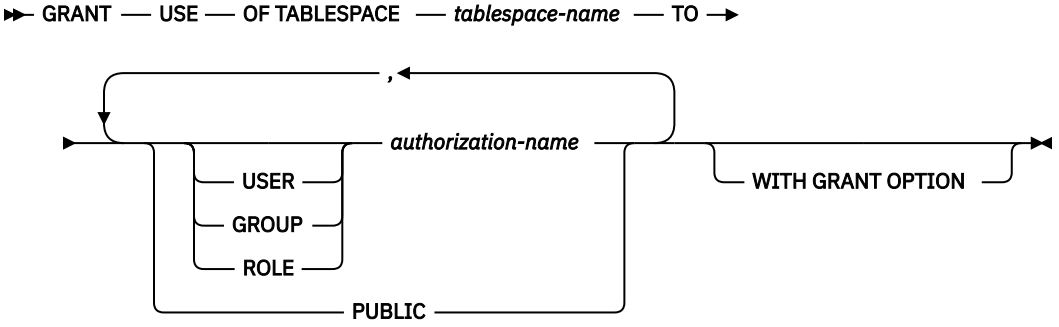
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- The WITH GRANT OPTION for use of the table space
- ACCESSCTRL, SECADM, SYSADM, or SYSCTRL authority

Syntax



Description

USE
Grants the privilege to specify or default to the table space when creating a table. The creator of a table space automatically receives USE privilege with grant option.

OF TABLESPACE *tablespace-name*
Identifies the table space on which the USE privilege is to be granted. The table space cannot be SYSCATSPACE (SQLSTATE 42838) or a system temporary table space (SQLSTATE 42809).

TO
Specifies to whom the USE privilege is granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name. The role name must exist at the current server (SQLSTATE 42704).

authorization-name
Lists the authorization IDs of one or more users, groups, or roles.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Grants the USE privilege to a set of users (authorization IDs).

WITH GRANT OPTION
Allows the specified *authorization-name* to GRANT the USE privilege to others.

Rules

- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:
  - If the security plug-in in effect for the instance cannot determine the status of the *authorization-name*, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as ROLE in the database, and as either GROUP or USER according to the security plug-in in effect, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as both USER and GROUP, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as USER only, or if it is undefined, USER is assumed.
  - If the *authorization-name* is defined according to the security plug-in in effect as GROUP only, GROUP is assumed.

– If the *authorization-name* is defined in the database as ROLE only, ROLE is assumed.

Example

Grant user BOBBY the ability to create tables in table space PLANS and to grant this privilege to others.

```
GRANT USE OF TABLESPACE PLANS TO BOBBY WITH GRANT OPTION
```

GRANT (table, view, or nickname privileges)

This form of the GRANT statement grants privileges on a table, view, or nickname.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

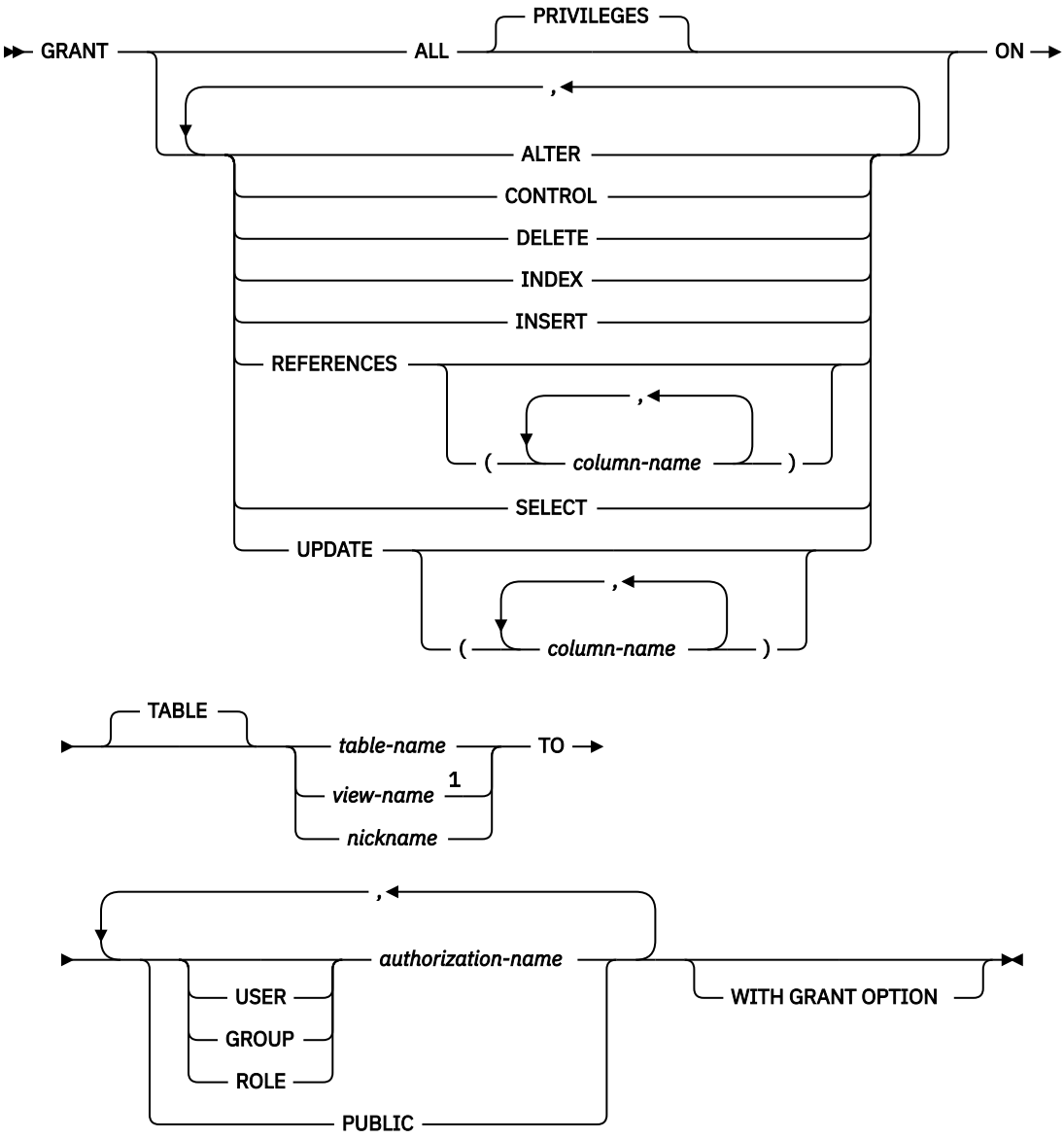
Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- CONTROL privilege on the referenced table, view, or nickname
- The WITH GRANT OPTION for each identified privilege. If ALL is specified, the authorization ID must have some grantable privilege on the identified table, view, or nickname
- ACCESSCTRL authority on the schema containing the identified table, view, or nickname
- ACCESSCTRL or SECADM authority

ACCESSCTRL authority on the schema, ACCESSCTRL authority on the database, or SECADM authority is required to grant the CONTROL privilege. ACCESSCTRL authority on the database or SECADM authority is required to grant privileges on catalog tables and views.

Syntax



Notes:

[1] ALTER, INDEX, and REFERENCES privileges are not applicable to views.

Description

ALL or ALL PRIVILEGES

Grants all the appropriate privileges, except CONTROL, on the base table, view, or nickname named in the ON clause.

If the authorization ID of the statement has CONTROL privilege on the table, view, or nickname, or ACCESSCTRL or SECADM authority, then all the privileges applicable to the object (except CONTROL) are granted. Otherwise, the privileges granted are all those grantable privileges that the authorization ID of the statement has on the identified table, view, or nickname.

If ALL is not specified, one or more of the keywords in the list of privileges must be specified.

ALTER

Grants the privilege to:

- Add columns to a base table definition.

- Create or drop a primary key or unique constraint on a base table.
- Create or drop a foreign key on a base table.

  The REFERENCES privilege on each column of the parent table is also required.
- Create or drop a check constraint on a base table.
- Create a trigger on a base table.
- Add, reset, or drop a column option for a nickname.
- Change a nickname column name or data type.
- Add or change a comment on a base table or a nickname.

CONTROL
Grants:

- All of the appropriate privileges in the list, that is:
  - ALTER, CONTROL, DELETE, INSERT, INDEX, REFERENCES, SELECT, and UPDATE to base tables
  - CONTROL, DELETE, INSERT, SELECT, and UPDATE to views
  - ALTER, CONTROL, INDEX, and REFERENCES to nicknames
- The ability to grant the previously mentioned privileges (except for CONTROL) to others.
- The ability to drop the base table, view, or nickname.

  This ability cannot be extended to others on the basis of holding CONTROL privilege. The only way that it can be extended is by granting the CONTROL privilege itself and that can only be done by an authorization ID with ACCESSCTRL or SECADM authority.
- The ability to execute the RUNSTATS utility on the table and indexes.
- The ability to execute the REORG utility on the table.
- The ability to issue the SET INTEGRITY statement against a base table, materialized query table, or staging table.

The definer of a base table, materialized query table, staging table, or nickname automatically receives the CONTROL privilege.

The definer of a view automatically receives the CONTROL privilege if the definer holds the CONTROL privilege on all tables, views, and nicknames identified in the fullselect.

DELETE
Grants the privilege to delete rows from the table or updatable view.

INDEX
Grants the privilege to create an index on a table, or an index specification on a nickname. This privilege cannot be granted on a view. The creator of an index or index specification automatically has the CONTROL privilege on the index or index specification (authorizing the creator to drop the index or index specification). In addition, the creator retains the CONTROL privilege even if the INDEX privilege is revoked.

INSERT
Grants the privilege to insert rows into the table or updatable view and to run the IMPORT utility.

REFERENCES
Grants the privilege to create and drop a foreign key referencing the table as the parent.

If the authorization ID of the statement has one of:

- ACCESSCTRL or SECADM authority
- CONTROL privilege on the table
- REFERENCES WITH GRANT OPTION on the table

then the grantee(s) can create referential constraints using all columns of the table as parent key, even those added later using the ALTER TABLE statement. Otherwise, the privileges granted are all

those grantable column REFERENCES privileges that the authorization ID of the statement has on the identified table.

The privilege can be granted on a nickname, although foreign keys cannot be defined to reference nicknames.

REFERENCES (*column-name*,...)
Grants the privilege to create and drop a foreign key using only those columns specified in the column list as a parent key. Each *column-name* must be an unqualified name that identifies a column of the table identified in the ON clause. Column level REFERENCES privilege cannot be granted on typed tables, typed views, or nicknames (SQLSTATE 42997).

SELECT
Grants the privilege to:

- Retrieve rows from the table or view.
- Create views on the table.
- Run the EXPORT utility against the table or view.

UPDATE
Grants the privilege to use the UPDATE statement on the table or updatable view identified in the ON clause.

If the authorization ID of the statement has one of:

- ACCESSCTRL or SECADM authority
- CONTROL privilege on the table or view
- UPDATE WITH GRANT OPTION on the table or view

then the grantee(s) can update all updatable columns of the table or view on which the grantor has with grant privilege as well as those columns added later using the ALTER TABLE statement. Otherwise, the privileges granted are all those grantable column UPDATE privileges that the authorization ID of the statement has on the identified table or view.

UPDATE (*column-name*,...)
Grants the privilege to use the UPDATE statement to update only those columns specified in the column list. Each *column-name* must be an unqualified name that identifies a column of the table or view identified in the ON clause. Column level UPDATE privilege cannot be granted on typed tables, typed views, or nicknames (SQLSTATE 42997).

ON TABLE *table-name* or *view-name* or *nickname*
Specifies the table, view, or nickname on which privileges are to be granted.

No privileges may be granted on an inoperative view or an inoperative materialized query table (SQLSTATE 51024). No privileges may be granted on a declared temporary table (SQLSTATE 42995).

TO
Specifies to whom the privileges are granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name. The role name must exist at the current server (SQLSTATE 42704).

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

A privilege that is granted to a group is not used for authorization checking:

- On static DML statements in a package
- On a base table while processing a CREATE VIEW statement

- On a base table while processing a CREATE TABLE statement for a materialized query table

Table privileges granted to groups only apply to statements that are dynamically prepared. For example, if the INSERT privilege on the PROJECT table has been granted to group D204 but not UBIQUITY (a member of D204) UBIQUITY could issue the statement:

```
EXEC SQL EXECUTE IMMEDIATE :INSERT_STRING;
```

where the content of the string is:

```
INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP)
VALUES ('AD3114', 'TOOL PROGRAMMING', 'D21', '000260');
```

but could not precompile or bind a program with the statement:

```
EXEC SQL INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP)
VALUES ('AD3114', 'TOOL PROGRAMMING', 'D21', '000260');
```

PUBLIC
Grants the privileges to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership". (Previous restrictions on the use of privileges granted to PUBLIC for static SQL statements and the CREATE VIEW statement have been removed.)

WITH GRANT OPTION
Allows the specified *authorization-name*s to GRANT the privileges to others.

If the specified privileges include CONTROL, the WITH GRANT OPTION applies to all the applicable privileges except for CONTROL (SQLSTATE 01516).

Rules

- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:
  - If the security plug-in in effect for the instance cannot determine the status of the *authorization-name*, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as ROLE in the database, and as either GROUP or USER according to the security plug-in in effect, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as both USER and GROUP, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined according to the security plug-in in effect as USER only, or if it is undefined, USER is assumed.
  - If the *authorization-name* is defined according to the security plug-in in effect as GROUP only, GROUP is assumed.
  - If the *authorization-name* is defined in the database as ROLE only, ROLE is assumed.
- In general, the GRANT statement will process the granting of privileges that the authorization ID of the statement is allowed to grant, returning a warning (SQLSTATE 01007) if one or more privileges was not granted. If no privileges were granted, an error is returned (SQLSTATE 42501). (If the package used for processing the statement was precompiled with LANGLEVEL set to SQL92E or MIA, a warning is returned (SQLSTATE 01007), unless the grantor has no privileges on the object of the grant operation.) If CONTROL privilege is specified, privileges will only be granted if the authorization ID of the statement has ACCESSCTRL or SECADM authority (SQLSTATE 42501).

Notes

- Privileges may be granted independently at every level of a table hierarchy. A user with a privilege on a supertable may affect the subtables. For example, an update specifying the supertable *T* may show up as a change to a row in the subtable *S* of *T* done by a user with UPDATE privilege on *T* but without UPDATE privilege on *S*. A user can only operate directly on the subtable if the necessary privilege is held on the subtable.

- Granting nickname privileges has no effect on data source object (table or view) privileges. Typically, data source privileges are required for the table or view that a nickname references when attempting to retrieve data.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. The following syntax is tolerated and ignored:
  - PUBLIC AT ALL LOCATIONS
- WITH GRANT OPTION is ignored when granting table or view (CONTROL) privilege

Examples

1. Grant all privileges on the table WESTERN_CR to PUBLIC.

```
GRANT ALL ON WESTERN_CR
   TO PUBLIC
```

2. Grant the appropriate privileges on the CALENDAR table so that users PHIL and CLAIRE can read it and insert new entries into it. Do not allow them to change or remove any existing entries.

```
GRANT SELECT, INSERT ON CALENDAR
   TO USER  PHIL, USER CLAIRE
```

3. Grant all privileges on the COUNCIL table to user FRANK and the ability to extend all privileges to others.

```
GRANT ALL ON COUNCIL
   TO USER FRANK WITH GRANT OPTION
```

4. GRANT SELECT privilege on table CORPDATA.EMPLOYEE to a user named JOHN. There is a user called JOHN and no group called JOHN.

```
GRANT SELECT ON CORPDATA.EMPLOYEE TO JOHN
```

   or

```
GRANT SELECT
  ON CORPDATA.EMPLOYEE TO USER JOHN
```

5. GRANT SELECT privilege on table CORPDATA.EMPLOYEE to a group named JOHN. There is a group called JOHN and no user called JOHN.

```
GRANT SELECT ON CORPDATA.EMPLOYEE TO JOHN
```

   or

```
GRANT SELECT ON CORPDATA.EMPLOYEE TO GROUP JOHN
```

6. GRANT INSERT and SELECT on table T1 to both a group named D024 and a user named D024.

```
GRANT INSERT, SELECT ON TABLE T1
  TO GROUP D024, USER D024
```

   In this case, both the members of the D024 group and the user D024 would be allowed to INSERT into and SELECT from the table T1. Also, there would be two rows added to the SYSCAT.TABAUTH catalog view.

7. GRANT INSERT, SELECT, and CONTROL on the CALENDAR table to user FRANK. FRANK must be able to pass the privileges on to others.

```
GRANT CONTROL ON TABLE CALENDAR
  TO FRANK WITH GRANT OPTION
```

   The result of this statement is a warning (SQLSTATE 01516) that CONTROL was not given the WITH GRANT OPTION. Frank now has the ability to grant any privilege on CALENDAR including INSERT

and SELECT as required. FRANK cannot grant CONTROL on CALENDAR to other users unless he has ACCESSCTRL or SECADM authority.

8. User JON created a nickname for an Oracle table that had no index. The nickname is ORAREM1. Later, the Oracle DBA defined an index for this table. User SHAWN now wants Db2 to know that this index exists, so that the optimizer can devise strategies to access the table more efficiently. SHAWN can inform Db2 of the index by creating an index specification for ORAREM1. Give SHAWN the index privilege on this nickname, so that he can create the index specification.

```
GRANT INDEX ON NICKNAME ORAREM1
  TO USER SHAWN
```

GRANT (workload privileges)

This form of the GRANT statement grants the USAGE privilege on a workload.

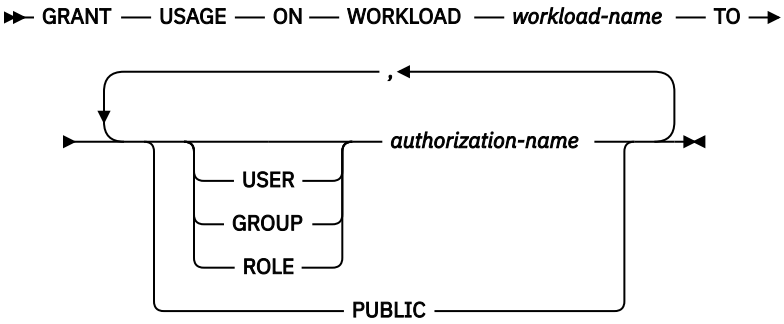
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include ACCESSCTRL, SECADM, or WLMADM authority.

Syntax



Description

USAGE
Grants the privilege to use a workload. Units of work that are submitted by a user will only be mapped to a workload on which the user has USAGE privilege. A user with SYSADM or DBADM authority automatically has USAGE privilege on any workload that exists at the current server.

ON WORKLOAD *workload-name*
Identifies the workload on which the USAGE privilege is to be granted. This is a one-part name. The *workload-name* must identify a workload that exists at the current server (SQLSTATE 42704). The name cannot be 'SYSDEFAULTADMWORKLOAD' (SQLSTATE 42832).

TO
Specifies to whom the USAGE privilege is granted.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group.

ROLE
Specifies that the *authorization-name* identifies an existing role at the current server (SQLSTATE 42704).

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles. The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Grants the USAGE privilege to a set of users (authorization IDs).

Rules

- For each *authorization-name* specified, if none of the keywords USER, GROUP, or ROLE is specified:
  - If the security plug-in in effect for the instance cannot determine the status of the *authorization-name*, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as ROLE in the database and as either GROUP or USER in the operating system, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as both USER and GROUP according to the security plug-in in effect, an error is returned (SQLSTATE 56092).
  - If the *authorization-name* is defined as USER only according to the security plug-in in effect, or if it is undefined, USER is assumed.
  - If the *authorization-name* is defined as GROUP only according to the security plug-in in effect, GROUP is assumed.
  - If the *authorization-name* is defined in the database as ROLE only, ROLE is assumed.

Notes

- The GRANT statement does not take effect until it is committed, even for the connection that issues the statement.
- If the database is created with the RESTRICT option, the USAGE privilege of the default user workload, SYSDEFAULTUSERWORKLOAD, must be granted explicitly by a user that has DBADM authority. If the database is created without the RESTRICT option, the USAGE privilege of SYSDEFAULTUSERWORKLOAD is granted to PUBLIC at database creation time.

Example

Grant user LISA the ability to use the workload CAMPAIGN.

```
GRANT USAGE ON WORKLOAD CAMPAIGN TO USER LISA
```

GRANT (XSR object privileges)

This form of the GRANT statement grants USAGE privilege on an XSR object.

Invocation

The GRANT statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if the DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

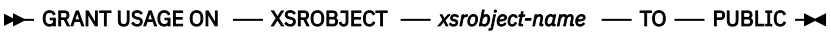
Authorization

One of the following authorities is required:

- ACCESSCTRL or SECADM authority
- ACCESSCTRL authority on the schema containing the XSR object

- Owner of the XSR object, as recorded in the OWNER column of the SYSCAT.XSROBJECTS catalog view

Syntax



Description

ON XSROBJECT *xsrobject-name*
This name identifies the XSR object for which the USAGE privilege is granted. The *xsrobject-name*, including the implicit or explicit schema qualifier, must uniquely identify an existing XSR object at the current server. If no XSR object by this name exists, an error is returned (SQLSTATE 42704).

TO PUBLIC
Grants the USAGE privilege to a set of users (authorization IDs).

Example

Grant every user the usage privilege on the XML schema MYSCHEMA:

```
GRANT USAGE ON XSROBJECT MYSCHEMA TO PUBLIC
```

IF

The IF statement selects an execution path based on the evaluation of a condition.

Invocation

This statement can be embedded in an:

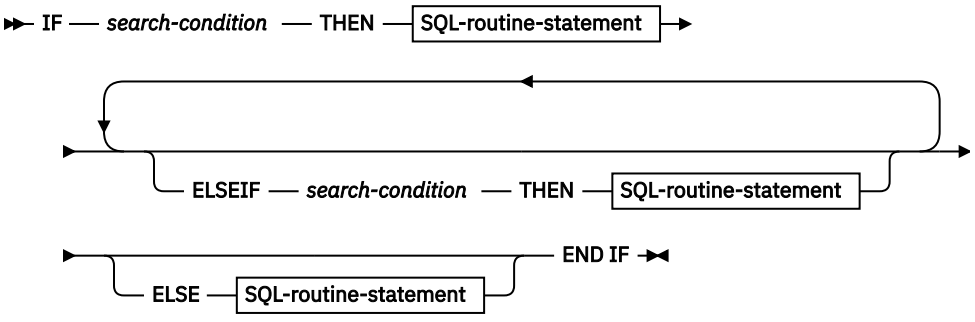
- SQL procedure definition
- Compound SQL (compiled) statement
- Compound SQL (inlined) statement

The compound statements can be embedded in an SQL procedure definition, SQL function definition, or SQL trigger definition. It is not an executable statement and cannot be dynamically prepared.

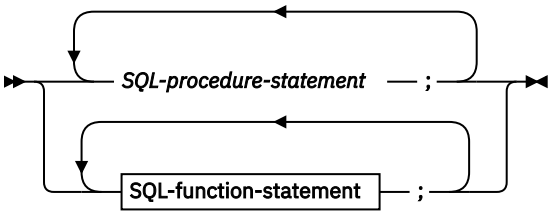
Authorization

Group privileges are not considered because this statement cannot be dynamically prepared.

Syntax



SQL-routine-statement



Description

search-condition
Specifies the condition for which an SQL statement should be invoked. If the condition is unknown or false, processing continues to the next search condition, until either a condition is true or processing reaches the ELSE clause.

SQL-procedure-statement
Specifies the statement to be invoked if the preceding *search-condition* is true. *SQL-procedure-statement* is only applicable when in the context of an SQL procedure or a compound SQL (compiled) statement. See *SQL-procedure-statement* in "Compound SQL (compiled)" statement.

SQL-function-statement
Specifies the statement to be invoked if the preceding *search-condition* is true. *SQL-function-statement* is only applicable when in the context of a compound SQL (inlined) statement, an SQL trigger, an SQL function, or an SQL method. See *SQL-function-statement* in "FOR".

Example

The following SQL procedure accepts two IN parameters: an employee number *employee_number* and an employee rating *rating*. Depending on the value of *rating*, the `employee` table is updated with new values in the `salary` and `bonus` columns.

```
CREATE PROCEDURE UPDATE_SALARY_IF
  (IN employee_number CHAR(6), INOUT rating SMALLINT)
  LANGUAGE SQL
  BEGIN
    DECLARE not_found CONDITION FOR SQLSTATE '02000';
    DECLARE EXIT HANDLER FOR not_found
      SET rating = -1;
    IF rating = 1
      THEN UPDATE employee
      SET salary = salary * 1.10, bonus = 1000
      WHERE empno = employee_number;
    ELSEIF rating = 2
      THEN UPDATE employee
      SET salary = salary * 1.05, bonus = 500
      WHERE empno = employee_number;
    ELSE UPDATE employee
      SET salary = salary * 1.03, bonus = 0
      WHERE empno = employee_number;
    END IF;
  END
```

INCLUDE

The INCLUDE statement inserts declarations into a source program.

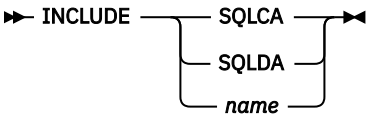
Invocation

This statement can only be embedded in an application program. It is not an executable statement.

Authorization

None required.

Syntax



Description

SQLCA
Indicates the description of an SQL communication area (SQLCA) is to be included.

SQLDA
Indicates the description of an SQL descriptor area (SQLDA) is to be included.

name
Identifies an external file containing text that is to be included in the source program being precompiled. It can be an SQL identifier without a file name extension or a literal enclosed by single quotation marks (' '). An SQL identifier assumes the filename extension of the source file being precompiled. If a file name extension is not provided by a literal enclosed by quotation marks, none is assumed.

Notes

- When a program is precompiled, the INCLUDE statement is replaced by source statements. Thus, the INCLUDE statement should be specified at a point in the program such that the resulting source statements are acceptable to the compiler.
- The external source file must be written in the host language specified by *name*. If it is greater than 18 bytes or contains characters that are not allowed in an SQL identifier, it must be enclosed by single quotation marks. INCLUDE *name* statements may be nested though not cyclical (for example, if A and B are modules and A contains an INCLUDE *name* statement, then it is not valid for A to call B and then B to call A).
- When the LANGLEVEL precompile option is specified with the SQL92E value, INCLUDE SQLCA should not be specified. SQLSTATE and SQLCODE variables may be defined within the host variable declare section.

Example

Include an SQLCA in a C program.

```
EXEC SQL INCLUDE SQLCA;

EXEC SQL DECLARE C1 CURSOR FOR
  SELECT DEPTNO, DEPTNAME, MGRNO FROM TDEPT
    WHERE ADMRDEPT = 'A00';

EXEC SQL OPEN C1;

while (SQLCODE==0) {
  EXEC SQL FETCH C1 INTO :dnum, :dname, :mnum;

(Print results)

}

EXEC SQL CLOSE C1;
```

INSERT

The INSERT statement inserts rows into a table, nickname, or view, or the underlying tables, nicknames, or views of the specified fullselect.

Inserting a row into a nickname inserts the row into the data source object to which the nickname refers. Inserting a row into a view also inserts the row into the table on which the view is based, if no INSTEAD OF trigger is defined for the insert operation on this view. If such a trigger is defined, the trigger will be executed instead.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- INSERT privilege on the target table, view, or nickname
- CONTROL privilege on the target table, view, or nickname
- INSERTIN privilege on the schema containing the target table, view, or nickname
- DATAACCESS authority on the schema containing the target table, view, or nickname
- DATAACCESS authority

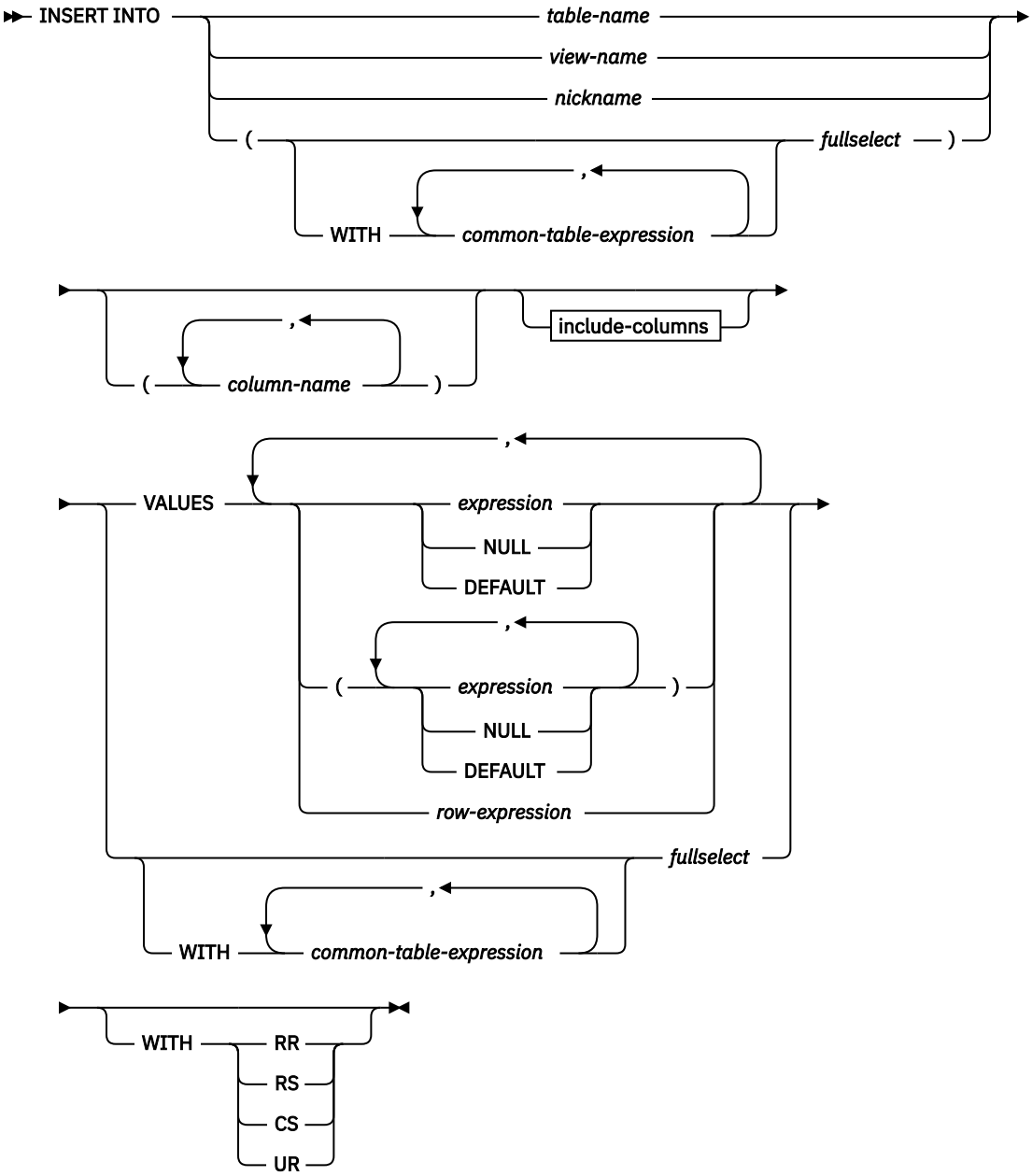
In addition, for each table, view, or nickname referenced in any fullselect used in the INSERT statement, the privileges held by the authorization ID of the statement must include at least one of the following authorities:

- SELECT privilege
- CONTROL privilege
- SELECTIN privilege on the schema containing the table, view or nickname
- DATAACCESS authority on the schema containing the table, view, or nickname
- DATAACCESS authority

GROUP privileges are not checked for static INSERT statements.

If the target of the insert operation is a nickname, the privileges on the object at the data source are not considered until the statement is executed at the data source. At this time, the authorization ID that is used to connect to the data source must have the privileges required for the operation on the object at the data source. The authorization ID of the statement can be mapped to a different authorization ID at the data source.

Syntax



include-columns

Description

INTO *table-name*, *view-name*, *nickname*, or *(fullselect)*

Identifies the object of the insert operation. The name must identify one of the following objects:

- A table, view or nickname that exists at the application server
- A table or view at a remote server specified using a remote-object-name

The object must not be a catalog table, a system-maintained materialized query table, a view of a catalog table, or a read-only view, unless an INSTEAD OF trigger is defined for the insert operation

on the subject view. Rows inserted into a nickname are placed in the data source object to which the nickname refers.

If the object of the insert operation is a fullselect, the fullselect must be insertable, as defined in the "Insertable views" Notes item in the description of the CREATE VIEW statement.

If the object of the insert operation is a nickname, the extended indicator variable values of DEFAULT and UNASSIGNED must not be used (SQLSTATE 22539).

If no INSTEAD OF trigger exists for the insert operation on this view, a value cannot be inserted into a view column that is derived from the following elements:

- A constant, expression, or scalar function
- The same base table column as some other column of the view

If the object of the insert operation is a view with such columns, a list of column names must be specified, and the list must not identify these columns.

A row can be inserted into a view or a fullselect that is defined using a UNION ALL if the row satisfies the check constraints of exactly one of the underlying base tables. If a row satisfies the check constraints of more than one table, or no table at all, an error is returned (SQLSTATE 23513).

A row cannot be inserted into a view or a fullselect that is defined using a UNION ALL if any base table of the view contains a before trigger and the before trigger contains an UPDATE, a DELETE, or an INSERT operation, or invokes any routine containing such operations (SQLSTATE 42987).

(*column-name*,...)
Specifies the columns for which insert values are provided. Each name must identify a column of the specified table, view, or nickname, or a column in the fullselect. The same column must not be identified more than once. If extended indicator variables are not enabled, a column that cannot accept inserted values (for example, a column based on an expression) must not be identified.

Omission of the column list is an implicit specification of a list in which every column of the table (that is not implicitly hidden) or view, or every item in the select-list of the fullselect is identified in left-to-right order. This list is established when the statement is prepared and, therefore, does not include columns that were added to a table after the statement was prepared.

include-columns
Specifies a set of columns that are included, along with the columns of *table-name* or *view-name*, in the intermediate result table of the INSERT statement when it is nested in the FROM clause of a fullselect. The *include-columns* are appended at the end of the list of columns that are specified for *table-name* or *view-name*.

INCLUDE
Specifies a list of columns to be included in the intermediate result table of the INSERT statement. This clause can only be specified if the INSERT statement is nested in the FROM clause of a fullselect.

column-name
Specifies a column of the intermediate result table of the INSERT statement. The name cannot be the same as the name of another include column or a column in *table-name* or *view-name* (SQLSTATE 42711).

data-type
Specifies the data type of the include column. The data type must be one that is supported by the CREATE TABLE statement.

VALUES
Introduces one or more rows of values to be inserted.

Each row specified in the VALUES clause must be assignable to the implicit or explicit column list and the columns identified in the INCLUDE clause, unless a row variable is used. When a row value list in parentheses is specified, the first value is inserted into the first column in the list, the second value into the second column, and so on. When a row expression is specified, the number of fields in the row type must match the number of names in the implicit or explicit column list.

expression
An *expression* can be any expression defined in the "Expressions" topic. If *expression* is a row type, it must not appear in parentheses. If *expression* is a variable, the host variable can include an indicator variable or in the case of a host structure, an indicator array, enabled for extended indicator variables. If extended indicator variables are enabled, the extended indicator variable values of default (-5) or unassigned (-7) must not be used (SQLSTATE 22539) if either of the following statements is true:

- The expression is more complex than a single host variable with explicit casts
- The target column has data type of structured type

NULL
Specifies the null value and should only be specified for nullable columns.

DEFAULT
Specifies that the default value is to be used. The result of specifying DEFAULT depends on how the column was defined, as follows:

- If the column was defined as a generated column based on an expression, the column value is generated by the system, based on that expression.
- If the IDENTITY clause is used, the value is generated by the database manager.
- If the ROW CHANGE TIMESTAMP clause is used, the value for each inserted row is generated by the database manager as a timestamp that is unique for the table partition within the database partition.
- If the WITH DEFAULT clause is used, the value inserted is as defined for the column (see *default-clause* in "CREATE TABLE").
- If the NOT NULL clause is used and the GENERATED clause is not used, or the WITH DEFAULT clause is not used or DEFAULT NULL is used, the DEFAULT keyword cannot be specified for that column (SQLSTATE 23502).
- When inserting into a nickname, the DEFAULT keyword will be passed through the INSERT statement to the data source only if the data source supports the DEFAULT keyword in its query language syntax.

row-expression
Specifies any row expression of the type described in "Row expressions" that does not include a column name. The number of fields in the row must match the target of the insert and each field must be assignable to the corresponding column.

WITH *common-table-expression*
Defines a common table expression for use with the fullselect that follows.

fullselect
Specifies a set of new rows in the form of the result table of a fullselect. There may be one, more than one, or none. If the result table is empty, SQLCODE is set to +100 and SQLSTATE is set to '02000'.

When the base object of the INSERT and the base object of the fullselect or any subquery of the fullselect, are the same table, the fullselect is completely evaluated before any rows are inserted.

The number of columns in the result table must equal the number of names in the column list. The value of the first column of the result is inserted in the first column in the list, the second value in the second column, and so on.

If the expression that specifies the value of a result column is a variable, the host variable can include an indicator variable enabled for extended indicator variables. If extended indicator variables are enabled, and the expression is more than a single host variable, or a host variable being explicitly cast, then the extended indicator variable values of default or unassigned must not be used (SQLSTATE 22539). The effects of default or unassigned values apply to the corresponding target columns of the *fullselect*.

WITH
Specifies the isolation level at which the statement is executed.

RR
Repeatable Read

RS
Read Stability

CS
Cursor Stability

UR
Uncommitted Read

The default isolation level of the statement is the isolation level of the package in which the statement is bound. The WITH clause has no effect on nicknames, which always use the default isolation level of the statement.

Rules

- ***Triggers:*** INSERT statements may cause triggers to be executed. A trigger may cause other statements to be executed, or may raise error conditions based on the inserted values. If an insert operation into a view causes an INSTEAD OF trigger to fire, validity, referential integrity, and constraints will be checked against the updates that are performed in the trigger, and not against the view that caused the trigger to fire, or its underlying tables.
- ***Default values:*** The value inserted in any column that is not in the column list is either the default value of the column or null. Columns that do not allow null values and are not defined with NOT NULL WITH DEFAULT must be included in the column list. Similarly, if you insert into a view, the value inserted into any column of the base table that is not in the view is either the default value of the column or null. Hence, all columns of the base table that are not in the view must have either a default value or allow null values. The only value that can be inserted into a generated column defined with the GENERATED ALWAYS clause is DEFAULT (SQLSTATE 428C9).
- ***Length:*** If the insert value of a column is a number, the column must be a numeric column with the capacity to represent the integral part of the number. If the insert value of a column is a string, the column must either be a string column with a length attribute at least as great as the length of the string, or a datetime column if the string represents a date, time, or timestamp.
- ***Assignment:*** Insert values are assigned to columns in accordance with specific assignment rules.
- ***Validity:*** If the table named, or the base table of the view named, has one or more unique indexes, each row inserted into the table must conform to the constraints imposed by those indexes. If a view whose definition includes WITH CHECK OPTION is named, each row inserted into the view must conform to the definition of the view. For an explanation of the rules governing this situation, see "CREATE VIEW".
- ***Referential integrity:*** For each constraint defined on a table, each non-null insert value of the foreign key must be equal to a primary key value of the parent table.
- ***Check constraint:*** Insert values must satisfy the check conditions of the check constraints defined on the table. An INSERT to a table with check constraints defined has the constraint conditions evaluated once for each row that is inserted.
- ***XML values:*** A value that is inserted into an XML column must be a well-formed XML document (SQLSTATE 2200M).
- ***Security policy:*** If the identified table or the base table of the identified view is protected with a security policy, the session authorization ID must have the label-based access control (LBAC) credentials that allow:
  - Write access to all protected columns for which a data value is explicitly provided (SQLSTATE 42512)
  - Write access for any explicit value provided for a DB2SECURITYLABEL column for security policies that were created with the RESTRICT NOT AUTHORIZED WRITE SECURITY LABEL option (SQLSTATE 23523)

  The session authorization ID must also have been granted a security label for write access for the security policy if an implicit value is used for a DB2SECURITYLABEL column (SQLSTATE 23523), which can happen when:

  - A value for the DB2SECURITYLABEL column is not explicitly provided
  - A value for the DB2SECURITYLABEL column is explicitly provided but the session authorization ID does not have write access for that value, and the security policy is created with the OVERRIDE NOT AUTHORIZED WRITE SECURITY LABEL option
- **Extended indicator variable usage**: If enabled, negative indicator variable values outside the range of -1 through -7 must not be input (SQLSTATE 22010). Also, if enabled, the default and unassigned extended indicator variable values must not appear in contexts in which they are not supported (SQLSTATE 22539).
- **Extended indicator variables**: In an INSERT statement, a value of unassigned has the effect of setting the column to its default value.

  If the target column is a column defined as GENERATED ALWAYS, then it must be assigned the DEFAULT keyword, or the extended indicator variable-based values of default or unassigned (SQLSTATE 428C9).

Notes

- After execution of an INSERT statement, the value of the third variable of the SQLERRD(3) portion of the SQLCA indicates the number of rows that were passed to the insert operation. In the context of an SQL procedure statement, the value can be retrieved using the ROW_COUNT variable of the GET DIAGNOSTICS statement. SQLERRD(5) contains the count of all triggered insert, update and delete operations.
- Unless appropriate locks already exist, one or more exclusive locks are acquired at the execution of a successful INSERT statement. Until the locks are released, an inserted row can only be accessed by:
  - The application process that performed the insert.
  - Another application process using isolation level UR through a read-only cursor, SELECT INTO statement, or subselect used in a subquery.
- For further information about locking, see the description of the COMMIT, ROLLBACK, and LOCK TABLE statements.
- If an application is running against a partitioned database, and it is bound with option INSERT BUF, then INSERT with VALUES statements which are not processed using EXECUTE IMMEDIATE may be buffered. It is assumed that such an INSERT statement is being processed inside a loop in the application's logic. Rather than execute the statement to completion, it attempts to buffer the new row values in one or more buffers. As a result the actual insertions of the rows into the table are performed later, asynchronous with the application's INSERT logic. Be aware that this asynchronous insertion may cause an error related to an INSERT to be returned on some other SQL statement that follows the INSERT in the application.

  This has the potential to dramatically improve INSERT performance, but is best used with clean data, due to the asynchronous nature of the error handling.
- When a row is inserted into a table that has an identity column, a value is generated for the identity column.
  - For a GENERATED ALWAYS identity column, the value is always generated.
  - For a GENERATED BY DEFAULT column, if a value is not explicitly specified (with a VALUES clause, or subselect), a value is generated.

  The first value generated is the value of the START WITH specification for the identity column.
- When a value is inserted for a user-defined distinct type identity column, the entire computation is done in the source type, and the result is cast to the distinct type before the value is actually assigned to the column. (There is no casting of the previous value to the source type before the computation.)
- When inserting into a GENERATED ALWAYS identity column, a value is always generated for the column, and users must not specify a value at insertion time. If a GENERATED ALWAYS identity column is listed in the column-list of the INSERT statement, with a non-DEFAULT value in the VALUES clause, an error occurs (SQLSTATE 428C9).

For example, assuming that EMPID is defined as an identity column that is GENERATED ALWAYS, then the command:

```
INSERT INTO T2 (EMPID, EMPNAME, EMPADDR)
  VALUES (:hv_valid_emp_id, :hv_name, :hv_addr)
```

will result in an error.

- When inserting into a GENERATED ALWAYS ROW CHANGE TIMESTAMP column, a value is always generated for the column, and users must not specify a value at insertion time (SQLSTATE 428C9) . The value generated is unique for each row inserted on the database partition.
- When inserting into a GENERATED BY DEFAULT column, you can specify an actual value for the column within the VALUES clause, or from a subselect. However, when a value is specified in the VALUES clause, the database manager does not perform any verification of the value. To guarantee uniqueness of IDENTITY column values, a unique index on the identity column must be created.

  When inserting into a table with a GENERATED BY DEFAULT identity column, without specifying a column list, the VALUES clause can specify the DEFAULT keyword to represent the value for the identity column. In such cases, the value for the identity column will be generated.

  ```
  INSERT INTO T2 (EMPID, EMPNAME, EMPADDR)
    VALUES (DEFAULT, :hv_name, :hv_addr)
  ```

  In this example, EMPID is defined as an identity column, and thus the value inserted into this column is generated by the database manager.
- The rules for inserting into an identity column with a subselect are similar to those for an insert with a VALUES clause. A value for an identity column may only be specified if the identity column is defined as GENERATED BY DEFAULT.

  For example, assume T1 and T2 are tables with the same definition, both containing columns *intcol1* and *identcol2* (both are type INTEGER and the second column has the identity attribute). Consider the following insert:

  ```
  INSERT INTO T2
    SELECT *
    FROM T1
  ```

  This example is logically equivalent to:

  ```
  INSERT INTO T2 (intcol1,identcol2)
    SELECT intcol1, identcol2
    FROM T1
  ```

  In both cases, the INSERT statement is providing an explicit value for the identity column of T2. This explicit specification can be given a value for the identity column, but the identity column in T2 must be defined as GENERATED BY DEFAULT. Otherwise, an error will result (SQLSTATE 428C9).

  If there is a table with a column defined as a GENERATED ALWAYS identity, it is still possible to propagate all other columns from a table with the same definition. For example, given the example tables T1 and T2 described previously, the intcol1 values from T1 to T2 can be propagated with the following SQL:

  ```
  INSERT INTO T2 (intcol1)
    SELECT intcol1
    FROM T1
  ```

  Note that, because identcol2 is not specified in the column-list, it will be filled in with its default (generated) value.
- When inserting a row into a single column table where the column is defined as a GENERATED ALWAYS identity column or a ROW CHANGE TIMESTAMP column, it is possible to specify a VALUES clause with the DEFAULT keyword. In this case, the application does not provide any value for the table, and the database manager generates the value for the identity or ROW CHANGE TIMESTAMP column.

```
INSERT INTO IDTABLE
  VALUES(DEFAULT)
```

Assuming the same single column table for which the column has the identity attribute, to insert multiple rows with a single INSERT statement, the following INSERT statement could be used:

```
INSERT INTO IDTABLE
  VALUES (DEFAULT), (DEFAULT), (DEFAULT), (DEFAULT)
```

- When a value for an identity column is generated, that generated value is consumed; the next time that a value is needed, a new value is generated. This is true even when an INSERT statement involving an identity column fails or is rolled back.

  For example, assume that a unique index has been created on the identity column. If a duplicate key violation is detected in generating a value for an identity column, an error occurs (SQLSTATE 23505) and the value generated for the identity column is considered to be consumed. This can occur when the identity column is defined as GENERATED BY DEFAULT and the system tries to generate a new value, but the user has explicitly specified values for the identity column in previous INSERT statements. Reissuing the same INSERT statement in this case can lead to success. The next value for the identity column will be generated, and it is possible that this next value will be unique, and that this INSERT statement will be successful.
- If the maximum value for the identity column is exceeded (or minimum value for a descending sequence) in generating a value for an identity column, an error occurs (SQLSTATE 23522). In this situation, the user would have to DROP and CREATE a new table with an identity column having a larger range (that is, change the data type or increment value for the column to allow for a larger range of values).

  For example, an identity column may have been defined with a data type of SMALLINT, and eventually the column runs out of assignable values. To redefine the identity column as INTEGER, the data would need to be unloaded, the table would have to be dropped and recreated with a new definition for the column, and then the data would be reloaded. When the table is redefined, it needs to specify a START WITH value for the identity column such that the next value generated will be the next value in the original sequence. To determine the end value, issue a query using MAX of the identity column (for an ascending sequence), or MIN of the identity column (for a descending sequence), before unloading the data.
- **Extended indicator variables and insert triggers**: No change in the activation of insert triggers results from use of extended indicator variables. If all columns in the implicit or explicit column list have been assigned to an extended indicator variable-based value of unassigned or default, an insert where all columns have their respective default values is attempted, and if successful, the insert trigger is activated.
- **Extended indicator variables and deferred error checks**: When extended indicator variables are enabled, validation that would otherwise be done in statement preparation, to recognize an insert into a non-updatable column, is deferred until statement execution. Whether an error should be reported can be determined only during execution.
- **Inserting into tables with row-begin, row-end, or transaction start-ID columns**: When a row is inserted into a table with these generated columns (for instance, a system-period temporal table), the database manager assigns values to the following columns:
  - A row-begin column is assigned a value that is generated using a reading of the time-of-day clock during execution of the first data change statement in the transaction that requires a value to be assigned to the row-begin or transaction start-ID column in a table, or a row in a system-period temporal table is deleted. The database manager ensures uniqueness of the generated values for a row-begin column across transactions. If multiple rows are inserted within a single SQL transaction, the values for the row-begin column are the same for all the rows and are unique from the values generated for the column for another transaction.
  - A row-end column is assigned the maximum value for the data type of the column (9999-12-30-00.00.00.000000000000).

  - A transaction start-ID column is assigned a unique timestamp value per transaction or the null value. The null value is assigned to the transaction start-ID column if the column is nullable. Otherwise, the value is generated using a reading of the time-of-day clock during execution of the first data change statement in the transaction that requires a value to be assigned to the row-begin or transaction start-ID column in a table, or a row in a system-period temporal table is deleted. If multiple rows are inserted within a single SQL transaction, the values for the transaction start-ID column are the same for all the rows and are unique from the values generated for the column for another transaction.
- **Inserting into a system-period temporal table**: When a row is inserted into a system-period temporal table, the database manager assigns values to columns as indicated for tables with row-begin, row-end, or transaction start-ID columns. Also, when a row is inserted, no rows are added to the history table associated with the system-period temporal table.
- **Inserting into application-period temporal tables**: An error is returned when a row is inserted into an application-period temporal table and the following conditions are met:
  - The application-period temporal table has either a primary key or unique constraint with the BUSINESS_TIME WITHOUT OVERLAPS clause defined, or a unique index with the BUSINESS_TIME WITHOUT OVERLAPS clause defined.
  - The period defined by the begin and end columns of the BUSINESS_TIME period overlap the period defined by the begin and end columns of the BUSINESS_TIME period for another row that matches the other columns of the same unique constraint or unique index.
- **Considerations for an INSERT without a column list**: An INSERT statement without a column list does not include implicitly hidden columns. Columns that are defined as implicitly hidden and not null must have a defined default value.

Examples

- *Example 1:* Insert a new department with the following specifications into the DEPARTMENT table:
  - Department number (DEPTNO) is 'E31'
  - Department name (DEPTNAME) is 'ARCHITECTURE'
  - Managed by (MGRNO) a person with number '00390'
  - Reports to (ADMRDEPT) department 'E01'.

```
INSERT INTO DEPARTMENT
  VALUES ('E31', 'ARCHITECTURE', '00390', 'E01')
```

- *Example 2:* Insert a new department into the DEPARTMENT table as in example 1, but do not assign a manager to the new department.

```
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT )
  VALUES ('E31', 'ARCHITECTURE', 'E01')
```

- *Example 3:* Insert two new departments using one statement into the DEPARTMENT table as in example 2, but do not assign a manager to the new department.

```
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT)
  VALUES ('B11', 'PURCHASING', 'B01'),
         ('E41', 'DATABASE ADMINISTRATION', 'E01')
```

- *Example 4:* Create a temporary table MA_EMP_ACT with the same columns as the EMP_ACT table. Load MA_EMP_ACT with the rows from the EMP_ACT table with a project number (PROJNO) starting with the letters 'MA'.

```
CREATE TABLE MA_EMP_ACT
         ( EMPNO CHAR(6)  NOT NULL,
          PROJNO CHAR(6)  NOT NULL,
          ACTNO SMALLINT  NOT NULL,
          EMPTIME DEC(5,2),
          EMSTDATE DATE,
          EMENDATE  DATE )
INSERT INTO MA_EMP_ACT
```

```
  SELECT * FROM EMP_ACT
    WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

- *Example 5:* Use a C program statement to add a skeleton project to the PROJECT table. Obtain the project number (PROJNO), project name (PROJNAME), department number (DEPTNO), and responsible employee (RESPEMP) from host variables. Use the current date as the project start date (PRSTDATE). Assign a null value to the remaining columns in the table.

```
EXEC SQL INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTDATE)
  VALUES (:PRJNO, :PRJNM, :DPTNO, :REMP, CURRENT DATE);
```

- *Example 6:* Specify an INSERT statement as the *data-change-table-reference* within a SELECT statement. Define an extra include column whose values are specified in the VALUES clause, which is then used as an ordering column for the inserted rows.

```
SELECT INORDER.ORDERNUM
  FROM NEW TABLE (INSERT INTO ORDERS(CUSTNO)INCLUDE (INSERTNUM INTEGER)
    VALUES(:CNUM1, 1), (:CNUM2, 2)) InsertedOrders
  ORDER BY INSERTNUM;
```

- *Example 7:* Use a C program statement to add a document to the DOCUMENTS table. Obtain values for the document ID (DOCID) column and the document data (XMLDOC) column from a host variable that binds to an SQL TYPE IS XML AS BLOB_FILE.

```
EXEC SQL INSERT INTO DOCUMENTS
  (DOCID, XMLDOC) VALUES (:docid, :xmldoc)
```

- *Example 8:* For the following INSERT statements, assume that table SALARY_INFO is defined with three columns, and that the last column is an implicitly hidden ROW CHANGE TIMESTAMP column. In the following statement, the implicitly hidden column is explicitly referenced in the column list and a value is provided for it in the VALUES clause.

```
INSERT INTO SALARY_INFO (LEVEL, SALARY, UPDATE_TIME)
  VALUES (2, 30000, CURRENT TIMESTAMP)
```

  The following INSERT statement uses an implicit column list. An implicit column list does not include implicitly hidden columns, so the VALUES clause only contains values for the other two columns.

```
INSERT INTO SALARY_INFO VALUES (2, 30000)
```

  In this case, the UPDATE_TIME column must be defined to have a default value, and that default value is used for the row that is inserted.

ITERATE

The ITERATE statement causes the flow of control to return to the beginning of a labelled loop.

Invocation

This statement can be embedded in an:

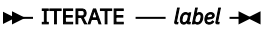
- SQL procedure definition
- Compound SQL (compiled) statement
- Compound SQL (inlined) statement

The compound statements can be embedded in an SQL procedure definition, SQL function definition, or SQL trigger definition. It is not an executable statement and cannot be dynamically prepared.

Authorization

None required.

Syntax



Description

label
Specifies the label of the FOR, LOOP, REPEAT, or WHILE statement to which the database server passes the flow of control.

Example

This example uses a cursor to return information for a new department. If the *not_found* condition handler was invoked, the flow of control passes out of the loop. If the value of *v_dept* is 'D11', an ITERATE statement passes the flow of control back to the top of the LOOP statement. Otherwise, a new row is inserted into the DEPARTMENT table.

```
CREATE PROCEDURE ITERATOR()
  LANGUAGE SQL
  BEGIN
    DECLARE v_dept CHAR(3);
    DECLARE v_deptname VARCHAR(29);
    DECLARE v_admdept CHAR(3);
    DECLARE at_end INTEGER DEFAULT 0;
    DECLARE not_found CONDITION FOR SQLSTATE '02000';
    DECLARE c1 CURSOR FOR
      SELECT deptno, deptname, admrdept
      FROM department
      ORDER BY deptno;
    DECLARE CONTINUE HANDLER FOR not_found
      SET at_end = 1;
    OPEN c1;
    ins_loop:
    LOOP
      FETCH c1 INTO v_dept, v_deptname, v_admdept;
      IF at_end = 1 THEN
        LEAVE ins_loop;
      ELSEIF v_dept = 'D11' THEN
        ITERATE ins_loop;
      END IF;
      INSERT INTO department (deptno, deptname, admrdept)
      VALUES ('NEW', v_deptname, v_admdept);
    END LOOP;
    CLOSE c1;
  END
```

LEAVE

The LEAVE statement transfers program control out of a loop or a compound statement.

Invocation

This statement can be embedded in an:

- SQL procedure definition
- Compound SQL (compiled) statement
- Compound SQL (inlined) statement

The compound statements can be embedded in an SQL procedure definition, SQL function definition, or SQL trigger definition. It is not an executable statement and cannot be dynamically prepared.

Authorization

None required.

Syntax

Description

label
Specifies the label of the compound, FOR, LOOP, REPEAT, or WHILE statement to exit.

Notes

- When a LEAVE statement transfers control out of a compound statement, all open cursors in the compound statement, except cursors that are used to return result sets, are closed.

Example

This example contains a loop that fetches data for cursor *c1*. If the value of SQL variable *at_end* is not zero, the LEAVE statement transfers control out of the loop.

```
CREATE PROCEDURE LEAVE_LOOP(OUT counter INTEGER)
  LANGUAGE SQL
  BEGIN
    DECLARE v_counter INTEGER;
    DECLARE v_firstnme VARCHAR(12);
    DECLARE v_midinit CHAR(1);
    DECLARE v_lastname VARCHAR(15);
    DECLARE at_end SMALLINT DEFAULT 0;
    DECLARE not_found CONDITION FOR SQLSTATE '02000';
    DECLARE c1 CURSOR FOR
      SELECT firstnme, midinit, lastname
        FROM employee;
    DECLARE CONTINUE HANDLER for not_found
      SET at_end = 1;
    SET v_counter = 0;
    OPEN c1;
    fetch_loop:
    LOOP
      FETCH c1 INTO v_firstnme, v_midinit, v_lastname;
      IF at_end <> 0 THEN LEAVE fetch_loop;
      END IF;
      SET v_counter = v_counter + 1;
    END LOOP fetch_loop;
    SET counter = v_counter;
    CLOSE c1;
  END
```

LOCK TABLE

The LOCK TABLE statement prevents concurrent application processes from using or changing a table. The lock is released when the unit of work issuing the LOCK TABLE statement either commits or terminates.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

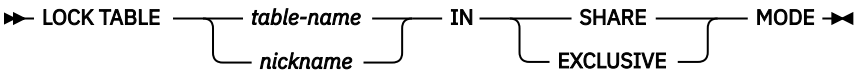
Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- SELECT privilege on the table
- SELECTIN privilege on the schema containing the table
- CONTROL privilege on the table

- DATAACCESS authority on the schema containing the table
- DATAACCESS authority

Syntax



Description

table-name* or *nickname
Identifies the table or nickname. The *table-name* must identify a table that exists at the application server, but it must not identify a catalog table, a created temporary table, or a declared temporary table (SQLSTATE 42995). If the *table-name* is a typed table, it must be the root table of the table hierarchy (SQLSTATE 428DR). When a nickname is specified, the database manager will lock the underlying object (that is, a table or view) of the data source to which the nickname refers.

IN SHARE MODE
Prevents concurrent application processes from executing any but read-only operations on the table.

IN EXCLUSIVE MODE
Prevents concurrent application processes from executing any operations on the table. Note that EXCLUSIVE MODE does not prevent concurrent application processes that are running at isolation level Uncommitted Read (UR) from executing read-only operations on the table.

Notes

- Locking is used to prevent concurrent operations. A lock is not necessarily acquired during execution of the LOCK TABLE statement if a suitable lock already exists. The lock that prevents concurrent operations is held at least until termination of the unit of work.
- In a partitioned database, if the LOCK TABLE statement is interrupted, the table may be locked on some database partitions but not on others. If this occurs, either issue another LOCK TABLE statement to complete the locking on all database partitions, or issue a COMMIT or ROLLBACK statement to release the current locks.
- This statement affects all database partitions in the database partition group.
- For partitioned tables, the only lock acquired for the LOCK TABLE statement is at the table level; no data partition locks are acquired.

Example

Obtain a lock on the table EMP. Do not allow other programs to read or update the table.

```
LOCK TABLE EMP IN EXCLUSIVE MODE
```

LOOP

The LOOP statement repeats the execution of a statement or a group of statements.

Invocation

This statement can be embedded in an:

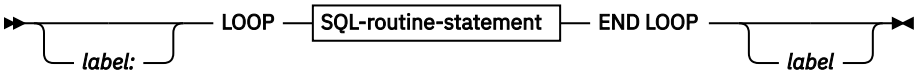
- SQL procedure definition
- Compound SQL (compiled) statement
- Compound SQL (inlined) statement

The compound statements can be embedded in an SQL procedure definition, SQL function definition, or SQL trigger definition. It is not an executable statement and cannot be dynamically prepared.

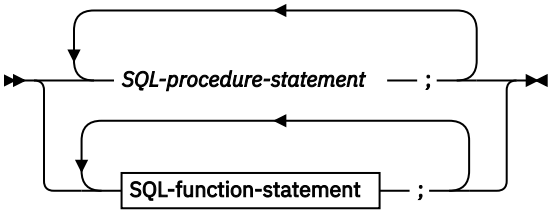
Authorization

No privileges are required to invoke the LOOP statement. However, the authorization ID of the statement must hold the necessary privileges to invoke the SQL statements that are embedded in the LOOP statement.

Syntax



SQL-routine-statement



Description

label
Specifies the label for the LOOP statement. If the beginning label is specified, that label can be specified on LEAVE and ITERATE statements. If the ending label is specified, a matching beginning label must be specified.

SQL-procedure-statement
Specifies the SQL statements that are to be invoked in the loop. *SQL-procedure-statement* is only applicable when in the context of an SQL procedure or Compound SQL (compiled) statement. See *SQL-procedure-statement* in "Compound SQL (compiled)" statement.

SQL-function-statement
Specifies the SQL statements that are to be invoked in the loop. *SQL-function-statement* is only applicable when in the context of an SQL function, SQL method, or Compound SQL (inlined) statement. See *SQL-function-statement* in "FOR".

Example

This procedure uses a LOOP statement to fetch values from the `employee` table. Each time the loop iterates, the OUT parameter *counter* is incremented and the value of *v_midinit* is checked to ensure that the value is not a single space (' '). If *v_midinit* is a single space, the LEAVE statement passes the flow of control outside of the loop.

```
CREATE PROCEDURE LOOP_UNTIL_SPACE(OUT counter INTEGER)
  LANGUAGE SQL
  BEGIN
    DECLARE v_counter INTEGER DEFAULT 0;
    DECLARE v_firstnme VARCHAR(12);
    DECLARE v_midinit CHAR(1);
    DECLARE v_lastname VARCHAR(15);
    DECLARE c1 CURSOR FOR
      SELECT firstnme, midinit, lastname
        FROM employee;
    DECLARE CONTINUE HANDLER FOR NOT FOUND
      SET counter = -1;
    OPEN c1;
    fetch_loop:
    LOOP
      FETCH c1 INTO v_firstnme, v_midinit, v_lastname;
      IF v_midinit = ' ' THEN
```

```
          LEAVE fetch_loop;
        END IF;
        SET v_counter = v_counter + 1;
      END LOOP fetch_loop;
      SET counter = v_counter;
      CLOSE c1;
    END
```

MERGE

The MERGE statement updates a target (a table or view, or the underlying tables or views of a fullselect) using data from a source (result of a table reference).

Rows in the target that match the source can be deleted or updated as specified, and rows that do not exist in the target can be inserted. Updating, deleting or inserting a row in a view updates, deletes or inserts the row in the tables on which the view is based.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- If an insert operation is specified, INSERT privilege on the table or view or INSERTIN privilege on the schema containing the table or view; if a delete operation is specified, DELETE privilege on the table or view or DELETEIN privilege on the schema containing the table or view; and if an update operation is specified, either:
  - UPDATE privilege on the table or view
  - UPDATE privilege on each column that is to be updated
  - UPDATEIN privilege on the schema containing the table or view
- CONTROL privilege on the table
- DATAACCESS authority on the schema containing the table or view
- DATAACCESS authority

The privileges held by the authorization ID of the statement must also include at least one of the following authorities:

- SELECT privilege on every table or view identified in the *table-reference*
- SELECTIN privilege on the schema containing the tables or views identified in the table-reference
- CONTROL privilege on the tables or views identified in the *table-reference*
- DATAACCESS authority on the schema containing the tables or views identified in the table-reference
- DATAACCESS authority

If *search-condition*, *insert-operation*, or *assignment-clause* includes a subquery, the privileges held by the authorization ID of the statement must also include at least one of the following authorities:

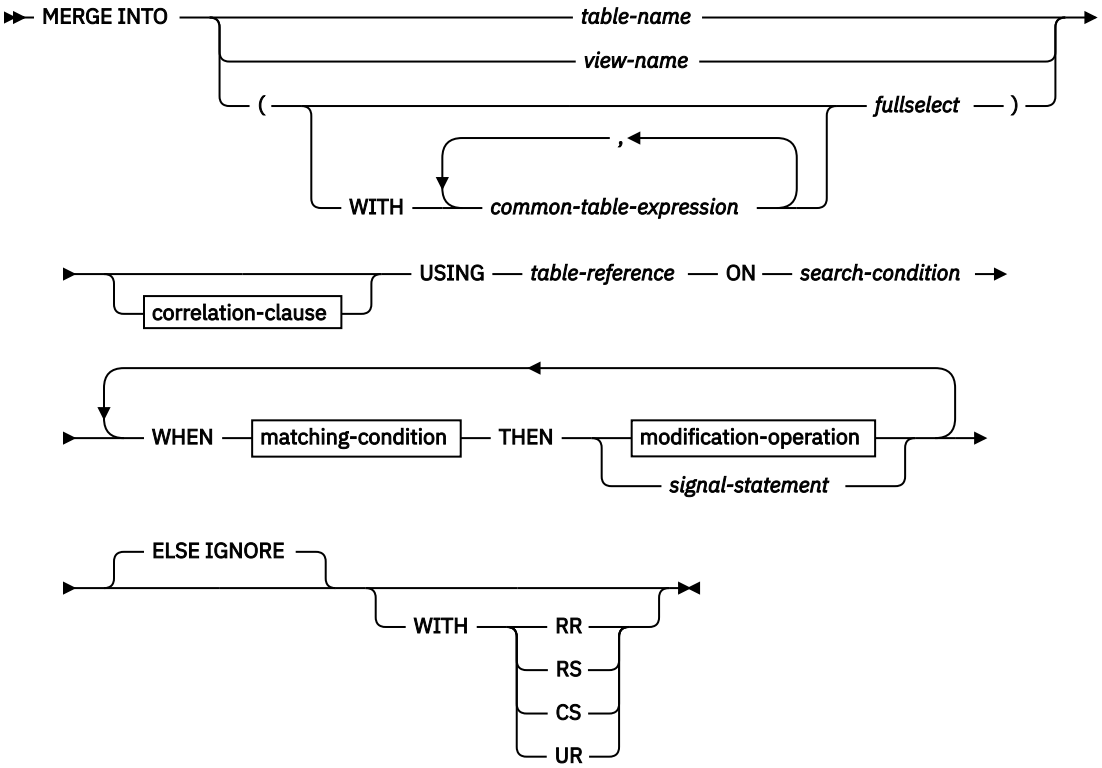
- SELECT privilege on every table or view identified in the subquery
- SELECTIN privilege on the schema containing the table or view identified in the subquery
- CONTROL privilege on the tables or views identified in the subquery
- DATAACCESS authority on the schema containing the table or view identified in the subquery
- DATAACCESS authority

If a *row-fullselect* is included in the assignment, the privileges held by the authorization ID of the statement must include at least one of the following authorities for each referenced table, view, or nickname:

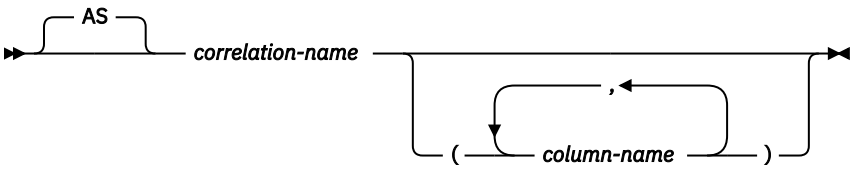
- SELECT privilege
- SELECTIN privilege on the schema containing the table, view or nickname referenced
- CONTROL privilege
- DATAACCESS authority on the schema containing the table, view or nickname referenced
- DATAACCESS authority

If an expression that refers to a function is specified, the privilege set must include any authority that is necessary to execute the function.

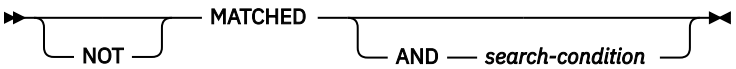
Syntax



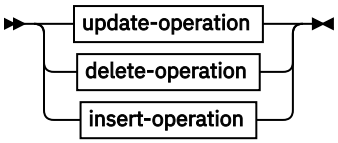
correlation-clause



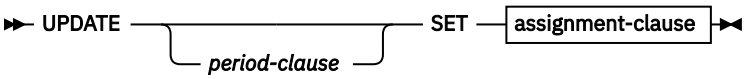
matching-condition



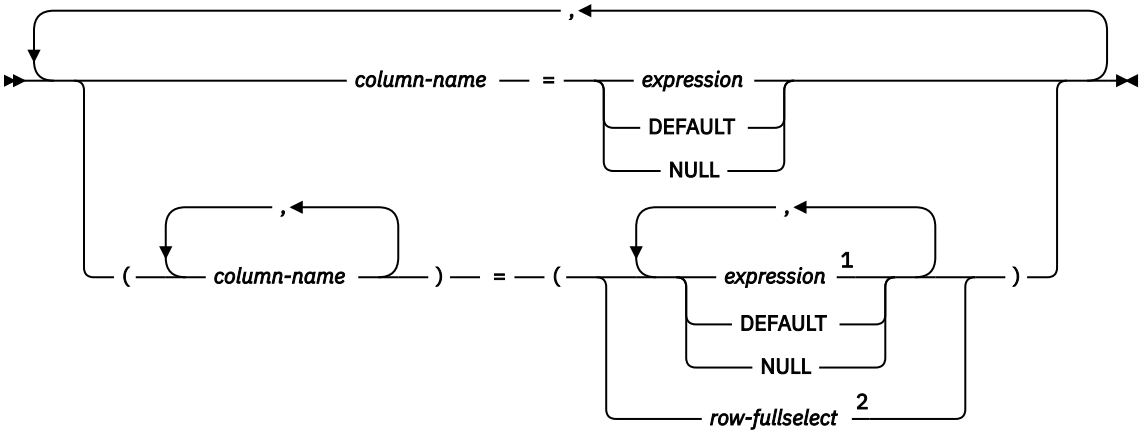
modification-operation



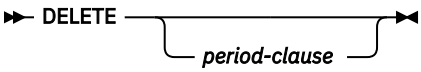
update-operation



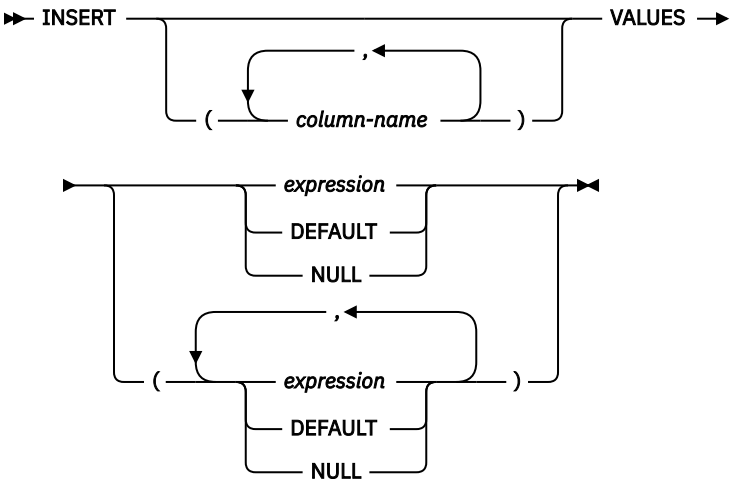
assignment-clause



delete-operation



insert-operation



period-clause

Notes:

[1] The number of expressions, NULLs, and DEFAULTs must match the number of column names.
[2] The number of columns in the select list must match the number of column names.

Description

table-name*, *view-name*, or *(fullselect)

Identifies the target of the update, delete, or insert operations of the merge. The name must identify a table or view that exists at the current server, but it must not identify a catalog table, a system-maintained materialized query table, a view of a catalog table, a read-only view, or a view that directly or indirectly contains a WHERE clause that references a subquery or a routine defined with NOT DETERMINISTIC or EXTERNAL ACTION (SQLSTATE 42807).

If the target of the merge operation is a fullselect, the fullselect must be updatable, deletable, or insertable as defined in the "Updatable views", "Deletable views", or "Insertable views" Notes items in the description of the CREATE VIEW statement.

You cannot use a *period-clause* in an update-operation or a delete-operation if the target of the merge operation is a union-all view or a fullselect.

You cannot use a nickname (a reference to a remote, federated table) as the target table.

correlation-clause
Can be used within *search-condition* or on the right side of an *assignment-clause* to designate a table, view, or fullselect. For a description of *correlation-clause*, see "table-reference" in the description of "Subselect".

USING *table-reference*
Specifies a set of rows as a result table to be merged into the target. If the result table is empty, a warning is returned (SQLSTATE 02000).

ON *search-condition*
Logically, a right join is performed between the target table and the *table-reference* using the ON *search-condition*. For those rows of the join result table where the search condition is true, the specified update or delete operation is performed. For those rows of the join result table where the result of the search condition is not true, the specified insert operation is performed.

The *search-condition* has the following restrictions (SQLSTATE 42972 unless otherwise noted):

- It cannot contain any subqueries, scalar or otherwise
- It cannot include any dereference operations or the DEREF function where the reference value is other than the object identifier column
- It cannot include an SQL function
- It cannot include an XMLQUERY or XMLEXISTS expression
- Any column that is referenced in an expression of the *search-condition* must be a column of the target table, view, or *table-reference*
- Any function that is referenced in an expression of the *join-condition* of a full outer join must be deterministic and have no external action
- It cannot be include an aggregate function (SQLSTATE 42903)

If the *search-condition* is false or unknown for every row in *table-reference*, a warning is returned (SQLSTATE 02000).

WHEN *matching-condition*
Specifies the condition under which the *modification-operation* or the *signal-statement* is executed. Each *matching-condition* is evaluated in order of specification. Rows for which the *matching-condition* evaluates to true are not considered in subsequent matching conditions.

MATCHED
Indicates the operation to be performed on the rows where the ON search condition is true. Only UPDATE, DELETE, or *signal-statement* can be specified after THEN.

AND *search-condition*
Specifies a further search condition to be applied against the rows that matched the ON search condition for the operation to be performed after THEN.

NOT MATCHED
Indicates the operation to be performed on the rows where the ON search condition is false or unknown. Only INSERT or *signal-statement* can be specified after THEN.

AND *search-condition*
Specifies a further search condition to be applied against the rows that did not match the ON search condition for the operation to be performed after THEN. This search condition applies only to rows that did not match ON *search condition*; if AND search condition references columns from the target table a syntax error might be returned (SQL0206N).

THEN *modification-operation*

Specifies the operation to execute when the *matching-condition* evaluates to true.

update-operation

Specifies the update operation to be executed for the rows where the *matching-condition* evaluates to true.

UPDATE

Introduces the update operation.

period-clause

Specifies that a period clause is applied to the update operation in the MERGE statement. For more information about the effects of a period clause specified in the context of an update operation, see the UPDATE statement topic.

SET

Introduces the assignment of values to column names.

assignment-clause

Specifies a list of column updates.

column-name

Identifies a column to be updated. The *column-name* must identify a column of the specified table or view, but not a view column derived from a scalar function, constant, or expression. A column must not be specified more than once (SQLSTATE 42701).

A view column derived from the same column as another column of the view can be updated, but both columns cannot be updated in the same MERGE statement (SQLSTATE 42701).

expression

Indicates the new value of the column. The *expression* must not include an aggregate function except when it occurs within a scalar fullselect (SQLSTATE 42903).

An *expression* can contain references to columns of the *table-name* or *view-name*. For each row that is updated, the value of such a column in an expression is the value of the column in the row before the row is updated.

If *expression* is a reference to a single column of the source table, the source table column value may have been specified with an extended indicator variable value. The effects of such indicator variables apply to the corresponding target columns of the *assignment-clause*.

If *expression* is a single host variable, or a host variable being explicitly cast, the host variable can include an indicator variable that is enabled for extended indicator variables.

When extended indicator variables are enabled, the extended indicator variable values of default (-5) or unassigned (-7) must not be used (SQLSTATE 22539) if either of the following statements is true:

- The expression is more complex than a single host variable with explicit casts
- The target column has data type of structured type

DEFAULT

The default value assigned to the column. DEFAULT can be specified only for columns that have a default value. For information about default values of data types, see the description of the DEFAULT clause in the "CREATE TABLE" statement.

DEFAULT must be specified for a column that was defined as GENERATED ALWAYS. A valid value can be specified for a column that was defined as GENERATED BY DEFAULT.

NULL
Specifies the null value as the new value of the column. Specify NULL only for nullable columns (SQLSTATE 23502).

row-fullselect
Specifies a fullselect that returns a single row. The result column values are assigned to each corresponding *column-name*. If the fullselect returns no rows, the null value is assigned to each column; an error occurs if any column to be updated is not nullable. An error also occurs if there is more than one row in the result.

A *row-fullselect* can contain references to columns of the target table of the MERGE statement. For each row that is updated, the value of such a column in an expression is the value of the column in the row before the row is updated. An error is returned if there is more than one row in the result (SQLSTATE 21000).

delete-operation
Specifies the delete operation to be executed for the rows where the *matching-condition* evaluates to true.

DELETE
Introduces the delete operation.

period-clause
Specifies that a period clause is applied to the delete operation in the MERGE statement. For more information about the effects of a period clause specified in the context of a delete operation, see the DELETE statement topic.

insert-operation
Specifies the insert operation to be executed for the rows where the *matching-condition* evaluates to true.

INSERT
Introduces a list of column names and row value expressions to be used for the insert operation.

The number of values for the row in the row value expression must equal the number of names in the insert column list. The first value is inserted in the first column in the list, the second value in the second column, and so on.

(*column-name,...*)
Specifies the columns for which the insert values are provided. Each name must identify a column of the table or view. The same column must not be identified more than once (SQLSTATE 42701). A view column that cannot accept insert values must not be identified. A value cannot be inserted into a view column that is derived from:

- A constant, expression, or scalar function
- The same base table column as some other column of the view

If the object of the operation is a view with such columns, a list of column names must be specified, and the list must not identify these columns.

Omission of the column list is an implicit specification of a list in which every column of the table (that is not defined as implicitly hidden) or view is identified in left-to-right order. This list is established when the statement is prepared, and therefore does not include columns that were added to a table after the statement was prepared.

VALUES
Introduces one or more rows of values to be inserted.

expression
Any expression that does not include a column name (SQLSTATE 42703).

If *expression* is a reference to a single column of the source table, the source table column value may have been specified with an extended indicator variable value. The effects of such indicator variables apply to the corresponding target columns of the *insert-operation*.

If *expression* is a single host variable, or a host variable being explicitly cast, the host variable can include an indicator variable (or in the case of a host structure, an indicator array) that is enabled for extended indicator variables.

When extended indicator variables are enabled, the extended indicator variable values of default (-5) or unassigned (-7) must not be used (SQLSTATE 22539) if either of the following statements is true:

- The expression is more complex than a single host variable with explicit casts
- The target column has data type of structured type

DEFAULT
The default value assigned to the column. DEFAULT can be specified only for columns that have a default value. For information about default values of data types, see the description of the DEFAULT clause in the "CREATE TABLE" statement.

DEFAULT must be specified for a column that was defined as GENERATED ALWAYS. A valid value can be specified for a column that was defined as GENERATED BY DEFAULT.

NULL
Specifies the null value as the value of the column. Specify NULL only for nullable columns (SQLSTATE 23502).

signal-statement
Specifies the SIGNAL statement that is to be executed to return an error when the *matching-condition* evaluates to true.

ELSE IGNORE
Specifies that no action is to be taken for the rows where no *matching-condition* evaluates to true. If all rows of *table-reference* are ignored, a warning is returned (SQLSTATE 02000).

WITH
Specifies the isolation level at which the MERGE statement is executed.

RR
Repeatable Read

RS
Read Stability

CS
Cursor Stability

UR
Uncommitted Read

The default isolation level of the statement is the isolation level of the package in which the statement is bound.

Rules

- More than one *modification-operation* (UPDATE SET, DELETE, or *insert-operation*), or *signal-statement* can be specified in a single MERGE statement.
- Each row in the target can only be operated on once. A row in the target can only be identified as MATCHED with one row in the result table of the *table-reference* (SQLSTATE 21506). A nested SQL operation (RI or trigger except INSTEAD OF trigger) cannot specify the target table (or a table within the same table hierarchy) as a target of an UPDATE, DELETE, INSERT, or MERGE statement (SQLSTATE 27000).
- ***Security policy:*** If the identified target table or the base table of the identified target view is protected with a security policy, the session authorization ID must have the label-based access control (LBAC) credentials that allow the following types of access.

– For the update operation:
  - Write access to all protected columns that are being updated (SQLSTATE 42512)
  - Write access for any explicit value provided for a DB2SECURITYLABEL column for security policies that were created with the RESTRICT NOT AUTHORIZED WRITE SECURITY LABEL option (SQLSTATE 23523)
  - Read and write access to all rows that are being updated (SQLSTATE 42519)

  The session authorization ID must also have been granted a security label for write access for the security policy if an implicit value is used for a DB2SECURITYLABEL column (SQLSTATE 23523), which can happen when:
  - The DB2SECURITYLABEL column is not included in the list of columns that are to be updated (and so it will be implicitly updated to the security label for write access of the session authorization ID)
  - A value for the DB2SECURITYLABEL column is explicitly provided but the session authorization ID does not have write access for that value, and the security policy is created with the OVERRIDE NOT AUTHORIZED WRITE SECURITY LABEL option

– For the delete operation:
  - Write access to all protected columns (SQLSTATE 42512)
  - Read and write access to all of the rows that are selected for deletion (SQLSTATE 42519)

– For the insert operation:
  - Write access to all protected columns for which a data value is explicitly provided (SQLSTATE 42512)
  - Write access for any explicit value provided for a DB2SECURITYLABEL column for security policies that were created with the RESTRICT NOT AUTHORIZED WRITE SECURITY LABEL option (SQLSTATE 23523)

  The session authorization ID must also have been granted a security label for write access for the security policy if an implicit value is used for a DB2SECURITYLABEL column (SQLSTATE 23523), which can happen when:
  - A value for the DB2SECURITYLABEL column is not explicitly provided
  - A value for the DB2SECURITYLABEL column is explicitly provided but the session authorization ID does not have write access for that value, and the security policy is created with the OVERRIDE NOT AUTHORIZED WRITE SECURITY LABEL option

- ***INSTEAD OF triggers***: If a view is specified as the target of the MERGE statement, either no INSTEAD OF triggers should be defined for the view, or an INSTEAD OF trigger should be defined for each of the update, delete, and insert operations (SQLSTATE 428FZ).
- **Extended indicator variable usage**: If enabled, negative indicator variable values outside the range of -1 through -7 must not be input (SQLSTATE 22010). Also, if enabled, the default and unassigned extended indicator variable values must not appear in contexts in which they are not supported (SQLSTATE 22539).
- **Extended indicator variables in the *assignment-clause***: An *expression* that is a reference to a single column of the source table, a single host variable, or a host variable being explicitly cast can result in assigning an extended indicator variable-based value. Assigning the extended indicator variable-based value of unassigned has the effect of leaving the target column set to its current value, as if it had not been specified in the statement. Assigning the extended indicator variable-based value of default assigns the default value of the column. For information on default values of data types, see the description of the DEFAULT clause in “CREATE TABLE ” on page 1351.

  If a target column is not updatable (for example, a column in a view that is defined as an expression), then it must be assigned the extended indicator variable-based value of unassigned (SQLSTATE 42808).

  If the target column is a column defined as GENERATED ALWAYS, then it must be assigned the DEFAULT keyword, or the extended indicator variable-based values of default or unassigned (SQLSTATE 428C9).

The *assignment-clause* must not assign all target columns to an extended indicator variable-based value of unassigned (SQLSTATE 22540).

- **Extended indicator variables in the *insert-operation***: An *expression* that is a reference to a single column of the source table, a single host variable, or a host variable being explicitly cast can result in inserting an extended indicator variable-based value. In *insert-operation*, a value of unassigned has the effect of setting the column to its default value.

  If a target column is not updatable, then it must be assigned the extended indicator variable-based value of unassigned (SQLSTATE 42808), unless it is a column defined as GENERATED ALWAYS. If the target column is a column defined as GENERATED ALWAYS, then it must be assigned the DEFAULT keyword, or the extended indicator variable-based values of default or unassigned (SQLSTATE 428C9).

For other rules that affect the update, insert, or delete operation portion of the MERGE statement, see the "Rules" section of the corresponding statement description.

Notes

- ***Order of processing***:
  1. Determine the set of rows to be processed from the source and target. If CURRENT TIMESTAMP is used in this statement, only one clock reading is done for the whole statement.
  2. Use the ON clause to classify these rows as either MATCHED or NOT MATCHED.
  3. Evaluate any *matching-condition* in the WHEN clauses.
  4. Evaluate any *expression* in any *assignment-clause* and *insert-operation*.
  5. Execute each *signal-statement*.
  6. Apply each *modification-operation* to the applicable rows in the order of specification. The constraints and triggers activated by each *modification-operation* are executed for the *modification-operation*. Statement-level triggers are activated even if no rows satisfy the *modification-operation*. Each *modification-operation* can affect the triggers and referential constraints of each subsequent *modification-operation*.
- ***Statement level atomicity***: If an error occurs during execution of the MERGE statement, the whole statement is rolled back.
- ***Number of rows updated***: When a MERGE statement completes execution, the value of the ROW_COUNT item for GET DIAGNOSTICS and SQLERRD(3) in the SQLCA is the number of rows operated on by the MERGE statement, excluding rows identified by the ELSE IGNORE clause. The value in SQLERRD(3) does not include the number of rows that were operated on as a result of constraints or triggers. The value in SQLERRD(5) includes the number of these rows.
- ***Inserted row cannot also be updated***: No attempt is made to update a row in the target that did not already exist before the MERGE statement was executed; that is, there are no updates of rows that were inserted by the MERGE statement.
- **Extended indicator variables and update triggers**: If a target column has been assigned with an extended indicator variable-based value of unassigned, that column is not considered to have been updated. That column is treated as if it had not been specified in the OF *column-name* list of any update trigger defined on the target table.
- **Extended indicator variables and insert triggers**: No change in the activation of insert triggers results from the use of extended indicator variables. If all columns in the implicit or explicit column list have been assigned to an extended indicator variable-based value of unassigned or default, an insert where all columns have their respective default values is attempted. If the insert is successful, the insert trigger is activated.
- **Extended indicator variables and deferred error checks**: When extended indicator variables are enabled, validation that would otherwise be done in statement preparation to recognize an insert into, or update of, a non-updatable column, is deferred until statement execution. Whether an error should be reported can be determined only during execution.
- **Considerations for system-period temporal tables**: When MERGE is processed for a system-period temporal table, the rows are impacted in the same way as if the specific data change operations

had been invoked. See UPDATE statement, DELETE statement, and INSERT statement topics for more information.

- **Considerations for application-period temporal tables and triggers**; When a row is deleted and the FOR PORTION OF BUSINESS_TIME clause is specified, additional rows may be implicitly inserted to reflect any portion of the row that was not deleted. Any existing delete triggers are activated for the rows deleted, and any existing insert triggers are activated for rows that are implicitly inserted. When a row is updated and the FOR PORTION OF BUSINESS_TIME clause is specified, additional rows may be implicitly inserted to reflect any portion of the row that was not updated. Any existing update triggers are activated for the rows updated, and any existing insert triggers are activated for rows that are implicitly inserted.
- **Considerations for a MERGE without a column list in the insert-operation**: A MERGE statement without a column list specified as part of the insert-operation does not include implicitly hidden columns. Columns that are defined as implicitly hidden and not null must have a defined default value.

Examples

- *Example 1:* For activities whose description has been changed, update the description in the archive table. For new activities, insert into the archive table. The archive and activities table both have activity as a primary key.

```
MERGE INTO archive ar
USING (SELECT activity, description FROM activities) ac
ON (ar.activity = ac.activity)
WHEN MATCHED THEN
   UPDATE SET
      description = ac.description
WHEN NOT MATCHED THEN
   INSERT
      (activity, description)
      VALUES (ac.activity, ac.description)
```

- *Example 2:* Using the shipment table, merge rows into the inventory table, increasing the quantity by part count in the shipment table for rows that match; else insert the new *partno* into the inventory table.

```
MERGE INTO inventory AS in
USING (SELECT partno, description, count FROM shipment
       WHERE shipment.partno IS NOT NULL) AS sh
ON (in.partno = sh.partno)
WHEN MATCHED THEN
   UPDATE SET
      description = sh.description,
      quantity = in.quantity + sh.count
WHEN NOT MATCHED THEN
   INSERT
      (partno, description, quantity)
      VALUES (sh.partno, sh.description, sh.count)
```

- *Example 3:* Using the transaction table, merge rows into the account table, updating the balance from the set of transactions against an account ID and inserting new accounts from the consolidated transactions where they do not already exist.

```
MERGE INTO account AS a
USING (SELECT id, sum(amount) sum_amount FROM transaction
       GROUP BY id) AS t
ON a.id = t.id
WHEN MATCHED THEN
   UPDATE SET
      balance = a.balance + t.sum_amount
WHEN NOT MATCHED THEN
   INSERT
      (id, balance)
      VALUES (t.id, t.sum_amount)
```

- *Example 4:* Using the transaction_log table, merge rows into the employee_file table, updating the phone and office with the latest transaction_log row based on the transaction time, and inserting the latest new employee_file row where the row does not already exist.

```
MERGE INTO employee_file AS e
USING (SELECT empid, phone, office
       FROM (SELECT empid, phone, office,
             ROW_NUMBER() OVER (PARTITION BY empid
             ORDER BY transaction_time DESC) rn
             FROM transaction_log) AS nt
             WHERE rn = 1) AS t
ON e.empid = t.empid
WHEN MATCHED THEN
   UPDATE SET
      (phone, office) =
      (t.phone, t.office)
WHEN NOT MATCHED THEN
   INSERT
      (empid, phone, office)
      VALUES (t.empid, t.phone, t.office)
```

- *Example 5:* Using dynamically supplied values for an employee row, update the master employee table if the data corresponds to an existing employee, or insert the row if the data is for a new employee. The following example is a fragment of code from a C program.

```
hv1 =
"MERGE INTO employee AS t
USING TABLE(VALUES(CAST (? AS CHAR(6)), CAST (? AS VARCHAR(12)),
                   CAST (? AS CHAR(1)), CAST (? AS VARCHAR(15)),
                   CAST (? AS SMALLINT), CAST (? AS INTEGER)))
                   s(empno, firstnme, midinit, lastname, edlevel, salary)
ON t.empno = s.empno
WHEN MATCHED THEN
   UPDATE SET
      salary = s.salary
WHEN NOT MATCHED THEN
   INSERT
      (empno, firstnme, midinit, lastname, edlevel, salary)
      VALUES (s.empno, s.firstnme, s.midinit, s.lastname, s.edlevel,
              s.salary)";
EXEC SQL PREPARE s1 FROM :hv1;
EXEC SQL EXECUTE s1 USING '000420', 'SERGE', 'K', 'FIELDING', 18, 39580;
```

- *Example 6:* Update the list of activities organized by Group A in the archive table. Delete all outdated activities and update the activities information (description and date) in the archive table if they have been changed. For new upcoming activities, insert into the archive. Signal an error if the date of the activity is not known. The date of the activities in the archive table must be specified. Each group has an activities table. For example, activities_groupA contains all activities that they organize, and the archive table contains all upcoming activities organized by different groups in a company. The archive table has (group, activity) as the primary key, and date is not nullable. All activities tables have activity as the primary key. The last_modified column in the archive is defined with CURRENT TIMESTAMP as the default value.

```
MERGE INTO archive ar
USING (SELECT activity, description, date, last_modified
       FROM activities_groupA) ac
ON (ar.activity = ac.activity) AND ar.group = 'A'
WHEN MATCHED AND ac.date IS NULL THEN
   SIGNAL SQLSTATE '70001'
      SET MESSAGE_TEXT =
         ac.activity CONCAT ' cannot be modified. Reason: Date is not known'
WHEN MATCHED AND ac.date < CURRENT DATE THEN
   DELETE
WHEN MATCHED AND ar.last_modified < ac.last_modified THEN
   UPDATE SET
      (description, date, last_modified) = (ac.description, ac.date, DEFAULT)
WHEN NOT MATCHED AND ac.date IS NULL THEN
   SIGNAL SQLSTATE '70002'
      SET MESSAGE_TEXT =
         ac.activity CONCAT ' cannot be inserted. Reason: Date is not known'
WHEN NOT MATCHED AND ac.date >= CURRENT DATE THEN
   INSERT
      (group, activity, description, date)
```

```
        VALUES ('A', ac.activity, ac.description, ac.date)
  ELSE IGNORE
```

OPEN

The OPEN statement opens a cursor so that it can be used to fetch rows from its result table.

Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared. When invoked using the command line processor, some options cannot be specified.

For more information, refer to "Using command line SQL statements and XQuery statements" in *Command Reference*.

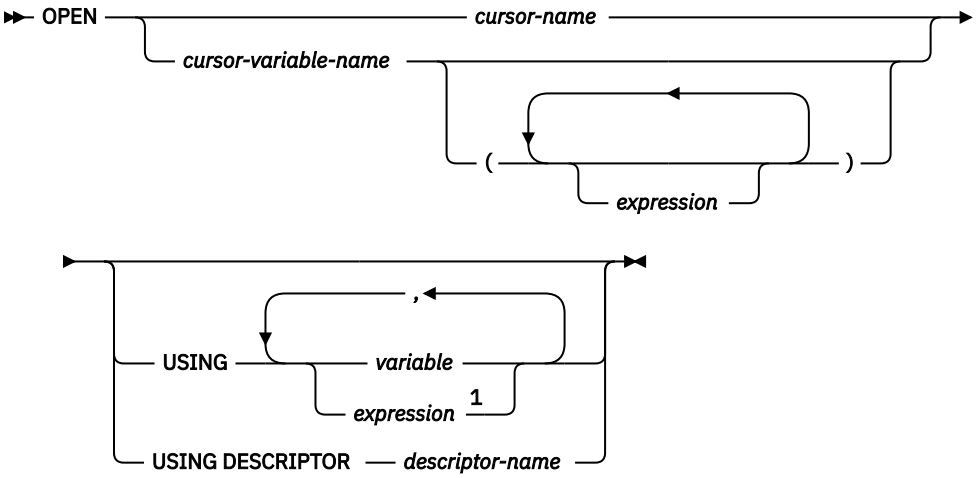
Authorization

If the *cursor-variable-name* references a global variable, then the privileges held by the authorization ID of the statement must include one of the following authorities:

- READ privilege on the global variable that is not defined in a module
- EXECUTE privilege on the module of the global variable that is defined in a module
- EXECUTEIN privilege on the schema containing the module of the global variable that is defined in a module
- DATAACCESS authority on the schema containing the module of the global variable that is defined in a module

Group privileges are not considered because this statement cannot be dynamically prepared.

Syntax



Notes:

[1] An expression other than a variable can only be used in compiled compound statements.

Description

cursor-name

Names a cursor that is defined in a DECLARE CURSOR statement that was stated earlier in the program. If *cursor-name* identifies a cursor in an SQL procedure declared as WITH RETURN TO CLIENT that is already in the open state, the existing open cursor becomes a result set cursor that is no longer accessible using cursor-name and a new cursor is opened that becomes accessible using

cursor-name. Otherwise, when the OPEN statement is executed, the cursor identified by *cursor-name* must be in the closed state.

The DECLARE CURSOR statement must identify a SELECT statement, in one of the following ways:

- Including the SELECT statement in the DECLARE CURSOR statement
- Including a *statement-name* that names a prepared SELECT statement.

The result table of the cursor is derived by evaluating the SELECT statement. The evaluation uses the current values of any special registers, global variables, or PREVIOUS VALUE expressions specified in the SELECT statement, and the current values of any host variables specified in the SELECT statement or the USING clause of the OPEN statement. The rows of the result table may be derived during the execution of the OPEN statement, and a temporary table may be created to hold them; or they may be derived during the execution of subsequent FETCH statements. In either case, the cursor is placed in the open state and positioned before the first row of its result table. If the table is empty, the state of the cursor is effectively "after the last row".

cursor-variable-name

Names a cursor variable. The value of the cursor variable must not be null (SQLSTATE 34000). A cursor variable that is directly or indirectly assigned a cursor value constructor can be used only in an OPEN statement that is in the same scope as the assignment (SQLSTATE 51044). If the cursor value constructor assigned to the cursor variable specified a *statement-name*, the OPEN statement must be in the same scope where that *statement-name* was explicitly or implicitly declared (SQLSTATE 51044).

When the OPEN statement is executed, the underlying cursor of the cursor variable must be in the closed state. The result table of the underlying cursor is derived by evaluating the SELECT statement or dynamic statement associated with the cursor variable. The evaluation uses the current values of any special registers, global variables, or PREVIOUS VALUE expressions specified in the SELECT statement, and the current values of any variables specified in the SELECT statement or the USING clause of the OPEN statement. The rows of the result table may be derived during the execution of the OPEN statement, and a temporary table may be created to hold them; or they may be derived during the execution of subsequent FETCH statements. In either case, the cursor is placed in the open state and positioned before the first row of its result table. If the table is empty, the state of the cursor is effectively "after the last row".

An OPEN statement using a *cursor-variable-name* can only be used within a compound SQL (compiled) statement.

(expression, ...)

Specifies the arguments associated with the named parameters of a parameterized cursor variable. The *cursor-value-constructor* assigned to the cursor variable must include a list of parameters with the same number of parameters as the number of arguments specified (SQLSTATE 07006 or 07004). The data type and value of the *n*th expression must be assignable to the *n*th parameter (SQLSTATE 07006 or 22018).

USING

Introduces the values that are substituted for the parameter markers or variables in the statement of the cursor. For an explanation of parameter markers, see "PREPARE".

If a *statement-name* is specified in the DECLARE CURSOR statement or the cursor value constructor associated with the cursor variable that includes parameter markers, USING must be used. If the prepared statement does not include parameter markers, USING is ignored.

If a *select-statement* is specified in the DECLARE CURSOR statement or the non-parameterized cursor value constructor associated with the cursor variable, USING may be used to override the variable values.

variable

Identifies a variable or a host structure declared in the program in accordance with the rules for declaring variables and host variables. The number of variables must be the same as the number of parameter markers in the prepared statement. The *n*th variable corresponds to the *n*th parameter

marker in the prepared statement. Where appropriate, locator variables and file reference variables can be provided as the source of values for parameter markers.

expression
Specifies values to associate with parameter markers using expressions. An OPEN statement that specifies expressions in the USING clause can only be used within a compound SQL (compiled) statement (SQLSTATE 42601). The number of expressions must be the same as the number of parameter markers in the prepared statement (SQLSTATE 07001). The *n*th expression corresponds to the *n*th parameter marker in the prepared statement. The data type and value of the nth expression must be assignable to the type associated with the nth parameter marker (SQLSTATE 07006).

Rules

- When the SELECT statement of the cursor is evaluated, each parameter marker in the statement is effectively replaced by its corresponding host variable. For a typed parameter marker, the attributes of the target variable are those specified by the CAST specification. For an untyped parameter marker, the attributes of the target variable are determined according to the context of the parameter marker.
- Let V denote a host variable that corresponds to parameter marker P. The value of V is assigned to the target variable for P in accordance with the rules for assigning a value to a column. Thus:
  - V must be compatible with the target.
  - If V is a string, its length (excluding trailing blanks for strings that are not long strings) must not be greater than the length attribute of the target.
  - If V is a number, the absolute value of its integral part must not be greater than the maximum absolute value of the integral part of the target.
  - If the attributes of V are not identical to the attributes of the target, the value is converted to conform to the attributes of the target.

  When the SELECT statement of the cursor is evaluated, the value used in place of P is the value of the target variable for P. For example, if V is CHAR(6), and the target is CHAR(8), the value used in place of P is the value of V padded with two blanks.
- The USING clause is intended for a prepared SELECT statement that contains parameter markers. However, it can also be used when the SELECT statement of the cursor is part of the DECLARE CURSOR statement or the non-parameterized cursor value constructor associated with the cursor variable. In this case the OPEN statement is executed as if each host variable in the SELECT statement were a parameter marker, except that the attributes of the target variables are the same as the attributes of the host variables in the SELECT statement. The effect is to override the values of the host variables in the SELECT statement of the cursor with the values of the host variables specified in the USING clause. A variable value override must not be used when opening a parameterized cursor variable since the SELECT statement will not include any other variables.
- SQL data change statements and routines that modify SQL data embedded in the cursor definition are completely executed, and the result set is stored in a temporary table when the cursor opens. If statement execution is successful, the SQLERRD(3) field contains the sum of the number of rows that qualified for insert, update, and delete operations. If an error occurs during execution of an OPEN statement involving a cursor that contains a data change statement within a fullselect, the results of that data change statement are rolled back.

  Explicit rollback of an OPEN statement, or rollback to a savepoint before an OPEN statement, closes the cursor. If the cursor definition contains a data change statement within the FROM clause of a fullselect, the results of the data change statement are rolled back.

  Changes to rows in a table that is targeted by a data change statement nested within a SELECT statement or a SELECT INTO statement are processed when the cursor opens, and are not undone if an error occurs during a fetch operation against that cursor.

Notes

- ***Closed state of cursors:*** All cursors in a program are in the closed state when the program is initiated and when it initiates a ROLLBACK statement.

  All cursors, except open cursors declared WITH HOLD, are in a closed state when a program issues a COMMIT statement.

  A cursor can also be in the closed state because a CLOSE statement was executed or an error was detected that made the position of the cursor unpredictable.

  The underlying cursor of a cursor variable is closed if the cursor variable goes out of scope and there are no other cursor variables that referenced that underlying cursor.
- To retrieve rows from the result table of a cursor, execute a FETCH statement when the cursor is open. The only way to change the state of a cursor from closed to open is to execute an OPEN statement.
- ***Effect of materialized result tables:*** In some cases, such as when the cursor is not read only, the result rows of a cursor are derived during the execution of FETCH statements. In other cases, the materialized result table method is used instead. With the materialized result table method the entire result table is transferred to a temporary buffer during the execution of the OPEN statement. When a temporary buffer is used, the results of a program can differ in these ways:
  - An error can occur during OPEN that would otherwise not occur until some later FETCH statement.
  - INSERT, UPDATE, and DELETE statements executed in the same transaction while the cursor is open cannot affect the result table.
  - Any NEXT VALUE expressions in the SELECT statement are evaluated for every row of the result table during OPEN.

  Conversely, if a temporary buffer is not used, INSERT, UPDATE, and DELETE statements executed while the cursor is open can affect the result table if issued from the same unit of work, and any NEXT VALUE expressions in the SELECT statement are evaluated as each row is fetched. This result table can also be affected by operations executed by the same unit of work, and the effect of such operations is not always predictable. For example, if cursor C is positioned on a row of its result table defined as SELECT * FROM T, and a new row is inserted into T, the effect of that insert on the result table is not predictable because its rows are not ordered. Thus a subsequent FETCH C may or may not retrieve the new row of T.
- Statement caching affects cursors declared open by the OPEN statement.
- ***Opening the same cursor multiple times:*** A cursor in an SQL procedure declared as WITH RETURN TO CLIENT can be opened even when a cursor with the same name is already in the open state. In this case, the existing open cursor becomes a result set cursor and is no longer accessible by its cursor name. A new cursor is opened and becomes accessible by the cursor name. Closing the new cursor does not make the cursor that was previously accessible by that name accessible by the cursor name again. The cursors that become result set cursors in this way cannot be accessed at the server and can be processed only at the client.
- When an SQL procedure, or an external stored procedure that uses non-blocking cursors, opens a cursor with return but does not fetch any rows from it, the access plan of the cursor query might not be evaluated. The query is evaluated when a row is fetched from the procedure's result set by the caller or client.

Examples

Example 1: Write the embedded statements in a COBOL program that will:

1. Define a cursor C1 that is to be used to retrieve all rows from the DEPARTMENT table for departments that are administered by (ADMRDEPT) department 'A00'.
2. Place the cursor C1 before the first row to be fetched.

```
EXEC SQL  DECLARE C1 CURSOR FOR
                SELECT DEPTNO, DEPTNAME, MGRNO
                  FROM DEPARTMENT
                  WHERE ADMRDEPT = 'A00'
END-EXEC.
```

```
EXEC SQL  OPEN C1
END-EXEC.
```

Example 2: Code an OPEN statement to associate a cursor DYN_CURSOR with a dynamically defined select-statement in a C program. Assuming two parameter markers are used in the predicate of the select-statement, two host variable references are supplied with the OPEN statement to pass integer and varchar(64) values between the application and the database. (The related host variable definitions, PREPARE statement, and DECLARE CURSOR statement are also shown in this example.)

```
EXEC SQL  BEGIN DECLARE SECTION;
  static short    hv_int;
  char            hv_vchar64[65];
  char            stmt1_str[200];
EXEC SQL  END DECLARE SECTION;

EXEC SQL  PREPARE STMT1_NAME FROM :stmt1_str;
EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;

EXEC SQL  OPEN DYN_CURSOR USING :hv_int, :hv_vchar64;
```

Example 3: Code an OPEN statement as in example 2, but in this case the number and data types of the parameter markers in the WHERE clause are not known.

```
EXEC SQL  BEGIN DECLARE SECTION;
  char    stmt1_str[200];
EXEC SQL  END DECLARE SECTION;
EXEC SQL  INCLUDE SQLDA;

EXEC SQL  PREPARE STMT1_NAME FROM :stmt1_str;
EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;

EXEC SQL  OPEN DYN_CURSOR USING DESCRIPTOR :sqlda;
```

Example 4: Create a procedure that does the following operations:

1. Assigns a cursor to the output cursor variable
2. Opens the cursor

```
CREATE PROCEDURE PROC1 (OUT P1 CURSOR)LANGUAGE SQL
BEGIN
SET P1=CURSOR FOR SELECT DEPTNO, DEPTNAME, MGRNO FROM DEPARTMENT WHERE
ADMRDEPT='A00'; --
OPEN P1; --
END;
```

PIPE

The PIPE statement is used to return a row from a compiled table function.

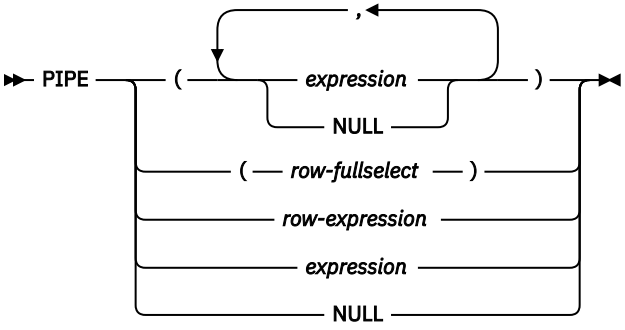
Invocation

This statement can be embedded in a compound SQL (compiled) statement of an SQL table function. It is not an executable statement and cannot be dynamically prepared.

Authorization

No privileges are required to invoke the PIPE statement. However, the authorization ID of the statement must hold the necessary privileges to invoke any expression that is embedded in the PIPE statement.

Syntax



Description

(*expression, ...*)
Specifies a row value is returned from the function. The number of expressions (or NULL keywords) in the list must match the RETURNS data type of the function and the value of each expression must be assignable to the corresponding column or field in the RETURNS data type of the function.

row-fullselect
Specifies a fullselect that returns a single row with the number of columns corresponding to the number of columns or fields in the RETURNS data type of the function. The value in each column of the row returned by the fullselect must be assignable to the corresponding column or field in the RETURNS data type of the function. If the result of the row fullselect is no rows, null values are returned.

row-expression
Specifies the row value is returned from the function. The number of fields in the row must match the RETURNS data type of the function and each field in the row must be assignable to the corresponding field in the RETURNS data type of the function. If the *row-expression* and the RETURNS data type are user-defined row types, the type names must be the same (SQLSTATE 42821).

expression
Specifies a scalar value is returned from the function. The RETURNS data type of the table function must have a single column and the expression value must be assignable to that column.

NULL
Specifies that a null value is returned from the function. A null value is returned for each column or row field.

Notes

- **Locally declared procedures:** The PIPE statement cannot be used within a procedure that is locally declared in the compound SQL (compiled) statement of an SQL table function.
- **Similar terms:** An SQL table function that uses a PIPE statement is sometimes referred to as a *pipelined* function.

Example

Create a table function called NEXT52 that returns a week number and date for the same day of the week for the next 52 weeks, along with the associated ISO week number.

```
CREATE OR REPLACE FUNCTION NEXT52 (START_TS TIMESTAMP)
   RETURNS TABLE (WEEKNUM SMALLINT, WEEKNUM_DATE DATE, ISO_WEEK SMALLINT)
BEGIN
   DECLARE WN INTEGER DEFAULT 1;
   DECLARE WND DATE;
   SET WND = START_TS;
   WHILE (WN < 53) DO
      SET WND = WND + 7 DAYS;
      PIPE (WN, WND, WEEK_ISO(WND));
      SET WN = WN + 1;
```

```
   END WHILE;
   RETURN;
END
```

PREPARE

The PREPARE statement is used by application programs to dynamically prepare an SQL statement for execution. The PREPARE statement creates an executable SQL statement, called a *prepared statement*, from a character string form of the statement, called a *statement string*.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

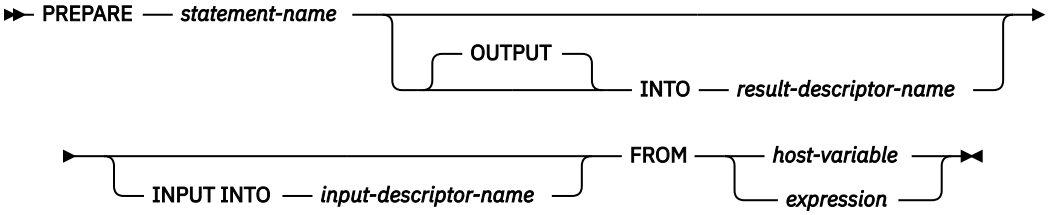
For statements where authorization checking is performed at statement preparation time (DML), the privileges held by the authorization ID of the statement must include those required to execute the SQL statement specified by the PREPARE statement. The authorization ID of the statement might be affected by the DYNAMICRULES bind option.

For statements where authorization checking is performed at statement execution time (DDL, GRANT, and REVOKE statements), no authorization is required to use this statement; however, the authorization is checked when the prepared statement is executed.

For statements involving tables that are protected with a security policy, the rules associated with the security policy are always evaluated at statement execution time.

If the authorization ID of the statement holds EXPLAIN, SQLADM, or DBADM authority, the user may prepare any statement; however, the ability to execute the statement is re-checked at statement execution time.

Syntax



Description

statement-name
Names the prepared statement. If the name identifies an existing prepared statement, that previously prepared statement is destroyed. The name must not identify a prepared statement that is the SELECT statement of an open cursor.

OUTPUT INTO
If OUTPUT INTO is used, and the PREPARE statement executes successfully, information about the output parameter markers in the prepared statement is placed in the SQLDA specified by *result-descriptor-name*.

result-descriptor-name
Specifies the name of an SQLDA. (The DESCRIBE statement may be used as an alternative to this clause.)

INPUT INTO

If INPUT INTO is used, and the PREPARE statement executes successfully, information about the input parameter markers in the prepared statement is placed in the SQLDA specified by *input-descriptor-name*. Input parameter markers are always considered nullable, regardless of usage.

input-descriptor-name

Specifies the name of an SQLDA. (The DESCRIBE statement may be used as an alternative to this clause.)

FROM

Introduces the statement string. The statement string is the value of the specified host variable.

host-variable

Specifies a host variable that is described in the program in accordance with the rules for declaring character string variables. It must be a fixed-length or varying-length character-string variable that is less than the maximum statement size of 2 097 152 bytes. Note that a CLOB(2097152) can contain a maximum size statement, but a VARCHAR cannot.

expression

An expression specifying the statement string. The expression must return a fixed-length or varying-length character-string type that is less than the maximum statement size of 2 097 152 bytes.

Rules

- ***Rules for statement strings:*** The statement string must be an executable statement that can be dynamically prepared. It must be one of the following SQL statements:
  - ALTER
  - CALL
  - COMMENT
  - COMMIT
  - Compound SQL (compiled)
  - Compound SQL (inlined)
  - CREATE
  - DECLARE GLOBAL TEMPORARY TABLE
  - DELETE
  - DROP
  - EXPLAIN
  - FLUSH EVENT MONITOR
  - FLUSH PACKAGE CACHE
  - GRANT
  - INSERT
  - LOCK TABLE
  - MERGE
  - REFRESH TABLE
  - RELEASE SAVEPOINT
  - RENAME
  - REVOKE
  - ROLLBACK
  - SAVEPOINT
  - select-statement
  - SET COMPILATION ENVIRONMENT

- SET CURRENT DECFLOAT ROUNDING MODE
- SET CURRENT DEFAULT TRANSFORM GROUP
- SET CURRENT DEGREE
- SET CURRENT EXPLAIN MODE
- SET CURRENT EXPLAIN SNAPSHOT
- SET CURRENT FEDERATED ASYNCHRONY
- SET CURRENT IMPLICIT XMLPARSE OPTION
- SET CURRENT ISOLATION
- SET CURRENT LOCALE LC_MESSAGES
- SET CURRENT LOCALE LC_TIME
- SET CURRENT LOCK TIMEOUT
- SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
- SET CURRENT MDC ROLLOUT MODE
- SET CURRENT OPTIMIZATION PROFILE
- SET CURRENT QUERY OPTIMIZATION
- SET CURRENT REFRESH AGE
- SET CURRENT TEMPORAL BUSINESS_TIME
- SET CURRENT TEMPORAL SYSTEM_TIME
- SET ENCRYPTION PASSWORD
- SET EVENT MONITOR STATE (only if DYNAMICRULES run behavior is in effect for the package)
- SET INTEGRITY
- SET PASSTHRU
- SET PATH
- SET ROLE (only if DYNAMICRULES run behavior is in effect for the package)
- SET SCHEMA
- SET SERVER OPTION
- SET SESSION AUTHORIZATION
- SET SQL_CCFLAGS
- SET USAGE LIST STATE (only if DYNAMICRULES run behavior is in effect for the package)
- SET variable
- TRANSFER OWNERSHIP (only if DYNAMICRULES run behavior is in effect for the package)
- TRUNCATE (only if DYNAMICRULES run behavior is in effect for the package)
- UPDATE

Notes

- ***Parameter markers:*** Although a statement string cannot include references to host variables, it can include *parameter markers*. These can be replaced by the values of host variables when the prepared statement is executed. In the case of a CALL statement, a parameter marker can also be used for OUT and INOUT arguments to the procedure. After the CALL is executed, the returned value for the argument will be assigned to the host variable corresponding to the parameter marker.

  A parameter marker is a question mark (?) or a colon followed by a name (:name) that is used where a host variable could be used if the statement string were a static SQL statement. For an explanation of how parameter markers are replaced by values, see "OPEN" and "EXECUTE".

  If the parameter marker is named, the name can include letters, numbers, and the symbols @, #, $, and _. The name is not folded to upper case.

Named parameter markers have the same syntax as host variables, but the two are not interchangeable. A host variable has a value and is used directly in a static SQL statement. A named parameter marker is a placeholder for a value in a dynamic SQL statement and the value is provided when the statement is executed.

There are two types of parameter markers:

Typed parameter marker
A parameter marker that is specified along with its target data type. It has the general form:

```
   CAST(? AS data-type)
```

This notation is not a function call, but a "promise" that the type of the parameter at run time will be of the data type specified or some data type that can be converted to the specified data type. For example, in:

```
  UPDATE EMPLOYEE
SET LASTNAME = TRANSLATE(CAST(? AS VARCHAR(12))
WHERE EMPNO = ?
```

the value of the argument of the TRANSLATE function will be provided at run time. The data type of that value will either be VARCHAR(12), or some type that can be converted to VARCHAR(12).

Untyped parameter marker
A parameter marker that is specified without its target data type. It has the form of a single question mark. The data type of an untyped parameter marker is provided by context. For example, the untyped parameter marker in the predicate of the previous update statement is the same as the data type of the EMPNO column.

Typed parameter markers can be used in dynamic SQL statements wherever a host variable is supported and the data type is based on the promise made in the CAST function.

Untyped parameter markers can be used in dynamic SQL statements as long as the data type of the parameter marker can be derived based on the context in the SQL statement (SQLSTATE 42610).

The following example results in an error since in the first context, *c1* would resolve to a string data type, but in the second context, *c1* would resolve to a numeric data type:

```
SELECT 'Hello' || c1, 5 + c1 FROM (VALUES(?)) AS T(c1)
```

However, the following statement is successful since the parameter marker associated with the derived column, *c1*, would resolve to a numeric data type for both contexts:

```
SELECT 7 + c1, 5 + c1 FROM (VALUES(?)) AS T(c1)
```

See "Determining data types of untyped expressions" for the rules for typing an untyped parameter marker.

- When a PREPARE statement is executed, the statement string is parsed and checked for errors. If the statement string is invalid, the error condition is reported in the SQLCA. Any subsequent EXECUTE or OPEN statement that references this statement will also receive the same error (due to an implicit prepare done by the system) unless the error has been corrected.
- Prepared statements can be referred to in the following kinds of statements, with the restrictions shown:

In...
The prepared statement...

DESCRIBE
can be any statement

DECLARE CURSOR
must be SELECT

EXECUTE
must *not* be SELECT

- A prepared statement can be executed many times. Indeed, if a prepared statement is not executed more than once and does not contain parameter markers, it is more efficient to use the EXECUTE IMMEDIATE statement rather than the PREPARE and EXECUTE statements.
- All prepared statements created by a unit of work remain in a prepared state until the application terminates, with the following exceptions:
  - A statement that is prepared within a package bound with KEEPDYNAMC NO and which is not used by an open cursor declared with the WITH HOLD option is no longer in a prepared state when the unit of work ends.
  - A dynamic statement that is bound with KEEPDYNAMIC NO and which is used by an open cursor declared with the WITH HOLD option is in a prepared state until the next unit of work boundary where the cursor is closed.

Examples

Example 1: Prepare and execute a non-select-statement in a COBOL program. Assume the statement is contained in a host variable HOLDER and that the program will place a statement string into the host variable based on some instructions from the user. The statement to be prepared does not have any parameter markers.

```
EXEC SQL  PREPARE STMT_NAME FROM :HOLDER
END-EXEC.
EXEC SQL  EXECUTE STMT_NAME
END-EXEC.
```

Example 2: Prepare and execute a non-select-statement as in example 1, except code it for a C program. Also assume the statement to be prepared can contain any number of parameter markers.

```
EXEC SQL  PREPARE STMT_NAME FROM :holder;
EXEC SQL  EXECUTE STMT_NAME USING DESCRIPTOR :insert_da;
```

Assume that the following statement is to be prepared:

```
INSERT INTO DEPT VALUES(?, ?, ?, ?)
```

The columns in the DEPT table are defined as follows:

```
DEPT_NO   CHAR(3) NOT NULL, -- department number
DEPTNAME  VARCHAR(29), -- department name
MGRNO     CHAR(6), -- manager number
ADMRDEPT  CHAR(3)  -- admin department number
```

To insert department number G01 named COMPLAINTS, which has no manager and reports to department A00, the structure INSERT_DA should have the values in Table 152 on page 1756 before issuing the EXECUTE statement.

Table 152. Required values for the INSERT_DA structure

SQLDA field	**Value**
SQLDAID	SQLDA
SQLDABC	192 (See note 1.)
SQLN	4
SQLD	4
SQLTYPE	452
SQLLEN	3
SQLDATA	*pointer to* G01
SQLIND	(See note 2.)

Table 152. Required values for the INSERT_DA structure (continued)	
SQLDA field	**Value**
SQLNAME	
SQLTYPE SQLLEN SQLDATA SQLIND SQLNAME	449 29 *pointer to* COMPLAINTS *pointer to* 0
SQLTYPE SQLLEN SQLDATA SQLIND SQLNAME	453 6 (See note 3.) *pointer to* -1
SQLTYPE SQLLEN SQLDATA SQLIND SQLNAME	453 3 *pointer to* A00 *pointer to* 0
Note: 1. This value is for a PREPARE done from a 32-bit application. If the PREPARE was done in a 64-bit application, then SQLDABC would have the value 240. 2. The value in SQLIND for this SQLVAR is ignored because the SQLTYPE identifies a non-nullable data type. 3. The value in SQLDATA for this SQLVAR is ignored because the value of SQLIND indicates this is a null value.	

REFRESH TABLE

The REFRESH TABLE statement refreshes the data in a materialized query table.

Invocation

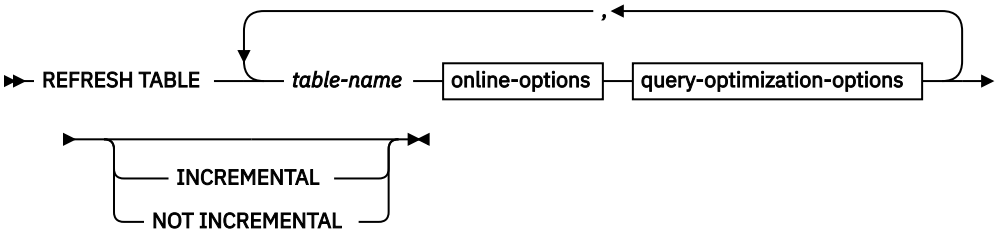
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

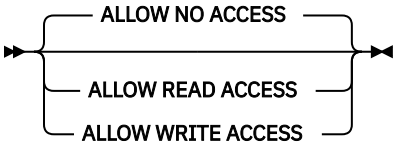
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- CONTROL privilege on the table
- DATAACCESS authority on the schema containing the materialized query table
- DATAACCESS authority

Syntax



online-options



query-optimization-options

Description

table-name

Identifies the table to be refreshed.

The name, including the implicit or explicit schema, must identify a table that already exists at the current server. The table must allow the REFRESH TABLE statement (SQLSTATE 42809). This includes materialized query tables defined with:

- REFRESH IMMEDIATE
- REFRESH DEFERRED

online-options

Specifies the accessibility of the table while it is being processed.

ALLOW NO ACCESS
Specifies that no other users can access the table while it is being refreshed, except if they are using the Uncommitted Read isolation level.

ALLOW READ ACCESS
Specifies that other users have read-only access to the table while it is being refreshed.

ALLOW WRITE ACCESS
Specifies that other users have read and write access to the table while it is being refreshed.

To prevent a rollback of the entire statement because of a lock timeout when using the ALLOW READ ACCESS or the ALLOW WRITE ACCESS option, it is recommended that you issue a SET CURRENT LOCK TIMEOUT statement (specifying the WAIT option) before executing the REFRESH TABLE statement, and to reset the special register to its previous value afterwards. Note, however, that the CURRENT LOCK TIMEOUT register only impacts a specific set of lock types, not all lock types.

query-optimization-options

Specifies the query optimization options for the refresh of REFRESH DEFERRED materialized query tables.

ALLOW QUERY OPTIMIZATION USING REFRESH DEFERRED TABLES WITH REFRESH AGE ANY
Specifies that when the CURRENT REFRESH AGE special register is set to 'ANY', the refresh of *table-name* will allow REFRESH DEFERRED materialized query tables to be used to optimize the query that is used to refresh *table-name*. If *table-name* is not a REFRESH DEFERRED materialized

query table, an error is returned (SQLSTATE 428FH). REFRESH IMMEDIATE materialized query tables are always considered for query optimization.

INCREMENTAL
Specifies an incremental refresh for the table by considering only the delta portion (if any) of its underlying tables or the content of an associated staging table (if one exists and its contents are consistent). If such a request cannot be satisfied (that is, the system detects that the materialized query table definition needs to be fully recomputed), an error (SQLSTATE 55019) is returned.

NOT INCREMENTAL
Specifies a full refresh for the table by recomputing the materialized query table definition.

If neither INCREMENTAL nor NOT INCREMENTAL is specified, the system will determine whether incremental processing is possible; if not, full refresh will be performed. If a staging table is present for the materialized query table that is to be refreshed, and incremental processing is not possible because the staging table is in a pending state, an error is returned (SQLSTATE 428A8). Full refresh will be performed if the staging table or the materialized query table is in an inconsistent state; otherwise, the contents of the staging table will be used for incremental processing.

Rules

- If REFRESH TABLE is issued on a materialized query table that references one or more nicknames, the authorization ID of the statement must have authority to select from the tables at the data source or from all schemas of the tables at the data source (SQLSTATE 42501).
- The NOT INCREMENTAL clause must be used if REFRESH TABLE is issued on a system-maintained column-organized MQT.

Notes

- When the statement is used to refresh a REFRESH IMMEDIATE materialized query table whose underlying tables have been loaded, attached, or detached, the system might choose to incrementally refresh the materialized query table with the delta portions of its underlying tables. When the statement is used to refresh a REFRESH DEFERRED materialized query table with a supporting staging table, the system might choose to incrementally refresh the materialized query table with the delta portions of its underlying tables that have been captured in the staging table. However, there are some situations in which this optimization is not possible, and a full refresh (that is, a recomputation of the materialized query table definition) is necessary to ensure data integrity. You can explicitly request incremental maintenance by specifying the INCREMENTAL option; if this optimization is not possible, the system returns an error (SQLSTATE 55019).
- If the ALLOW QUERY OPTIMIZATION USING REFRESH DEFERRED TABLES WITH REFRESH AGE ANY option is used, ensure that the refresh order is correct for REFRESH DEFERRED materialized query tables. For example, consider two materialized query tables, MQT1 and MQT2, whose materialized queries share the same underlying tables. The materialized query for MQT2 can be calculated using MQT1, instead of the underlying tables. If separate statements are used to refresh these two materialized query tables, and MQT2 is refreshed first, the system might choose to use the contents of MQT1, which have not yet been refreshed, to refresh MQT2. In this case, MQT1 would contain current data, but MQT2 could still contain stale data, even though both were refreshed at almost the same time. The correct refresh order, if two REFRESH statements are used instead of one, is to refresh MQT1 first.
- If the materialized query table has an associated staging table, the staging table is pruned when the refresh is successfully performed.
- Any label-based access control on the base tables or on the materialized query table does not interfere with the refresh process. The refresh happens as if label-based access control were not present. The automatic protection that is associated with the materialized query table when it is created ensures that the data from the base tables remains protected when it is passed into the materialized query table.
- For materialized query table only, SET INTEGRITY FOR *mqt_name* IMMEDIATE CHECKED is the same as REFRESH TABLE *mqt_name*.

- **Refresh use of materialized query tables:** Materialized query tables are not used to evaluate the *select-statement* during the processing of the REFRESH TABLE statement.
- **Refresh isolation level:** The isolation level used to evaluate the *select-statement* is the isolation level specified on the *isolation-level* clause of the *select-statement*. Or, if the *isolation-level* clause was not specified, the isolation level of the materialized query table recorded when CREATE TABLE or ALTER TABLE was issued is used to evaluate the *select-statement*.
- Consider the statement:

  ```
  SET INTEGRITY FOR T IMMEDIATE CHECKED
  ```

  In the following scenarios, neither the INCREMENTAL check option for T nor an incremental refresh of T---if T is a materialized query table (MQT) or a staging table---is supported:

  – New constraints have been added to T while it is in set integrity pending state
  – When a LOAD REPLACE operation against T, it parents, or its underlying tables has taken place
  – When the NOT LOGGED INITIALLY WITH EMPTY TABLE option has been activated after the last integrity check on T, its parents, or its underlying tables
  – The cascading effect of full processing, when any parent of T (or underlying table, if T is a materialized query table or a staging table) has been checked for integrity non-incrementally
  – If the table space containing the table or its parent (or underlying table of a materialized query table or a staging table) has been rolled forward to a point in time, and the table and its parent (or underlying table if the table is a materialized query table or a staging table) reside in different table spaces
  – T is an MQT, and a LOAD REPLACE or LOAD INSERT operation directly into T has taken place after the last refresh
- Incremental processing will be used whenever the situation allows it, because it is more efficient. The INCREMENTAL option is not needed in most cases. It is needed, however, to ensure that integrity checks are indeed processed incrementally. If the system detects that full processing is needed to ensure data integrity, an error is returned (SQLSTATE 55019).
- If the conditions for full processing described in the previous bullet are not satisfied, the system will perform an incremental refresh (if it is a materialized query table) when the user does not specify the NOT INCREMENTAL option for the statement `SET INTEGRITY FOR T IMMEDIATE CHECKED`.

RELEASE (connection)

The RELEASE (Connection) statement places one or more connections in the release-pending state.

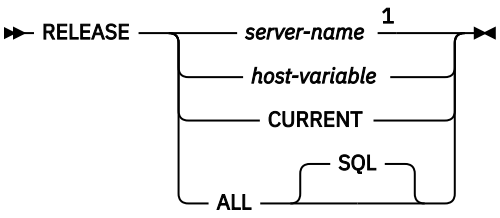
Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax



Notes:

[1] Note that an application server named CURRENT or ALL can only be identified by a host variable or a delimited identifier.

Description

server-name* or *host-variable

Identifies the application server by the specified *server-name* or a *host-variable* which contains the *server-name*.

If a *host-variable* is specified, it must be a character string variable with a length attribute that is not greater than 8, and it must not include an indicator variable. The *server-name* that is contained within the *host-variable* must be left-aligned and must not be delimited by quotation marks.

Note that the *server-name* is a database alias identifying the application server. It must be listed in the application requester's local directory.

The specified database-alias or the database-alias contained in the host variable must identify an existing connection of the application process. If the database-alias does not identify an existing connection, an error (SQLSTATE 08003) is raised.

CURRENT

Identifies the current connection of the application process. The application process must be in the connected state. If not, an error (SQLSTATE 08003) is raised.

ALL or ALL SQL

Identifies all existing connections of the application process. This form of the RELEASE statement places all existing connections of the application process in the release-pending state. All connections will therefore be destroyed during the next commit operation. An error or warning does not occur if no connections exist when the statement is executed.

Examples

- *Example 1:* The SQL connection to IBMSTHDB is no longer needed by the application. The following statement will cause it to be destroyed during the next commit operation:

```
EXEC SQL RELEASE IBMSTHDB;
```

- *Example 2:* The current connection is no longer needed by the application. The following statement will cause it to be destroyed during the next commit operation:

```
EXEC SQL RELEASE CURRENT;
```

- *Example 3:* If an application has no need to access the databases after a commit but will continue to run for a while, then it is better not to tie up those connections unnecessarily. The following statement can be executed before the commit to ensure all connections will be destroyed at the commit:

```
EXEC SQL RELEASE ALL;
```

RELEASE SAVEPOINT

The RELEASE SAVEPOINT statement is used to indicate that the application no longer wishes to have the named savepoint maintained. After this statement has been invoked, rollback to the savepoint is no longer possible.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax

Description

savepoint-name
Specifies the savepoint that is to be released. Any savepoints nested within the named savepoint are also released. Rollback to that savepoint, or any savepoint nested within it, is no longer possible. If the named savepoint does not exist in the current savepoint level (see the "Rules" section in the description of the SAVEPOINT statement), an error is returned (SQLSTATE 3B001). The specified *savepoint-name* cannot begin with 'SYS' (SQLSTATE 42939).

Notes

- The name of the savepoint that was released can now be reused in another SAVEPOINT statement, regardless of whether the UNIQUE keyword was specified on an earlier SAVEPOINT statement specifying this same savepoint name.

Example

Release a savepoint named SAVEPOINT1.

```
RELEASE SAVEPOINT SAVEPOINT1
```

RENAME

The RENAME statement renames an existing table or index.

Invocation

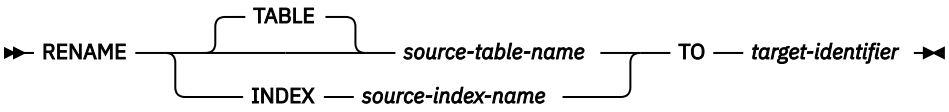
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- CONTROL privilege on the table or index
- Ownership of the table or index, as recorded in the OWNER column of the SYSCAT.TABLES catalog view for a table, and the SYSCAT.INDEXES catalog view for an index
- ALTERIN privilege on the schema
- SCHEMAADM authority on the schema
- DBADM authority

Syntax



Description

TABLE *source-table-name*

Names the existing table that is to be renamed. The name, including the schema name, must identify a table that already exists in the database (SQLSTATE 42704). It must not be the name of a catalog table (SQLSTATE 42832), a materialized query table, a typed table (SQLSTATE 42997), a created temporary table, a declared global temporary table (SQLSTATE 42995), a nickname, or an object other than a table or an alias (SQLSTATE 42809). The TABLE keyword is optional.

The name must not identify a table that is referenced in a row permission definition or a column mask definition (SQLSTATE 42917).

INDEX *source-index-name*

Names the existing index that is to be renamed. The name, including the schema name, must identify an index that already exists in the database (SQLSTATE 42704). It must not be the name of an index on a created temporary table or a declared global temporary table (SQLSTATE 42995). The schema name must not be SYSIBM, SYSCAT, SYSFUN, or SYSSTAT (SQLSTATE 42832).

target-identifier

Specifies the new name for the table or index without a schema name. The schema name of the source object is used to qualify the new name for the object. The qualified name must *not* identify a table, view, alias, or index that already exists in the database (SQLSTATE 42710).

Rules

When renaming a table, the source table must not:

- Be referenced in any existing materialized query table definitions
- Be referenced in any existing statistical view definition. This includes the system-generated statistical view that is created as part of index creation which includes an expression-based key.

  Note: With the release of Db2 11.5.7, renaming of tables with an index having an expression-based key is possible, if the expression does not contain qualified names.
- Be the subject table of an existing trigger
- Be a parent or dependent table in any referential integrity constraints
- Be the scope of any existing reference column
- Be referenced in a spatial registration with Spatial Extender. Unregister the spatial column or columns, and then reregister them after the rename is completed.
- Be referenced by an XSR object that has been enabled for decomposition

An error (SQLSTATE 42986) is returned if the source table violates one or more of these conditions.

When renaming an index:

- The source index must not be a system-generated index for an implementation table on which a typed table is based (SQLSTATE 42858).

Notes

- CHECK constraints with three part names are not supported and will return SQL0750. Use only the column name instead.
- Catalog entries are updated to reflect the new table or index name.

- *All* authorizations associated with the source table or index name are *transferred* to the new table or index name (the authorization catalog tables are updated appropriately).
- Indexes defined over the source table are *transferred* to the new table (the index catalog tables are updated appropriately).
- RENAME TABLE invalidates any packages that are dependent on the source table. RENAME INDEX invalidates any packages that are dependent on the source index.
- If an alias is used for the *source-table-name*, it must resolve to a table name. The alias is not changed by the RENAME statement and continues to refer to the old table name. The table is renamed within its original schema
- A table with primary key or unique constraints can be renamed if none of the primary key or unique constraints are referenced by any foreign key.

Examples

- *Example 1*: Change the name of the EMP table to EMPLOYEE.

```
RENAME TABLE EMP TO EMPLOYEE
RENAME TABLE ABC.EMP TO EMPLOYEE
```

- *Example 2*: Change the name of the index NEW-IND to IND.

```
RENAME INDEX NEW-IND TO IND
RENAME INDEX ABC.NEW-IND TO IND
```

RENAME STOGROUP

The RENAME STOGROUP statement renames an existing storage group.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include either SYSCTRL or SYSADM authority.

Syntax

Description

source-storagegroup-name
Identifies the storage group to rename; *source-storagegroup-name* must identify a storage group that exists at the current server (SQLSTATE 42704). This is a one-part name.

target-storagegroup-name
Names the storage group. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *target-storagegroup-name* must not identify a storage group that already exists in the catalog (SQLSTATE 42710). The *target-storagegroup-name* must not begin with the characters 'SYS' (SQLSTATE 42939).

Rules

- The RENAME STOGROUP statement cannot be executed while a database partition server is being added (SQLSTATE 55071).

RENAME TABLESPACE

The RENAME TABLESPACE statement renames an existing table space.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include either SYSCTRL or SYSADM authority.

Syntax

Description

source-tablespace-name
Specifies the existing table space that is to be renamed, as a one-part name. It is an SQL identifier (either ordinary or delimited). The table space name must identify a table space that already exists in the catalog (SQLSTATE 42704).

target-tablespace-name
Specifies the new name for the table space, as a one-part name. It is an SQL identifier (either ordinary or delimited). The new table space name must *not* identify a table space that already exists in the catalog (SQLSTATE 42710), and it cannot start with 'SYS' (SQLSTATE 42939).

Rules

- The SYSCATSPACE table space cannot be renamed (SQLSTATE 42832).
- Any table spaces with "rollforward pending" or "rollforward in progress" states cannot be renamed (SQLSTATE 55039)

Notes

- Renaming a table space will update the minimum recovery time of a table space to the point in time when the rename took place. This implies that a roll forward at the table space level must be to at least this point in time.
- The new table space name must be used when restoring a table space from a backup image, where the rename was done after the backup was created.

Example

Change the name of the table space USERSPACE1 to DATA2000:

```
RENAME TABLESPACE USERSPACE1 TO DATA2000
```

REPEAT

The REPEAT statement executes a statement or group of statements until a search condition is true.

Invocation

This statement can be embedded in an:

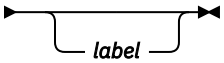
- SQL procedure definition
- Compound SQL (compiled) statement
- Compound SQL (inlined) statement

The compound statements can be embedded in an SQL procedure definition, SQL function definition, or SQL trigger definition. It is not an executable statement and cannot be dynamically prepared.

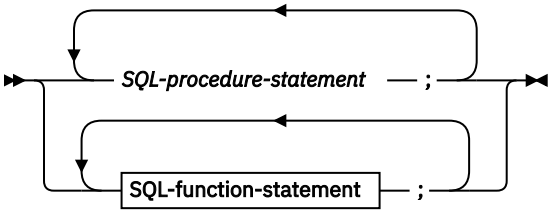
Authorization

No privileges are required to invoke the REPEAT statement. However, the authorization ID of the statement must hold the necessary privileges to invoke the SQL statements and search condition that are embedded in the REPEAT statement.

Syntax



SQL-routine-statement



Description

label
Specifies the label for the REPEAT statement. If the beginning label is specified, that label can be specified on LEAVE and ITERATE statements. If an ending label is specified, a matching beginning label also must be specified.

SQL-procedure-statement
Specifies the SQL statements to execute within the loop. *SQL-procedure-statement* is only applicable when in the context of an SQL procedure or a compound SQL (compiled) statement. See *SQL-procedure-statement* in "Compound SQL (compiled)" statement.

SQL-function-statement
Specifies the SQL statements to execute within the loop. *SQL-function-statement* is only applicable when in the context of an SQL trigger, SQL function, or SQL method. See *SQL-function-statement* in "FOR".

search-condition
The *search-condition* is evaluated after each execution of the REPEAT loop. If the condition is true, the loop will exit. If the condition is unknown or false, the looping continues.

Example

A REPEAT statement fetches rows from a table until the *not_found* condition handler is invoked.

```
CREATE PROCEDURE REPEAT_STMT(OUT counter INTEGER)
  LANGUAGE SQL
  BEGIN
    DECLARE v_counter INTEGER DEFAULT 0;
    DECLARE v_firstnme VARCHAR(12);
    DECLARE v_midinit CHAR(1);
    DECLARE v_lastname VARCHAR(15);
    DECLARE at_end SMALLINT DEFAULT 0;
    DECLARE not_found CONDITION FOR SQLSTATE '02000';
    DECLARE c1 CURSOR FOR
      SELECT firstnme, midinit, lastname
        FROM employee;
    DECLARE CONTINUE HANDLER FOR not_found
      SET at_end = 1;
    OPEN c1;
    fetch_loop:
    REPEAT
      FETCH c1 INTO v_firstnme, v_midinit, v_lastname;
      SET v_counter = v_counter + 1;
      UNTIL at_end > 0
    END REPEAT fetch_loop;
    SET counter = v_counter;
    CLOSE c1;
  END
```

RESIGNAL

The RESIGNAL statement is used within a condition handler to resignal the condition that activated the handler, or to raise an alternate condition so that it can be processed at a higher level. It causes an exception, warning, or not found condition to be returned, along with optional message text.

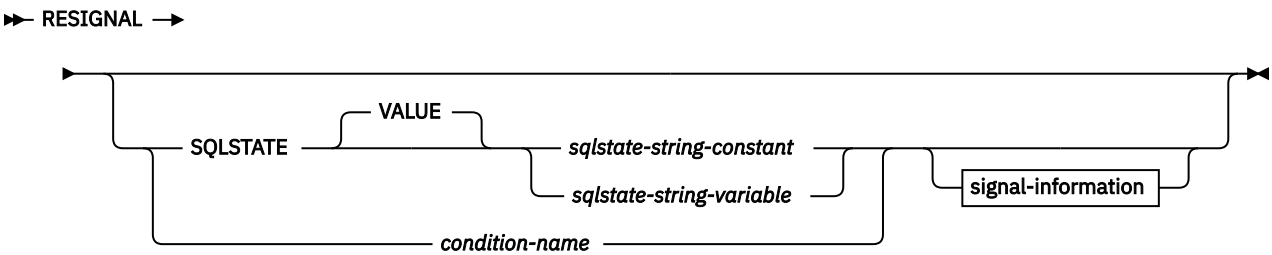
Invocation

This statement can only be embedded in a condition handler within a compound SQL (compiled) statement. The compound SQL (compiled) statement can be embedded in an SQL procedure definition, SQL function definition, or SQL trigger definition.

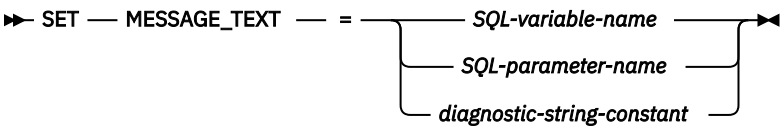
Authorization

If a module condition is referenced, the privileges held by the authorization ID of the statement must include EXECUTE privilege on the module or EXECUTEIN privilege or DATAACCESS on the schema containing the module.

Syntax



signal-information



Description

SQLSTATE VALUE *sqlstate-string-constant*

The specified string constant represents an SQLSTATE. It must be a character string constant with exactly 5 characters that follow the rules for SQLSTATEs:

- Each character must be from the set of digits ("0" through "9") or non-accented upper case letters ("A" through "Z")
- The SQLSTATE class (first two characters) cannot be "00", since this represents successful completion.

If the SQLSTATE does not conform to these rules, an error is raised (SQLSTATE 428B3).

SQLSTATE VALUE

Specifies the SQLSTATE that will be returned. Any valid SQLSTATE value can be used. The specified value must follow the rules for SQLSTATEs:

- Each character must be from the set of digits ("0" through "9") or upper case letters ("A" through "Z") without diacritical marks
- The SQLSTATE class (first two characters) cannot be "00", since this represents successful completion.

If the SQLSTATE does not conform to these rules, an error is returned.

sqlstate-string-constant

The *sqlstate-string-constant* must be a character string constant with exactly 5 characters.

sqlstate-string-variable

The specified SQL variable or SQL parameter must be of data type CHAR(5) and must not be the null value.

condition-name

Specifies the name of a condition that will be returned. The *condition-name* must be declared within the compound-statement or identify a condition that exists at the current server.

SET MESSAGE_TEXT =

Specifies a string that describes the error or warning. The string is returned in the `sqlerrmc` field of the SQLCA. If the actual string is longer than 70 bytes, it is truncated without warning.

SQL-variable-name

Identifies an SQL variable, declared within the compound statement, that contains the message text.

SQL-parameter-name

Identifies an SQL parameter, defined for the routine, that contains the message text. The SQL parameter must be defined as a CHAR or VARCHAR data type.

diagnostic-string-constant

Specifies a character string constant that contains the message text.

Notes

- If a RESIGNAL statement is issued without specifying an SQLSTATE clause or a *condition-name*, the identical condition that invoked the handler is returned. The SQLSTATE, SQLCODE and the SQLCA associated with the condition are unchanged.
- If a RESIGNAL statement is issued using a *condition-name* that has no associated SQLSTATE value and the condition is not handled, SQLSTATE 45000 is returned and the SQLCODE is set to -438. Note that such a condition will not be handled by a condition handler for SQLSTATE 45000 that is within the scope of the routine issuing the RESIGNAL statement.
- If a RESIGNAL statement is issued using an SQLSTATE value or a *condition-name* with an associated SQLSTATE value, the SQLCODE returned is based on the SQLSTATE value as follows:
  - If the specified SQLSTATE class is either "01" or "02" a warning or not found condition is returned and the SQLCODE is set to +438.

  - Otherwise, an exception condition is returned and the SQLCODE is set to -438.
- A RESIGNAL statement has the indicated fields of the SQLCA set as follows:
  - `sqlerrd` fields are set to zero
  - `sqlwarn` fields are set to blank
  - `sqlerrmc` is set to the first 70 bytes of MESSAGE_TEXT
  - `sqlerrml` is set to the length of `sqlerrmc`, or to zero if no SET MESSAGE_TEXT clause is specified
  - `sqlerrp` is set to ROUTINE
- Refer to the "Notes" section under "SIGNAL statement" for further information about SQLSTATE values.

Example

This example detects a division by zero error. The IF statement uses a SIGNAL statement to invoke the *overflow* condition handler. The condition handler uses a RESIGNAL statement to return a different SQLSTATE value to the client application.

```
CREATE PROCEDURE divide ( IN numerator INTEGER,
                          IN denominator INTEGER,
                          OUT result INTEGER)
LANGUAGE SQL
BEGIN
  DECLARE overflow CONDITION FOR SQLSTATE '22003';
  DECLARE CONTINUE HANDLER FOR overflow
    RESIGNAL SQLSTATE '22375';
  IF denominator = 0 THEN
    SIGNAL overflow;
  ELSE
    SET result = numerator / denominator;
  END IF;
END
```

RETURN

The RETURN statement is used to return from a routine. For SQL functions or methods, it returns the result of the function or method. For an SQL procedure, it optionally returns an integer status value.

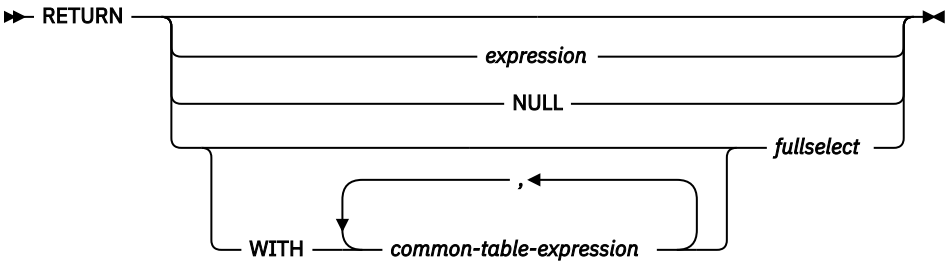
Invocation

This statement can be embedded in an SQL function, SQL method, or SQL procedure. It is not an executable statement and cannot be dynamically prepared.

Authorization

No privileges are required to invoke the RETURN statement. However, the authorization ID of the statement must hold the necessary privileges to invoke any expression or fullselect that is embedded in the RETURN statement.

Syntax



Description

expression
Specifies a value that is returned from the routine:

- If the routine is a function or method other than a compiled table function, one of *expression*, NULL, or *fullselect* must be specified (SQLSTATE 42631) and the data type of the result must be assignable to the RETURNS type of the routine (SQLSTATE 42866).
- If the routine is an inlined table function, a scalar expression (other than a scalar fullselect) cannot be specified (SQLSTATE 428F1). If the routine is a compiled table function, an expression cannot be specified.
- If the routine is a procedure, the data type of *expression* must be INTEGER (SQLSTATE 428F2). A procedure cannot return NULL or a *fullselect*.

NULL
Specifies that the function or method returns a null value of the data type defined in the RETURNS clause. NULL cannot be specified for a RETURN from a table function, row function, or procedure.

WITH *common-table-expression*
Defines a common table expression for use with the *fullselect* that follows.

fullselect
Specifies the row or rows to be returned for the function. The number of columns in the *fullselect* must match the number of columns in the function result (SQLSTATE 42811). In addition, the static column types of the *fullselect* must be assignable to the declared column types of the function result, using the rules for assignment to columns (SQLSTATE 42866).

The *fullselect* cannot be specified for a RETURN from a procedure or a compiled table function.

If the routine is a scalar function or method, then the *fullselect* must return one column (SQLSTATE 42823) and, at most, one row (SQLSTATE 21000).

If the routine is a row function, it must return, at most, one row (SQLSTATE 21505). However, one or more columns can be returned.

If the routine is an inlined table function, it can return zero or more rows with one or more columns. If the fullselect has zero result rows, no row is returned to the result table by the RETURN statement.

Rules

- The execution of an SQL function or method must end with a RETURN statement (SQLSTATE 42632).
- In an SQL table function using a compound SQL (compiled) statement, an *expression*, NULL, or *fullselect*cannot be specified. Rows are returned from the function using the PIPE statement and the RETURN statement is required as the last statement to execute when the function exits (SQLSTATE 2F005).
- In an SQL table or row function using a compound SQL (inlined) statement, the only RETURN statement allowed is the one at the end of the compound statement. (SQLSTATE 429BD).

Notes

- When a value is returned from a procedure, the caller can access the value:
  - using the GET DIAGNOSTICS statement to retrieve the DB2_RETURN_STATUS when the SQL procedure was called from another SQL procedure
  - using the parameter bound for the return value parameter marker in the escape clause CALL syntax (?=CALL...) in a CLI application
  - directly from the `sqlerrd[0]` field of the SQLCA, after processing the CALL of an SQL procedure. This field is only valid if the SQLCODE is zero or positive (assume a value of -1 otherwise).

Example

Use a RETURN statement to return from an SQL procedure with a status value of zero if successful, and -200 if not.

```
BEGIN
...
  GOTO FAIL;
...
  SUCCESS: RETURN 0;
  FAIL: RETURN -200;
END
```

REVOKE (database authorities)

This form of the REVOKE statement revokes authorities that apply to the entire database.

Invocation

This statement can be embedded in an application program or issued by using dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

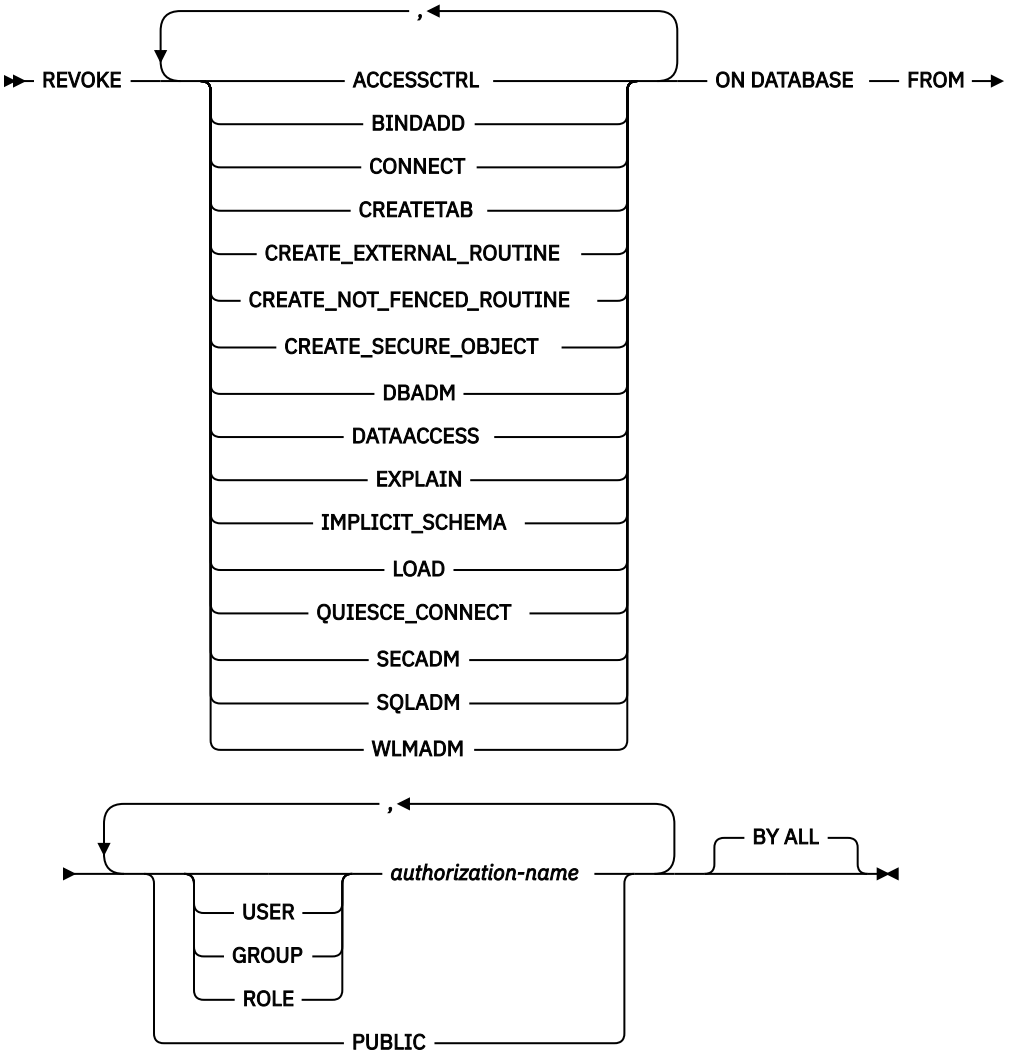
Authorization

To revoke ACCESSCTRL, CREATE_SECURE_OBJECT, DATAACCESS, DBADM, or SECADM authority, SECADM authority is needed.

Note: In Db2 11.5.7 and later, to revoke CREATE_EXTERNAL_ROUTINE or CREATE_NOT_FENCED_ROUTINE authority, SYSADM, SECADM or ACCESSCTRL authority is needed.

To revoke other authorities, ACCESSCTRL or SECADM authority is needed.

Syntax



Description

ACCESSCTRL
Revokes the authority to grant and revoke most database authorities and object privileges.

BINDADD
Revokes the authority to create packages. The creator of a package automatically has the CONTROL privilege on that package and retains this privilege even if the creator's BINDADD authority is later revoked.

The BINDADD authority cannot be revoked from an *authorization-name* holding DBADM authority without also revoking the DBADM authority.

CONNECT
Revokes the authority to access the database.

Revoking the CONNECT authority from a user does not affect any privileges that were granted to that user on objects in the database. If the user is again granted the CONNECT authority, all previously held privileges are still valid (assuming they were not explicitly revoked).

The CONNECT authority cannot be revoked from an *authorization-name* holding DBADM authority without also revoking the DBADM authority (SQLSTATE 42504).

CREATETAB
Revokes the authority to create tables. The creator of a table automatically has the CONTROL privilege on that table, and retains this privilege even if the creator's CREATETAB authority is later revoked.

The CREATETAB authority cannot be revoked from an *authorization-name* holding DBADM authority without also revoking the DBADM authority (SQLSTATE 42504).

CREATE_EXTERNAL_ROUTINE
Revokes the authority to register external routines. When an external routine is registered, it continues to exist, even if CREATE_EXTERNAL_ROUTINE is later revoked from the authorization ID that registered the routine.

CREATE_EXTERNAL_ROUTINE authority cannot be revoked from an *authorization-name* holding DBADM or CREATE_NOT_FENCED_ROUTINE authority without also revoking DBADM or CREATE_NOT_FENCED_ROUTINE authority (SQLSTATE 42504).

CREATE_NOT_FENCED_ROUTINE
Revokes the authority to register routines that run in the database manager's process. When a routine is registered as not fenced, it continues to run in this manner, even if CREATE_NOT_FENCED_ROUTINE is later revoked from the authorization ID that registered the routine.

CREATE_NOT_FENCED_ROUTINE authority cannot be revoked from an *authorization-name* holding DBADM authority without also revoking the DBADM authority (SQLSTATE 42504).

CREATE_SECURE_OBJECT
Revokes the authority to create secure triggers and secure functions. Revokes the authority to alter the secure attribute of such objects as well.

DATAACCESS
Revokes the authority to access data.

DBADM
Revokes the DBADM authority.

DBADM authority cannot be revoked from PUBLIC (because it cannot be granted to PUBLIC).

CAUTION: Revoking DBADM authority does not automatically revoke any privileges that were held by the authorization-name on objects in the database.

EXPLAIN
Revokes the authority to explain, prepare, and describe static and dynamic statements without requiring access to data.

IMPLICIT_SCHEMA
Revokes the authority to implicitly create a schema. It does not affect the ability to create objects in existing schemas or to process a CREATE SCHEMA statement.

IMPLICIT_SCHEMA authority cannot be revoked from an *authorization-name* holding DBADM authority without also revoking the DBADM authority (SQLSTATE 42504).

LOAD
Revokes the authority to LOAD in this database.

QUIESCE_CONNECT
Revokes the authority to access the database while it is quiesced.

SECADM
Revokes the authority to administer database security.

SQLADM
Revokes the authority to monitor and tune SQL statements.

WLMADM
Revokes the authority to manage workload manager objects.

FROM
Indicates from whom the authorities are revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name.

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

The list of authorization IDs cannot include the authorization ID of the user that is issuing the statement (SQLSTATE 42502).

PUBLIC
Revokes the authorities from PUBLIC.

BY ALL
Revokes each named privilege from all named users who were explicitly granted those privileges, regardless of who granted them. This behavior is the default.

Rules

Security administrator mandatory: The database must have at least one authorization ID of type USER with the SECADM authority. The SECADM authority cannot be revoked from every user authorization ID (SQLSTATE 42523).

- For each *authorization-name* specified, if USER, GROUP, or ROLE is not specified, then:
  - For all rows for the specified object in the SYSCAT.DBAUTH catalog view where the grantee is *authorization-name*:
    - USER is assumed if all rows have a GRANTEETYPE of 'U'.
    - GROUP is assumed if all rows have a GRANTEETYPE of 'G'.
    - ROLE is assumed if all rows have a GRANTEETYPE of 'R'.
    - An error is returned (SQLSTATE 56092) if all rows do not have the same value for GRANTEETYPE.

Notes

- Revoking a specific privilege does not necessarily revoke the ability to complete a task. A user can proceed with a task if other privileges are held by PUBLIC, a group, or a role, or if the user holds a higher-level authority, such as DBADM.
- *Syntax alternatives*: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products.
  - CREATE_NOT_FENCED can be specified in place of CREATE_NOT_FENCED_ROUTINE.
  - SYSTEM can be specified in place of DATABASE.
  - NOT INCLUDING DEPENDENT PRIVILEGES can be specified as a syntax alternative.

Examples

- *Example 1:* Given that USER6 is only a user and not a group, revoke the privilege to create tables from the user USER6.

```
REVOKE CREATETAB ON DATABASE FROM USER6
```

- *Example 2:* Revoke BINDADD authority on the database from a group named D024. Two rows exist in the SYSCAT.DBAUTH catalog view for this grantee; one with a GRANTEETYPE of U and one with a GRANTEETYPE of G.

```
REVOKE BINDADD ON DATABASE FROM GROUP D024
```

In this case, the GROUP keyword must be specified; otherwise, an error occurs (SQLSTATE 56092).

- *Example 3:* Revoke security administrator authority from user Walid.

```
REVOKE SECADM ON DATABASE FROM USER Walid
```

- *Example 4:* A user with SECADM authority revokes the CREATE_SECURE_OBJECT authority from user Haytham.

```
REVOKE CREATE_SECURE_OBJECT ON DATABASE FROM USER HAYTHAM
```

REVOKE (exemption)

This form of the REVOKE statement revokes an exemption to a label-based access control (LBAC) access rule.

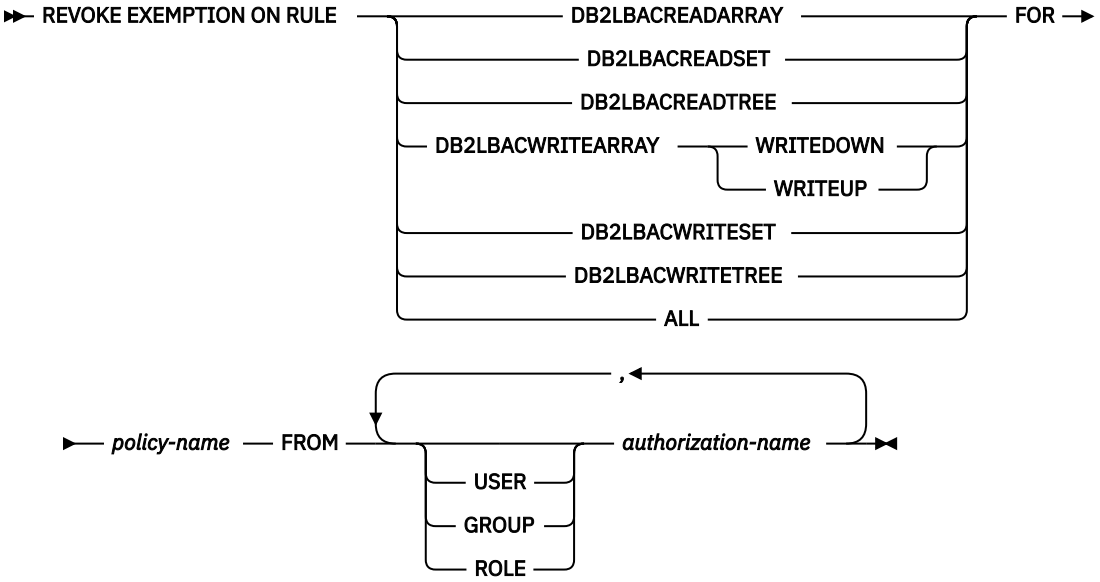
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include SECADM authority.

Syntax



Description

EXEMPTION ON RULE
Revokes the exemption on an access rule.

DB2LBACREADARRAY
Revokes an exemption on the predefined DB2LBACREADARRAY rule.

DB2LBACREADSET
Revokes an exemption on the predefined DB2LBACREADSET rule.

DB2LBACREADTREE
Revokes an exemption on the predefined DB2LBACREADTREE rule.

DB2LBACWRITEARRAY
Revokes an exemption on the predefined DB2LBACWRITEARRAY rule.

WRITEDOWN
Specifies that the exemption only applies to write down.

WRITEUP
Specifies that the exemption only applies to write up.

DB2LBACWRITESET
Revokes an exemption on the predefined DB2LBACWRITESET rule.

DB2LBACWRITETREE
Revokes an exemption on the predefined DB2LBACWRITETREE rule.

ALL
Revokes the exemptions on all of the predefined rules.

FOR *policy-name*
Specifies the name of the security policy on which exemptions are to be revoked.

FROM
Specifies from whom the exemption is revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name.

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

Rules

- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:
  - For all rows for the specified object in the SYSCAT.SECURITYPOLICYEXEMPTIONS catalog view where the grantee is *authorization-name*:
    - If all rows have a GRANTEETYPE of 'U', USER is assumed.
    - If all rows have a GRANTEETYPE of 'G', GROUP is assumed.
    - If all rows have a GRANTEETYPE of 'R', ROLE is assumed.
    - If all rows do not have the same value for GRANTEETYPE, an error is returned (SQLSTATE 56092).

Examples

- *Example 1:* Revoke the exemption on access rule DB2LBACREADSET for security policy DATA_ACCESS from user WALID.

```
REVOKE EXEMPTION ON RULE DB2LBACREADSET FOR DATA_ACCESS
  FROM USER WALID
```

- *Example 2:* Revoke an exemption on access rule DB2LBACWRITEARRAY with the WRITEDOWN option for security policy DATA_ACCESS from user BOBBY.

```
REVOKE EXEMPTION ON RULE DB2LBACWRITEARRAY WRITEDOWN
  FOR DATA_ACCESS FROM USER BOBBY
```

- *Example 3:* Revoke an exemption on access rule DB2LBACWRITEARRAY with the WRITEUP option for security policy DATA_ACCESS from user BOBBY.

```
REVOKE EXEMPTION ON RULE DB2LBACWRITEARRAY WRITEUP
  FOR DATA_ACCESS FROM USER BOBBY
```

REVOKE (global variable privileges)

This form of the REVOKE statement revokes one or more privileges on a created global variable.

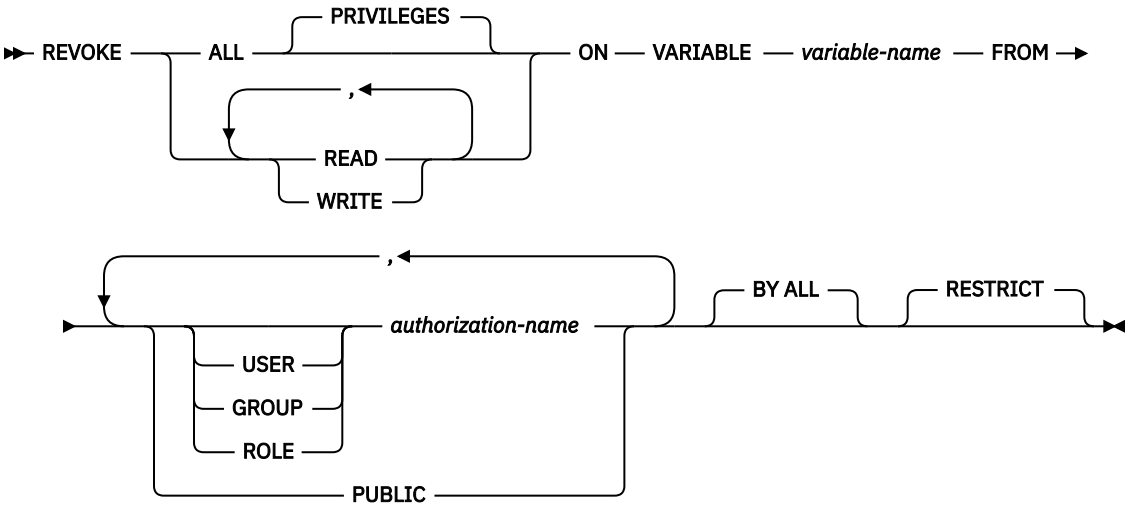
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include database ACCESSCTRL or SECADM authority or ACCESSCTRL authority on the schema containing the global variable.

Syntax



Description

ALL PRIVILEGES
Revokes all privileges held by an *authorization-name* for the specified global variable. If ALL is not specified, READ or WRITE must be specified. READ or WRITE must not be specified more than once.

READ
Revokes the privilege to read the value of the specified global variable.

WRITE
Revokes the privilege to assign a value to the specified global variable.

ON VARIABLE *variable-name*
Identifies the global variable on which one or more privileges are to be revoked. The *variable-name* must identify a global variable that exists at the current server and is not a module variable (SQLSTATE 42704).

FROM
Specifies from whom the privileges are revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group.

ROLE
Specifies that the *authorization-name* identifies an existing role at the current server (SQLSTATE 42704).

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles. The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Revokes the specified privileges from PUBLIC.

BY ALL
Revokes each specified privilege from all named users who were explicitly granted those privileges, regardless of who granted them. This is the default behavior.

RESTRICT
Specifies that the statement is to fail if any objects depend on the privileges being revoked. This is the default behavior.

Rules

- For each *authorization-name* specified, if none of the keywords USER, GROUP, or ROLE is specified, then for all rows for the specified object in the SYSCAT.VARIABLEAUTH catalog view where the grantee is *authorization-name*:
  - If GRANTEETYPE is 'U', USER is assumed.
  - If GRANTEETYPE is 'G', GROUP is assumed.
  - If GRANTEETYPE is 'R', ROLE is assumed.
  - If GRANTEETYPE does not have the same value, an error is returned (SQLSTATE 56092.
- If any SQL function, SQL method, procedure, view, trigger, or another global variable contains a global variable and depends on the privilege being revoked, the revoke operation will fail (SQLSTATE 42893).

Notes

- If the READ privilege on a global variable is revoked, packages with a dependency to write the value of the global variable (for example, by the SET statement) are not affected, because writing to a global variable is controlled by the WRITE privilege on that global variable.
- If the WRITE privilege on a global variable is revoked, packages with a dependency to read the value of the global variable are not affected, because reading from a global variable is controlled by the READ privilege on that global variable.
- Revoking a privilege does not necessarily impair the ability to perform the action. A user might be able to proceed if the required privilege is held through membership in a different group or role, or by PUBLIC.

Example

Revoke the WRITE privilege on global variable MYSCHEMA.MYJOB_PRINTER from user ZUBIRI.

```
REVOKE WRITE ON VARIABLE MYSCHEMA.MYJOB_PRINTER FROM ZUBIRI
```

REVOKE (index privileges)

This form of the REVOKE statement revokes the CONTROL privilege on an index.

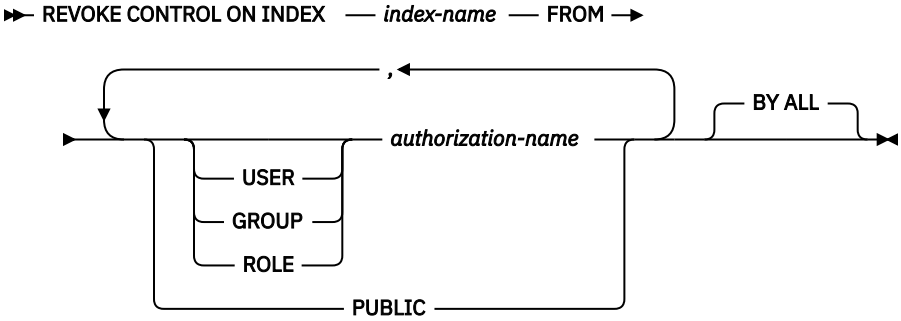
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include ACCESSCTRL or SECADM authority or ACCESSCTRL authority on the schema containing the index.

Syntax



Description

CONTROL
Revokes the privilege to drop the index. This is the CONTROL privilege for indexes, which is automatically granted to creators of indexes.

ON INDEX *index-name*
Specifies the name of the index on which the CONTROL privilege is to be revoked.

FROM
Indicates from whom the privileges are revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name.

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Revokes the privileges from PUBLIC.

BY ALL
Revokes the privilege from all named users who were explicitly granted that privilege, regardless of who granted it. This is the default behavior.

Rules

- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:
  - For all rows for the specified object in the SYSCAT.INDEXAUTH catalog view where the grantee is *authorization-name*:
    - If all rows have a GRANTEETYPE of 'U', USER is assumed.
    - If all rows have a GRANTEETYPE of 'G', GROUP is assumed.
    - If all rows have a GRANTEETYPE of 'R', ROLE is assumed.
    - If all rows do not have the same value for GRANTEETYPE, an error is returned (SQLSTATE 56092).

Notes

- Revoking a specific privilege does not necessarily revoke the ability to perform the action. A user can proceed with a task if other privileges are held by PUBLIC, a group, or a role, or if the user holds authorities such as ALTERIN on the schema of an index.

Examples

- *Example 1:* Given that USER4 is only a user and not a group, revoke the privilege to drop an index DEPTIDX from the user USER4.

```
REVOKE CONTROL ON INDEX DEPTIDX FROM KIESLER
```

- *Example 2:* Revoke the privilege to drop an index LUNCHITEMS from the user CHEF and the group WAITERS.

```
REVOKE CONTROL ON INDEX LUNCHITEMS
  FROM USER CHEF, GROUP WAITERS
```

REVOKE (module privileges)

This form of the REVOKE statement revokes the privilege on a module.

Invocation

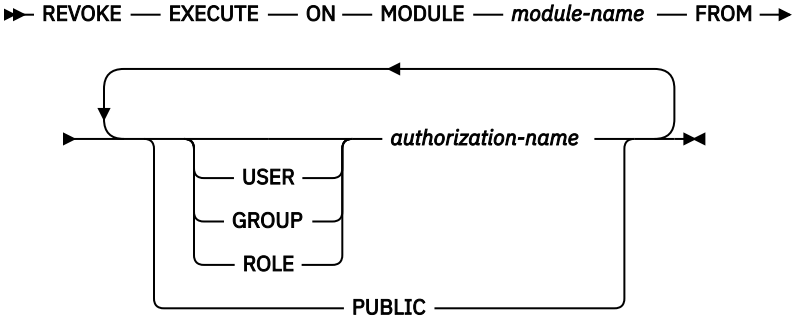
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include ACCESSCTRL or SECADM authority or ACCESSCTRL authority on the schema containing the module.

Note: In Db2 11.5.7 and later, the needed authorities are different if the module is SYSIBMADM.UTL_DIR. In this case, the authorities that are held by the authorization ID of the statement must include at least one of ACCESSCTRL, SECADM, or SYSADM.

Syntax



Description

EXECUTE

Revokes the privilege to reference published module objects. This includes revoking the privilege to:

- Execute any published routine defined in the module.
- Read from and write to any published global variables defined in the module.
- Reference any published user-defined types defined in the module.

- Reference any published conditions defined in the module.

ON MODULE ***module-name***
Identifies the module on which the privilege is revoked. The *module-name* must identify a module that exists at the current server (SQLSTATE 42704).

FROM
Indicates from whom the privilege is revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name. The role name must exist at the current server (SQLSTATE 42704).

authorization-name
Lists one or more authorization IDs. The same *authorization-name* must not be specified more than once

PUBLIC
Grants the privilege to a set of users (authorization IDs). For more information, see "Authorization, privileges and object ownership".

Example

The following example demonstrate how to revoke the EXECUTE privilege from a module named *myModa* from user *jones*

```
REVOKE EXECUTE ON MODULE MYMODA FROM JONES
```

REVOKE (package privileges)

This form of the REVOKE statement revokes CONTROL, BIND, and EXECUTE privileges against a package.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

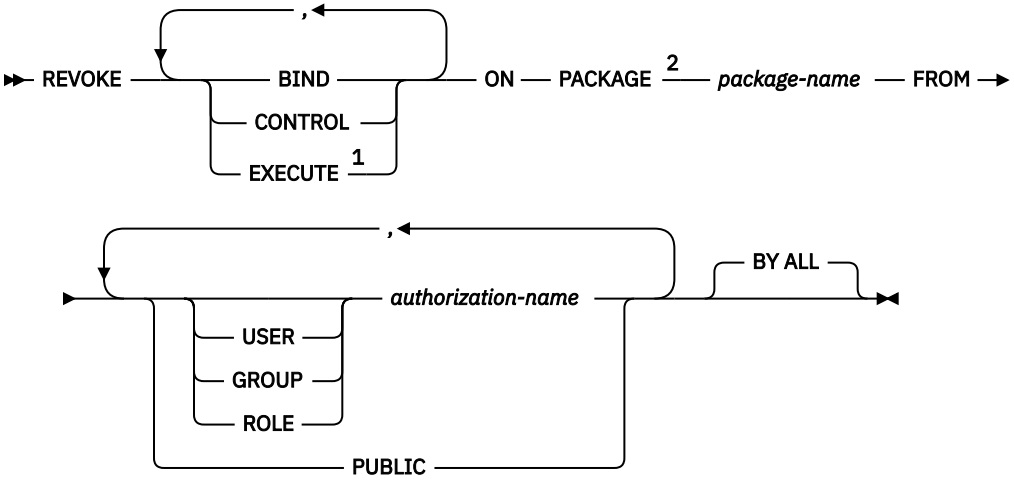
Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- CONTROL privilege on the referenced package
- ACCESSCTRL on the schema containing the package
- ACCESSCTRL or SECADM authority

ACCESSCTRL or SECADM authority or ACCESSCTRL authority on the schema containing the package is required to revoke the CONTROL privilege.

Syntax



Notes:

[1] RUN can be used as a synonym for EXECUTE.
[2] PROGRAM can be used as a synonym for PACKAGE.

Description

BIND
Revokes the privilege to execute BIND or REBIND on-or to add a new version of- the referenced package.

The BIND privilege cannot be revoked from an *authorization-name* that holds CONTROL privilege on the package, without also revoking the CONTROL privilege.

CONTROL
Revokes the privilege to drop the package and to extend package privileges to other users.

Revoking CONTROL does not revoke the other package privileges.

EXECUTE
Revokes the privilege to execute the package.

The EXECUTE privilege cannot be revoked from an *authorization-name* that holds CONTROL privilege on the package without also revoking the CONTROL privilege.

ON PACKAGE *package-name*
Specifies the name of the package on which privileges are to be revoked. The revoking of a package privilege applies to all versions of the package.

FROM
Indicates from whom the privileges are revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name.

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Revokes the privileges from PUBLIC.

BY ALL
Revokes each named privilege from all named users who were explicitly granted those privileges, regardless of who granted them. This is the default behavior.

Rules

- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:
  - For all rows for the specified object in the SYSCAT.PACKAGEAUTH catalog view where the grantee is *authorization-name*:
    - If all rows have a GRANTEETYPE of 'U', USER is assumed.
    - If all rows have a GRANTEETYPE of 'G', GROUP is assumed.
    - If all rows have a GRANTEETYPE of 'R', ROLE is assumed.
    - If all rows do not have the same value for GRANTEETYPE, an error is returned (SQLSTATE 56092).

Notes

- Revoking a specific privilege does not necessarily revoke the ability to perform the action. A user can proceed with a task if other privileges are held by PUBLIC, a group, or a role, or if the user holds privileges such as ALTERIN or SCHEMAADM on the schema of a package.

Examples

- *Example 1:* Revoke the EXECUTE privilege on package CORPDATA.PKGA from PUBLIC.

```
REVOKE EXECUTE
  ON PACKAGE CORPDATA.PKGA
  FROM PUBLIC
```

- *Example 2:* Revoke CONTROL authority on the RRSP_PKG package for the user FRANK and for PUBLIC.

```
REVOKE CONTROL
  ON PACKAGE RRSP_PKG
  FROM USER FRANK, PUBLIC
```

REVOKE (role)

This form of the REVOKE statement revokes roles from users, groups, or other roles.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

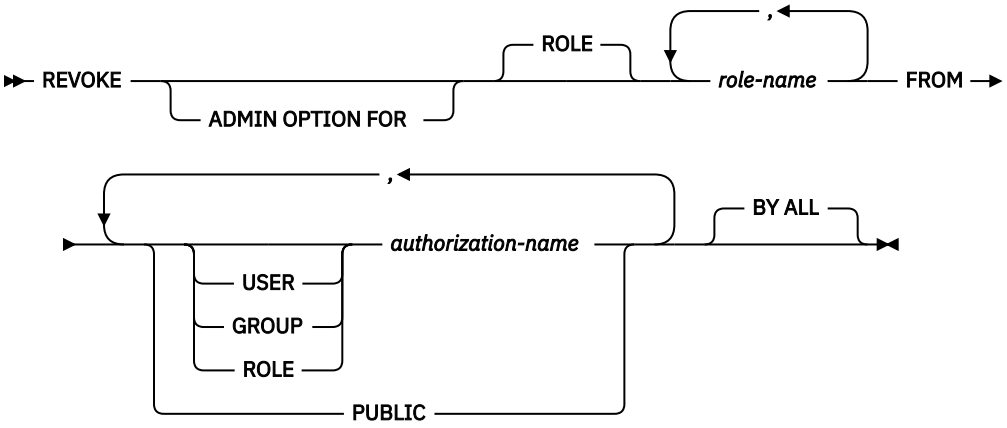
Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- The WITH ADMIN OPTION on the role
- SECADM authority

SECADM authority is required to revoke the ADMIN OPTION FOR *role-name* from an *authorization-name* or to revoke a *role-name* from an *authorization-name* that has the WITH ADMIN OPTION on that role.

Syntax



Description

ADMIN OPTION FOR
Revokes the WITH ADMIN OPTION on *role-name*. The WITH ADMIN OPTION on *role-name* must be held by *authorization-name* or by PUBLIC, if PUBLIC is specified (SQLSTATE 42504). If the ADMIN OPTION FOR clause is specified, only the WITH ADMIN OPTION on ROLE *role-name* is revoked, not the role itself.

ROLE *role-name*
Specifies the role that is to be revoked. The *role-name* must identify an existing role at the current server (SQLSTATE 42704) that has been granted to *authorization-name* or to PUBLIC, if PUBLIC is specified (SQLSTATE 42504).

FROM
Specifies from whom the role is revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group.

ROLE
Specifies that the *authorization-name* identifies an existing role at the current server (SQLSTATE 42704).

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles. The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Revokes the specified roles from PUBLIC.

BY ALL
Revokes the *role-name* from each specified *authorization-name* that was explicitly granted that role, regardless of who granted it. This is the default behavior.

Rules

- For each *authorization-name* specified, if none of the keywords USER, GROUP, or ROLE is specified, then for all rows for the specified object in the SYSCAT.ROLEAUTH catalog view where the grantee is *authorization-name*:
  - If GRANTEETYPE is 'U', USER is assumed.
  - If GRANTEETYPE is 'G', GROUP is assumed.
  - If GRANTEETYPE is 'R', ROLE is assumed.

  - If GRANTEETYPE does not have the same value, an error is returned (SQLSTATE 56092.
- The *role-name* must not identify a role, or a role that contains *role-name*, if the role has either EXECUTE privilege on a routine or USAGE privilege on a sequence, and an SQL object other than a package is dependent on the routine or sequence (SQLSTATE 42893). The owner of the SQL object is either *authorization-name* or any user that is a member of *authorization-name*, where *authorization-name* is a role.

Notes

- If a role is revoked from an *authorization-name* or from PUBLIC, all privileges that the role held are no longer available to the *authorization-name* or to PUBLIC through that role.
- Revoking a role does not necessarily revoke the ability to perform a particular action by way of a privilege that was granted to that role. A user might still be able to proceed if other privileges are held by PUBLIC, by a group to which the user belongs, by another role granted to the user, or if the user has a higher level authority, such as DBADM.

Examples

- *Example 1:* Revoke the role INTERN from the role DOCTOR and the role DOCTOR from the role SPECIALIST.

```
REVOKE ROLE INTERN FROM ROLE DOCTOR
REVOKE ROLE DOCTOR FROM ROLE SPECIALIST
```

- *Example 2:* Revoke the role INTERN from PUBLIC.

```
REVOKE ROLE INTERN FROM PUBLIC
```

- *Example 3:* Revoke the role SPECIALIST from user BOB and group TORONTO.

```
REVOKE ROLE SPECIALIST FROM USER BOB, GROUP TORONTO BY ALL
```

REVOKE (routine privileges)

This form of the REVOKE statement revokes privileges on a routine (function, method, or procedure) that is not defined in a module.

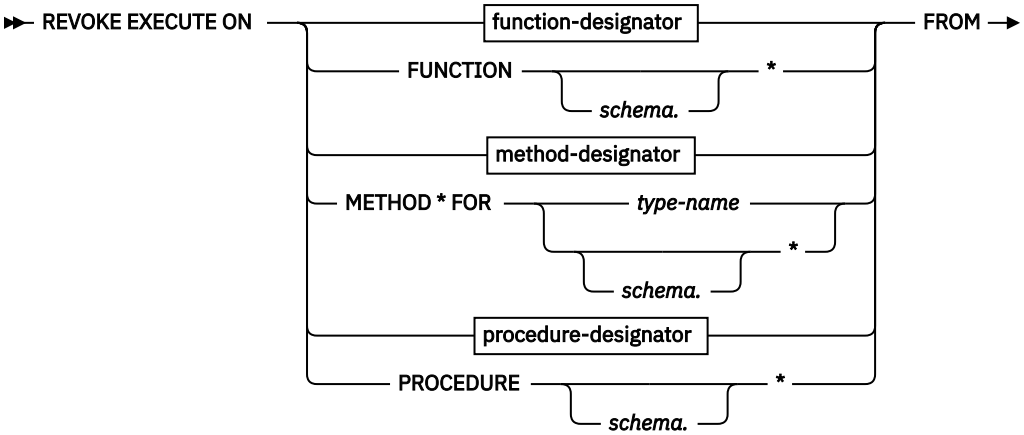
Invocation

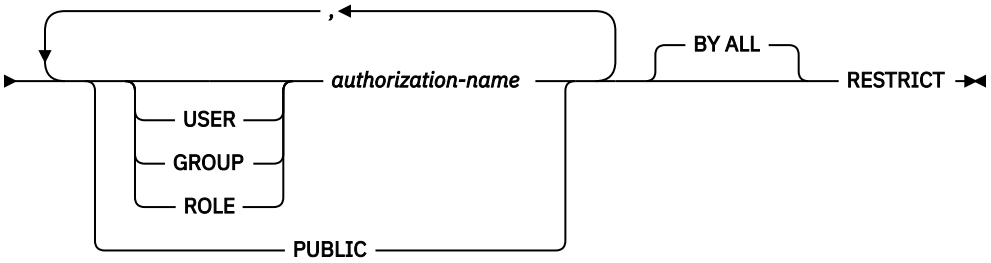
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

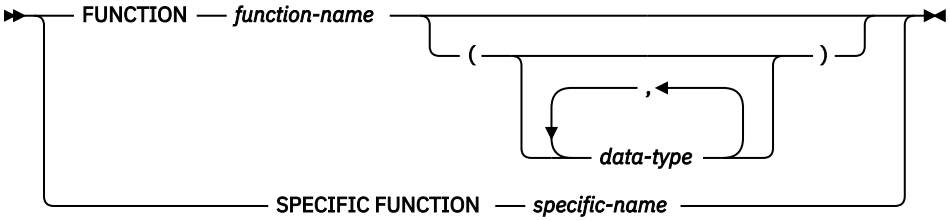
The privileges held by the authorization ID of the statement must include ACCESSCTRL or SECADM authority or ACCESSCTRL authority on the schema containing the routine.

Syntax

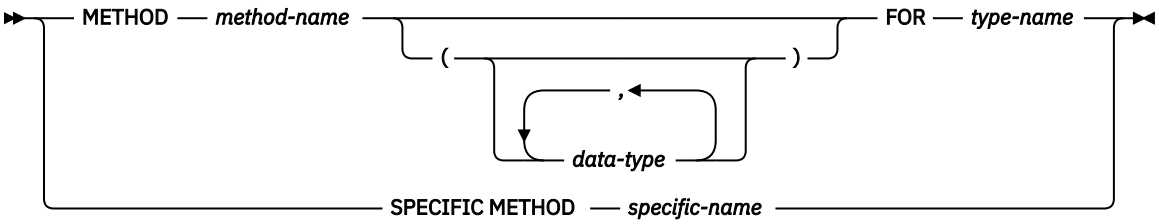




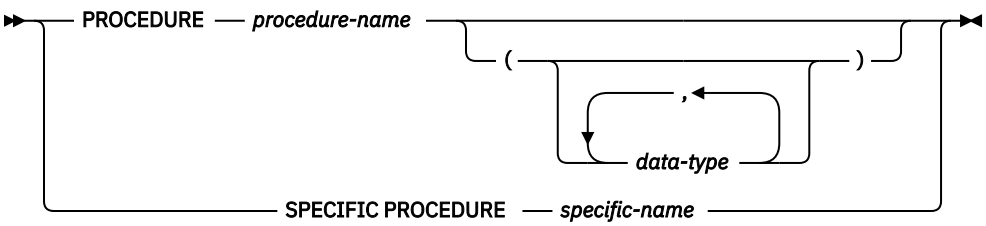
function-designator



method-designator



procedure-designator



Description

EXECUTE

Revokes the privilege to run the identified user-defined function, method, or procedure.

function-designator
Uniquely identifies the function from which the privilege is revoked. For more information, see "Function, method, and procedure designators" on page 745.

FUNCTION *schema.**
Identifies the explicit grant for all the existing and future functions in the schema. Revoking the *schema*.* privilege does not revoke any privileges that were granted on a specific function. In dynamic SQL statements, if a schema is not specified, the schema in the CURRENT SCHEMA special register will be used. In static SQL statements, if a schema is not specified, the schema in the QUALIFIER precompile/bind option will be used.

method-designator
Uniquely identifies the method from which the privilege is revoked. For more information, see "Function, method, and procedure designators" on page 745.

METHOD *
Identifies the explicit grant for all the existing and future methods for the type *type-name*. Revoking the * privilege does not revoke any privileges that were granted on a specific method.

FOR *type-name*
Names the type in which the specified method is found. The name must identify a type already described in the catalog (SQLSTATE 42704). In dynamic SQL statements, the value of the CURRENT SCHEMA special register is used as a qualifier for an unqualified type name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified type names. An asterisk (*) can be used in place of *type-name* to identify the explicit grant on all existing and future methods for all existing and future types in the schema. Revoking the privilege using an asterisk for method and *type-name* does not revoke any privileges that were granted on a specific method or on all methods for a specific type.

procedure-designator
Uniquely identifies the procedure from which the privilege is revoked. For more information, see "Function, method, and procedure designators" on page 745.

PROCEDURE *schema.**
Identifies the explicit grant for all the existing and future procedures in the schema. Revoking the *schema*.* privilege does not revoke any privileges that were granted on a specific procedure. In dynamic SQL statements, if a schema is not specified, the schema in the CURRENT SCHEMA special register will be used. In static SQL statements, if a schema is not specified, the schema in the QUALIFIER precompile/bind option will be used.

FROM
Specifies from whom the EXECUTE privilege is revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name.

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Revokes the EXECUTE privilege from PUBLIC.

BY ALL
Revokes the EXECUTE privilege from all named users who were explicitly granted the privilege, regardless of who granted it. This is the default behavior.

RESTRICT

Specifies that the EXECUTE privilege cannot be revoked if both of the following conditions are true (SQLSTATE 42893):

- The specified routine is used in a view, trigger, constraint, index extension, SQL function, SQL method, aggregate interface function, transform group, or is referenced as the SOURCE of a sourced function.
- The loss of the EXECUTE privilege would cause the owner of the view, trigger, constraint, index extension, SQL function, SQL method, aggregate interface function, transform group, or sourced function to no longer be able to execute the specified routine.

Rules

- It is not possible to revoke the EXECUTE privilege on a function or method defined with schema 'SYSIBM' or 'SYSFUN' (SQLSTATE 42832).
- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:
  - For all rows for the specified object in the SYSCAT.ROUTINEAUTH catalog view where the grantee is *authorization-name*:
    - If all rows have a GRANTEETYPE of 'U', USER is assumed.
    - If all rows have a GRANTEETYPE of 'G', GROUP is assumed.
    - If all rows have a GRANTEETYPE of 'R', ROLE is assumed.
    - If all rows do not have the same value for GRANTEETYPE, an error is returned (SQLSTATE 56092).

Notes

- If a package depends on a routine (function, method, or procedure), and the EXECUTE privilege on that routine is revoked from PUBLIC, a user, or a role, the package becomes inoperative if the routine is a function or a method, and the package becomes invalid if the routine is a procedure, unless the package owner still holds the EXECUTE privilege on the routine. The package owner can still hold the EXECUTE privilege if:
  - The package owner was explicitly granted the EXECUTE privilege
  - The package owner is a member of a role that holds the EXECUTE privilege
  - The EXECUTE privilege was granted to PUBLIC
  - The EXECUTEIN privilege was granted to PUBLIC

  Because group privileges are not considered for static packages, the package becomes inoperative (in the case of a function or a method) or invalid (in the case of a procedure) even if a group to which the package owner belongs holds the EXECUTE privilege.

Examples

- *Example 1:* Revoke the EXECUTE privilege on function CALC_SALARY from user JONES. Assume that there is only one function in the schema with function name CALC_SALARY.

```
REVOKE EXECUTE ON FUNCTION CALC_SALARY FROM JONES RESTRICT
```

- *Example 2:* Revoke the EXECUTE privilege on procedure VACATION_ACCR from all users at the current server.

```
REVOKE EXECUTE ON PROCEDURE VACATION_ACCR FROM PUBLIC RESTRICT
```

- *Example 3:* Revoke the EXECUTE privilege on function NEW_DEPT_HIRES from HR (Human Resources). The function has two input parameters of type INTEGER and CHAR(10), respectively. Assume that the schema has more than one function named NEW_DEPT_HIRES.

```
REVOKE EXECUTE ON FUNCTION NEW_DEPT_HIRES (INTEGER, CHAR(10))
  FROM HR RESTRICT
```

- *Example 4:* Revoke the EXECUTE privilege on method SET_SALARY for type EMPLOYEE from user Jones.

```
REVOKE EXECUTE ON METHOD SET_SALARY FOR EMPLOYEE FROM JONES RESTRICT
```

REVOKE (schema privileges and authorities)

This form of the REVOKE statement revokes the privileges and authorities on a schema.

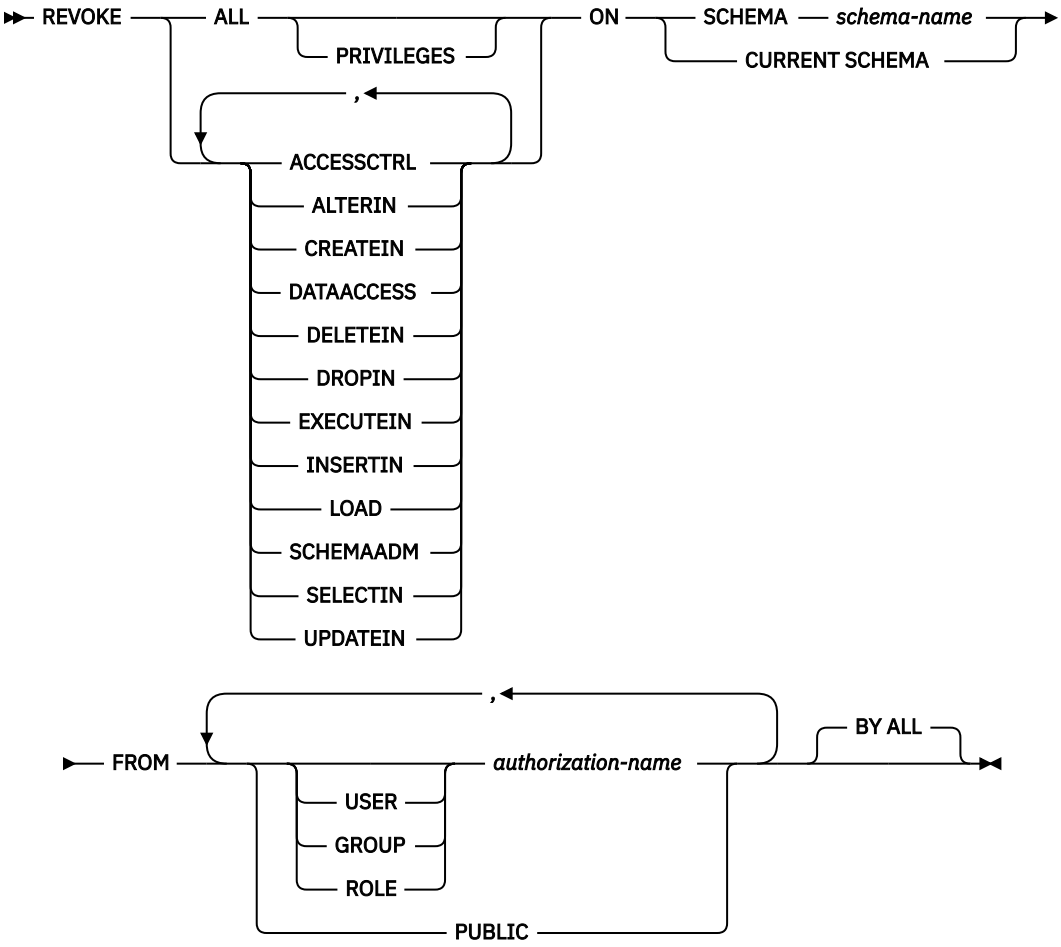
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

- The privileges held by the authorization ID of the statement must include ACCESSCTRL or SECADM authority or ACCESSCTRL authority on the schema-name.
- To revoke schema ACCESSCTRL, the authorization must have SECADM or database ACCESSCTRL authority.

Syntax



Description

ALL or ALL PRIVILEGES
Revoke all of the following schema privileges on the schema that is named in the ON clause:

- ALTERIN
- CREATEIN
- DELETEIN
- DROPIN
- EXECUTEIN
- INSERTIN
- SELECTIN
- UPDATEIN

If ALL is not specified, one or more of the keywords in the list of privileges must be specified.

ACCESSCTRL
Revokes the authority to grant and revoke schema-level privileges. For more information, see Schema access control authority (ACCESSCTRL).

ALTERIN
Revokes the privilege to alter or comment on objects in the schema.

CREATEIN
Revokes the privilege to create objects in the schema.

DATAACCESS
Revokes the authority to access data in the schema. For more information, see Schema data access authority (DATAACCESS).

DELETEIN
Revokes the privilege to delete all objects in the schema.

DROPIN
Revokes the privilege to drop objects in the schema.

EXECUTEIN
Revokes the privilege to execute user-defined functions, methods, procedures, packages, and modules defined in the schema.

INSERTIN
Revokes the privilege to insert data in to all objects in the schema.

LOAD
Revokes the authority to load in this schema. For more information, see Schema load authority (LOAD).

SCHEMAADM
Revokes the schema administrator authority. For more information, see Schema administration authority (SCHEMAADM).

SELECTIN
Revokes the privilege to select from all objects in the schema.

UPDATEIN
Revokes the privilege to update all objects in the schema.

ON

SCHEMA *schema-name*
Specifies the name of the schema on which privileges are to be revoked.

CURRENT SCHEMA
Specifies that the privileges will be revoked from the schema described by the DB2® special register CURRENT SCHEMA.

FROM
Indicates from whom the privileges are revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name.

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Revokes the privileges from PUBLIC.

BY ALL
Revokes each named privilege from all named users who were explicitly granted those privileges, regardless of who granted them. This is the default behavior.

Rules

- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:

- If the security plug-in in effect for the instance cannot determine the status of the authorization-name, an error is returned (SQLSTATE 56092).
- If the *authorization-name* is defined as ROLE in the database, and as either GROUP or USER according to the security plug-in in effect, an error is returned (SQLSTATE 56092).
- If the *authorization-name* is defined according to the security plug-in in effect as both USER and GROUP, an error is returned (SQLSTATE 56092).
- If the *authorization-name* is defined according to the security plug-in in effect as USER only, or if it is undefined, USER is assumed.
- If the *authorization-name* is defined according to the security plug-in in effect as GROUP only, GROUP is assumed.
- If the *authorization-name* is defined in the database as ROLE only, ROLE is assumed.

Notes

- Revoking a specific privilege does not necessarily revoke the ability to perform the action. A user can proceed with a task if other privileges are held by PUBLIC, a group, or a role, or if the user holds a higher level authority such as DBADM.

Examples

- *Example 1:* Given that USER4 is only a user and not a group, revoke the privilege to create objects in schema DEPTIDX from the user USER4.

```
REVOKE CREATEIN ON SCHEMA DEPTIDX FROM USER4
```

- *Example 2:* Revoke the privilege to drop objects in schema LUNCH from the user CHEF and the group WAITERS.

```
REVOKE DROPIN ON SCHEMA LUNCH
  FROM USER CHEF, GROUP WAITERS
```

REVOKE (security label)

This form of the REVOKE statement revokes a label-based access control (LBAC) security label.

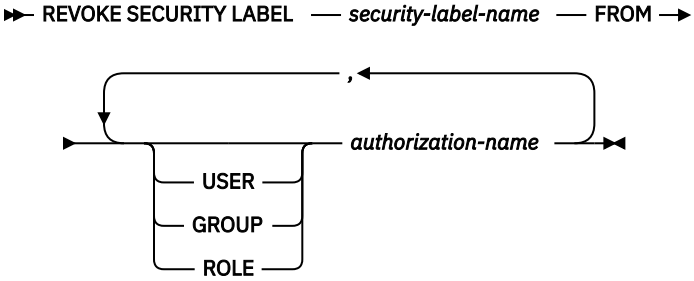
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include SECADM authority.

Syntax



Description

SECURITY LABEL *security-label-name*
Revokes the security label *security-label-name*. The name must be qualified with a security policy (SQLSTATE 42704) and must identify a security label that exists at the current server (SQLSTATE 42704), and that is held by *authorization-name* (SQLSTATE 42504).

FROM
Specifies from whom the specified security label is revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name. The role name must exist at the current server (SQLSTATE 42704).

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

Rules

- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:
  - For all rows for the specified object in the SYSCAT.SECURITYLABELACCESS catalog view where the grantee is *authorization-name*:
    - If all rows have a GRANTEETYPE of 'U', USER is assumed.
    - If all rows have a GRANTEETYPE of 'G', GROUP is assumed.
    - If all rows have a GRANTEETYPE of 'R', ROLE is assumed.
    - If all rows do not have the same value for GRANTEETYPE, an error is returned (SQLSTATE 56092).

Example

Revoke the security label EMPLOYEESECLABEL, which is part of the security policy DATA_ACCESS, from user WALID.

```
REVOKE SECURITY LABEL DATA_ACCESS.EMPLOYEESECLABEL
  FROM USER WALID
```

REVOKE (sequence privileges)

This form of the REVOKE statement revokes privileges on a sequence.

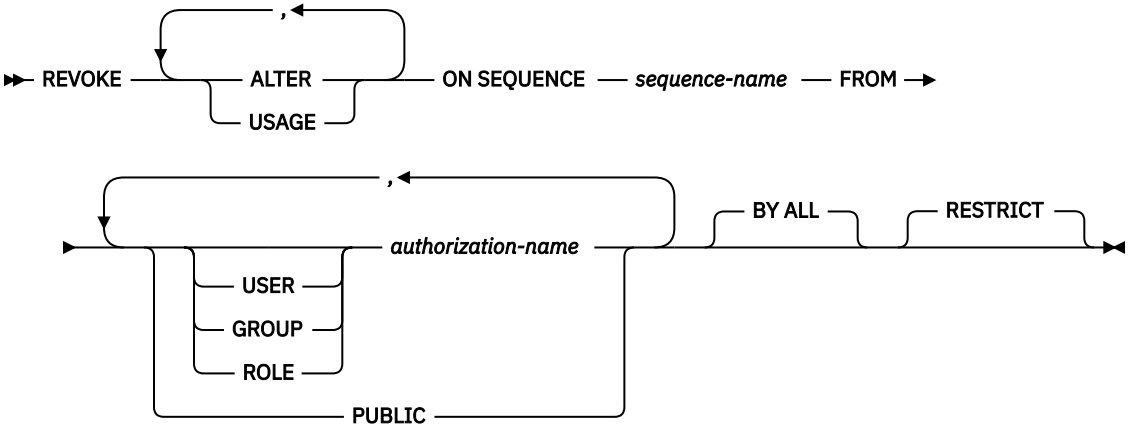
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include ACCESSCTRL or SECADM authority or ACCESSCTRL authority on the schema containing the sequence-name.

Syntax



Description

ALTER
Revokes the privilege to change the properties of a sequence or to restart sequence number generation using the ALTER SEQUENCE statement.

USAGE
Revokes the privilege to reference a sequence using *nextval-expression* or *prevval-expression*.

ON SEQUENCE *sequence-name*
Identifies the sequence on which the specified privileges are to be revoked. The sequence name, including an implicit or explicit schema qualifier, must uniquely identify an existing sequence at the current server. If no sequence by this name exists, an error is returned (SQLSTATE 42704).

FROM
Specifies from whom the privileges are revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name.

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Revokes the specified privileges from PUBLIC.

BY ALL
Revokes each specified privilege from all named users who were explicitly granted those privileges, regardless of who granted them. This is the default behavior.

RESTRICT
This optional keyword indicates that the statement will fail if any objects depend on the privilege being revoked.

Rules

- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:

– For all rows for the specified object in the SYSCAT.SEQUENCEAUTH catalog view where the grantee is *authorization-name*:
  - If all rows have a GRANTEETYPE of 'U', USER is assumed.
  - If all rows have a GRANTEETYPE of 'G', GROUP is assumed.
  - If all rows have a GRANTEETYPE of 'R', ROLE is assumed.
  - If all rows do not have the same value for GRANTEETYPE, an error is returned (SQLSTATE 56092).

Notes

- Revoking a privilege on a sequence from the authorization ID under which a package was bound will cause the package to become invalid if the authorization ID does not continue to hold the privilege on the sequence through different means; for example, through membership in a role that holds the privilege.
- Revoking a specific privilege does not necessarily remove the ability to perform an action. A user can proceed if other privileges are held by PUBLIC or by a group to which the user belongs, or if the user has a higher level of authority, such as DBADM.

Examples

- *Example 1:* Revoke the USAGE privilege on a sequence called GENERATE_ID from user ENGLES. There is one row in the SYSCAT.SEQUENCEAUTH catalog view for this sequence and grantee, and the GRANTEETYPE value is U.

```
REVOKE USAGE ON SEQUENCE GENERATE_ID FROM ENGLES
```

- *Example 2:* Revoke alter privileges on sequence GENERATE_ID that were previously granted to all local users. (Grants to specific users are not affected.)

```
REVOKE ALTER ON SEQUENCE GENERATE_ID FROM PUBLIC
```

- *Example 3:* Revoke all privileges on sequence GENERATE_ID from users PELLOW and MLI, and from group PLANNERS.

```
REVOKE ALTER, USAGE ON SEQUENCE GENERATE_ID
  FROM USER PELLOW, USER MLI, GROUP PLANNERS
```

REVOKE (server privileges)

This form of the REVOKE statement revokes the privilege to access and use a specified data source in pass-through mode.

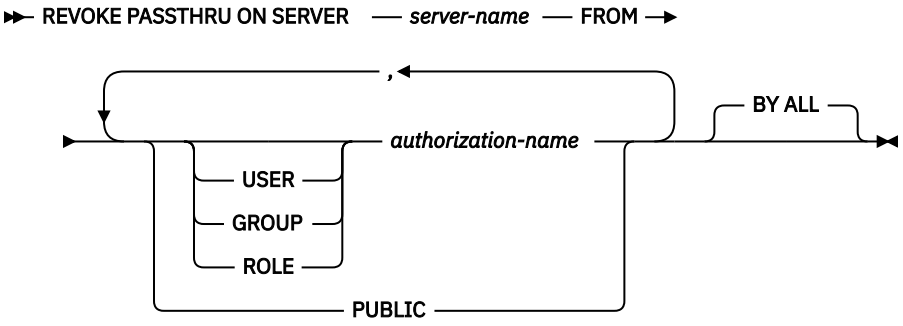
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include ACCESSCTRL or SECADM authority.

Syntax



Description

SERVER *server-name*
Names the data source for which the privilege to use in pass-through mode is being revoked. *server-name* must identify a data source that is described in the catalog.

FROM
Specifies from whom the privilege is revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name.

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Revokes from PUBLIC the privilege to pass through to *server-name*.

BY ALL
Revokes the privilege from all named users who were explicitly granted that privilege, regardless of who granted it. This is the default behavior.

Rules

- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:
  - For all rows for the specified object in the SYSCAT.PASSTHRUAUTH catalog view where the grantee is *authorization-name*:
    - If all rows have a GRANTEETYPE of 'U', USER is assumed.
    - If all rows have a GRANTEETYPE of 'G', GROUP is assumed.
    - If all rows have a GRANTEETYPE of 'R', ROLE is assumed.
    - If all rows do not have the same value for GRANTEETYPE, an error is returned (SQLSTATE 56092).

Examples

- *Example 1:* Revoke USER6's privilege to pass through to data source MOUNTAIN.

```
REVOKE PASSTHRU ON SERVER MOUNTAIN FROM USER USER6
```

- *Example 2:* Revoke group D024's privilege to pass through to data source EASTWING.

```
REVOKE PASSTHRU ON SERVER EASTWING FROM GROUP D024
```

The members of group D024 will no longer be able to use their group ID to pass through to EASTWING. But if any members have the privilege to pass through to EASTWING under their own user IDs, they will retain this privilege.

REVOKE (SETSESSIONUSER privilege)

This form of the REVOKE statement revokes one or more SETSESSIONUSER privileges from one or more authorization IDs.

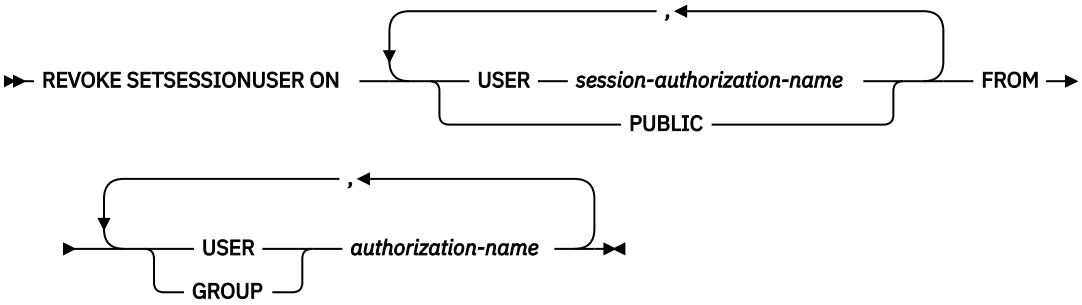
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include SECADM authority.

Syntax



Description

SETSESSIONUSER ON
Revokes the privilege to assume the identity of a new authorization ID.

USER *session-authorization-name*
Specifies the authorization ID that the *authorization-name* is able to assume, using the SET SESSION AUTHORIZATION statement. The *session-authorization-name* must identify a user that the *authorization-name* can assume, not a group (SQLSTATE 42504).

PUBLIC
Specifies that all privileges to set the session authorization will be revoked.

FROM
Specifies from whom the privilege is revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

authorization-name,...
Lists the authorization IDs of one or more users or groups.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

Examples

- *Example 1:* User PAUL holds the privilege to set the session authorization to WALID and therefore to execute SQL statements as user WALID. The following statement revokes that privilege.

```
REVOKE SETSESSIONUSER ON USER WALID
  FROM USER PAUL
```

- *Example 2:* User GUYLAINE holds the privilege to set the session authorization to BOBBY, RICK, or KEVIN and therefore to execute SQL statements as BOBBY, RICK, or KEVIN. The following statement revokes the privilege to use two of those authorization IDs. After this statement executes, GUYLAINE will only be able to set the session authorization to KEVIN.

```
REVOKE SETSESSIONUSER ON USER BOBBY, USER RICK
  FROM USER GUYLAINE
```

- *Example 3:* The group ACCTG and user WALID can set session authorization to any authorization ID. The following statement revokes that privilege from both ACCTG and WALID.

```
REVOKE SETSESSIONUSER ON PUBLIC
  FROM USER WALID, GROUP ACCTG
```

REVOKE (table space privileges)

This form of the REVOKE statement revokes the USE privilege on a table space.

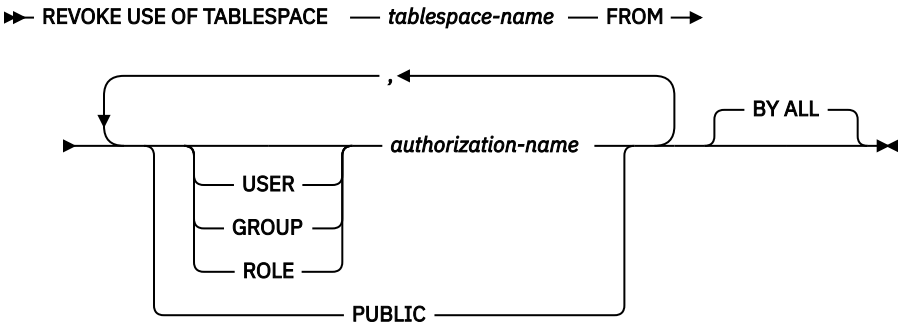
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include ACCESSCTRL, SECADM, SYSCTRL, or SYSADM authority.

Syntax



Description

USE
Revokes the privilege to specify or default to the table space when creating a table.

OF TABLESPACE *tablespace-name*
Specifies the table space on which the USE privilege is to be revoked. The table space cannot be SYSCATSPACE (SQLSTATE 42838) or a SYSTEM TEMPORARY table space (SQLSTATE 42809).

FROM
Indicates from whom the USE privilege is revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name.

authorization-name
Lists the authorization IDs of one or more users, groups, or roles.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Revokes the USE privilege from PUBLIC.

BY ALL
Revokes the privilege from all named users who were explicitly granted that privilege, regardless of who granted it. This is the default behavior.

Rules

- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:
  - For all rows for the specified object in the SYSCAT.TBSPACEAUTH catalog view where the grantee is *authorization-name*:
    - If all rows have a GRANTEETYPE of 'U', USER is assumed.
    - If all rows have a GRANTEETYPE of 'G', GROUP is assumed.
    - If all rows have a GRANTEETYPE of 'R', ROLE is assumed.
    - If all rows do not have the same value for GRANTEETYPE, an error is returned (SQLSTATE 56092).

Notes

- Revoking the USE privilege does not necessarily revoke the ability to create tables in that table space. A user may still be able to create tables in that table space if the USE privilege is held by PUBLIC or a group, or if the user has a higher level authority, such as DBADM.

Example

Revoke the privilege to create tables in table space PLANS from the user BOBBY.

```
REVOKE USE OF TABLESPACE PLANS FROM USER BOBBY
```

REVOKE (table, view, or nickname privileges)

This form of the REVOKE statement revokes privileges on a table, view, or nickname.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

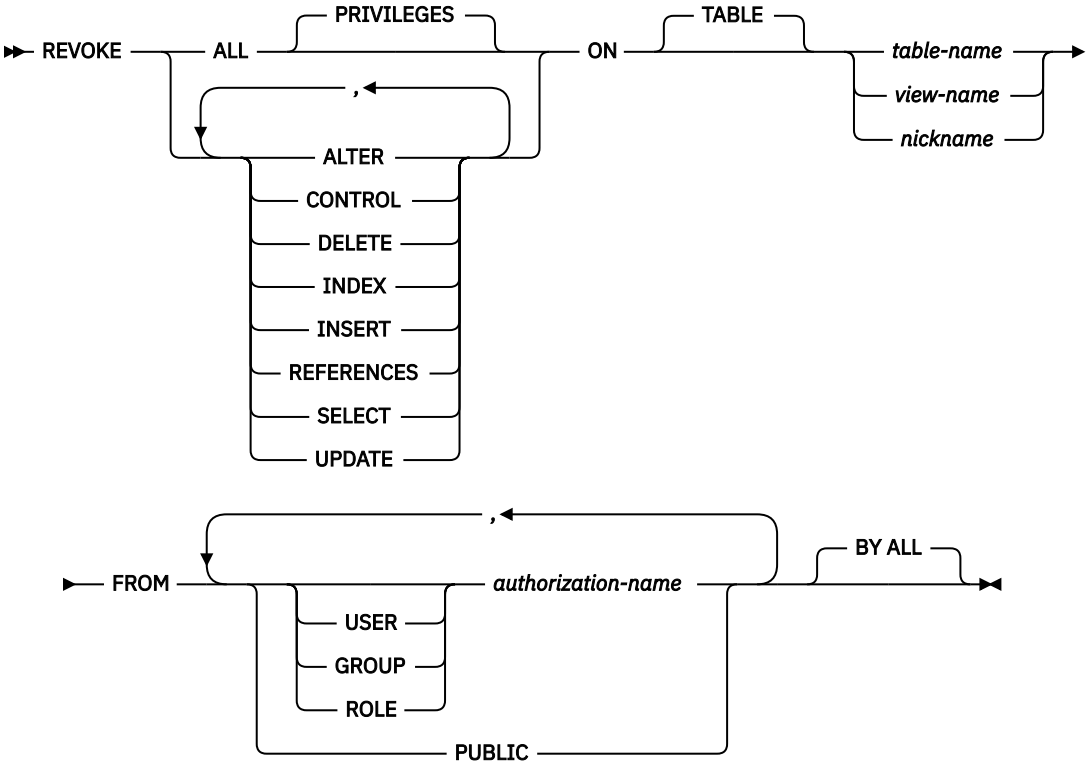
Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- CONTROL privilege on the referenced table, view, or nickname
- ACCESSCTRL authority on the schema containing the identified table, view, or nickname
- ACCESSCTRL or SECADM authority

Schema ACCESSCTRL, ACCESSCTRL or SECADM authority is required to revoke the CONTROL privilege. ACCESSCTRL or SECADM authority is required to revoke privileges on catalog tables and views.

Syntax



Description

ALL or ALL PRIVILEGES
Revokes all privileges (except CONTROL) held by an authorization-name for the specified tables, views, or nicknames.

If ALL is not used, one or more of the keywords listed in the option stack (ALTER through UPDATE) must be used. Each keyword revokes the privilege described, but only as it applies to the tables, views, or nicknames named in the ON clause. The same keyword must not be specified more than once.

ALTER
Revokes the privilege to add columns to the base table definition; create or drop a primary key or unique constraint on the table; create or drop a foreign key on the table; add/change a comment on the table, view, or nickname; create or drop a check constraint; create a trigger; add, reset, or drop a column option for a nickname; or, change nickname column names or data types.

CONTROL
Revokes the ability to drop the table, view, or nickname, and the ability to execute the RUNSTATS utility on the table and indexes.

Revoking CONTROL privilege from an *authorization-name* does not revoke other privileges granted to the user on that object.

DELETE
Revokes the privilege to delete rows from the table, updatable view, or nickname.

INDEX
Revokes the privilege to create an index on the table or an index specification on the nickname. The creator of an index or index specification automatically has the CONTROL privilege over the index or index specification (authorizing the creator to drop the index or index specification). In addition, the creator retains this privilege even if the INDEX privilege is revoked.

INSERT
Revokes the privileges to insert rows into the table, updatable view, or nickname, and to run the IMPORT utility.

REFERENCES
Revokes the privilege to create or drop a foreign key referencing the table as the parent. Any column level REFERENCES privileges are also revoked.

SELECT
Revokes the privilege to retrieve rows from the table or view, to create a view on a table, and to run the EXPORT utility against the table or view.

Revoking SELECT privilege may cause some views to be marked inoperative. (For information about inoperative views, see "CREATE VIEW".)

UPDATE
Revokes the privilege to update rows in the table, updatable view, or nickname. Any column level UPDATE privileges are also revoked.

ON TABLE *table-name* or *view-name* or *nickname*
Specifies the table, view, or nickname on which privileges are to be revoked. The *table-name* cannot be a declared temporary table (SQLSTATE 42995).

FROM
Indicates from whom the privileges are revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group name.

ROLE
Specifies that the *authorization-name* identifies a role name.

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles.

The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Revokes the privileges from PUBLIC.

BY ALL
Revokes each named privilege from all named users who were explicitly granted those privileges, regardless of who granted them. This is the default behavior.

Rules

- For each *authorization-name* specified, if neither USER, GROUP, nor ROLE is specified, then:
  - For all rows for the specified object in the SYSCAT.TABAUTH and SYSCAT.COLAUTH catalog views where the grantee is *authorization-name*:
    - If all rows have a GRANTEETYPE of 'U', USER is assumed.
    - If all rows have a GRANTEETYPE of 'G', GROUP is assumed.
    - If all rows have a GRANTEETYPE of 'R', ROLE is assumed.
    - If all rows do not have the same value for GRANTEETYPE, an error is returned (SQLSTATE 56092).

Notes

- If a privilege is revoked from the *authorization-name* that is the owner of the view (as recorded in the OWNER column in SYSCAT.VIEWS), that privilege is also revoked from any dependent views.
- If the owner of the view loses a SELECT privilege on some object on which the view definition depends (or an object upon which the view definition depends is dropped, or made inoperative in the case of another view), the view will be made inoperative.

  However, if a user who holds ACCESSCTRL or SECADM authority explicitly revokes all privileges on the view from the owner, then the record of the OWNER will not appear in SYSCAT.TABAUTH but nothing will happen to the view - it remains operative.
- Privileges on inoperative views cannot be revoked.
- A package might become invalid when the authorization ID under which the package was bound loses a privilege on an object on which the package depends. The privilege can be lost in one of the following ways:
  - The privilege is revoked from the authorization ID
  - The privilege is revoked from a role of which the authorization ID is a member
  - The privilege is revoked from PUBLIC

  A package remains invalid until a bind or rebind operation on the application is successfully executed, or the application is executed and the database manager successfully rebinds the application (using information stored in the catalogs). Packages marked invalid due to a revoke may be successfully rebound without any additional grants.

  For example, if a package owned by USER1 contains a SELECT from table T1, and the SELECT privilege on table T1 is revoked from USER1, the package will be marked invalid. If SELECT authority is granted again, or if the user holds DBADM authority, the package is successfully rebound when executed.

  Another example is a package owned by USER1, who is a member of role R1. The package contains a SELECT from table T1, and the SELECT privilege on table T1 is revoked from role R1. The package will be marked invalid, assuming USER1 does not hold the SELECT privilege on table T1 by other means.
- Packages, triggers or views that include the use of OUTER(*Z*) in the FROM clause, are dependent on having SELECT privilege on every subtable or subview of *Z*. Similarly, packages, triggers, or views that include the use of DEREF(*Y*) where *Y* is a reference type with a target table or view *Z*, are dependent on having SELECT privilege on every subtable or subview of *Z*. Such packages might become invalid, and such triggers or views made inoperative when the authorization ID under which the packages were bound, or the owner of the triggers or views loses the SELECT privilege. The SELECT privilege can be lost in one of the following ways:
  - SELECT privilege is revoked from the authorization ID
  - SELECT privilege is revoked from a role of which the authorization ID is a member
  - SELECT privilege is revoked from PUBLIC
- Table, view, or nickname privileges cannot be revoked from an *authorization-name* with CONTROL on the object without also revoking the CONTROL privilege (SQLSTATE 42504).
- Revoking a specific privilege does not necessarily revoke the ability to perform the action. A user can proceed with a task if other privileges are held by PUBLIC, a group, or a role, or if the user holds privileges such as ALTERIN on the schema of a table or a view.
- If the owner of the materialized query table loses a SELECT privilege on a table on which the materialized query table definition depends (or a table upon which the materialized query table definition depends is dropped), the materialized query table will be dropped.

  However, if a user who holds SECADM or ACCESSCTRL authority explicitly revokes all privileges on the materialized query table from the owner, then the record in SYSTABAUTH for the OWNER will be deleted, but nothing will happen to the materialized query table - it remains operative.
- Revoking nickname privileges has no affect on data source object (table or view) privileges.

- Revoking the SELECT privilege for a table or view that is directly or indirectly referenced in an SQL function or method body may fail if the SQL function or method body cannot be dropped because some other object is dependent on it (SQLSTATE 42893).
- Revoking the SELECT privilege causes an SQL function or method body to be dropped when:
  - The owner of the SQL function or method body loses the SELECT privilege on some object on which the SQL function or method body definition depends; note that the privilege can be lost because of a revoke from PUBLIC or from a role of which the owner is a member
  - An object on which the SQL function or method body definition depends is dropped

  However, the revoke fails if another object depends on the function or method (SQLSTATE 42893).
- **Revoking WITH GRANT OPTION:** The only way to revoke the WITH GRANT OPTION is to revoke the privilege itself and then grant it again without specifying WITH GRANT OPTION.
- **Revoking column privileges:** The only way to revoke column privileges is to revoke the privilege from the entire table itself and then grant it again for each column.

Examples

- *Example 1:* Revoke SELECT privilege on table EMPLOYEE from user ENGLES. There is one row in the SYSCAT.TABAUTH catalog view for this table and grantee and the GRANTEETYPE value is U.

```
REVOKE SELECT
  ON TABLE EMPLOYEE
  FROM ENGLES
```

- *Example 2:* Revoke update privileges on table EMPLOYEE previously granted to all local users. Note that grants to specific users are not affected.

```
REVOKE UPDATE
  ON EMPLOYEE
  FROM PUBLIC
```

- *Example 3:* Revoke all privileges on table EMPLOYEE from users PELLOW and MLI and from group PLANNERS.

```
REVOKE ALL
  ON EMPLOYEE
  FROM USER PELLOW, USER MLI, GROUP PLANNERS
```

- *Example 4:* Revoke SELECT privilege on table CORPDATA.EMPLOYEE from a user named JOHN. There is one row in the SYSCAT.TABAUTH catalog view for this table and grantee and the GRANTEETYPE value is U.

```
REVOKE SELECT
  ON CORPDATA.EMPLOYEE FROM JOHN
```

  or

```
REVOKE SELECT
  ON CORPDATA.EMPLOYEE FROM USER JOHN
```

  Note that an attempt to revoke the privilege from GROUP JOHN would result in an error, since the privilege was not previously granted to GROUP JOHN.
- *Example 5:* Revoke SELECT privilege on table CORPDATA.EMPLOYEE from a group named JOHN. There is one row in the SYSCAT.TABAUTH catalog view for this table and grantee and the GRANTEETYPE value is G.

```
REVOKE SELECT
  ON CORPDATA.EMPLOYEE FROM JOHN
```

  or

```
REVOKE SELECT
  ON CORPDATA.EMPLOYEE FROM GROUP JOHN
```

- *Example 6:* Revoke user SHAWN's privilege to create an index specification on nickname ORAREM1.

```
REVOKE INDEX
  ON ORAREM1 FROM USER SHAWN
```

REVOKE (workload privileges)

This form of the REVOKE statement revokes the USAGE privilege on a workload.

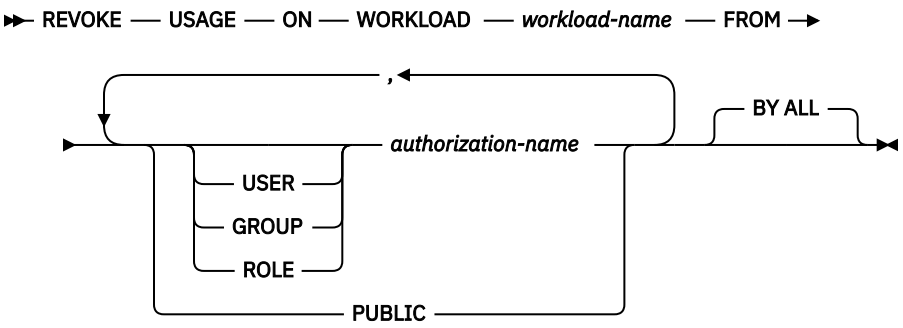
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include ACCESSCTRL, SECADM, or WLMADM authority.

Syntax



Description

USAGE
Revokes the privilege to use a workload.

ON WORKLOAD *workload-name*
Identifies the workload on which the USAGE privilege is to be revoked. This is a one-part name. The *workload-name* must identify a workload that exists at the current server (SQLSTATE 42704). The name cannot be 'SYSDEFAULTADMWORKLOAD' (SQLSTATE 42832).

FROM
Specifies from whom the USAGE privilege is revoked.

USER
Specifies that the *authorization-name* identifies a user.

GROUP
Specifies that the *authorization-name* identifies a group.

ROLE
Specifies that the *authorization-name* identifies an existing role at the current server (SQLSTATE 42704).

authorization-name,...
Lists the authorization IDs of one or more users, groups, or roles. The list of authorization IDs cannot include the authorization ID of the user issuing the statement (SQLSTATE 42502).

PUBLIC
Revokes the USAGE privilege from PUBLIC.

BY ALL
Revokes the USAGE privilege from all named users who were explicitly granted that privilege, regardless of who granted it. This is the default behavior.

Rules

- For each *authorization-name* specified, if none of the keywords USER, GROUP, or ROLE is specified, then for all rows for the specified object in the SYSCAT.WORKLOADAUTH catalog view where the grantee is *authorization-name*:
  - If GRANTEETYPE is 'U', USER is assumed.
  - If GRANTEETYPE is 'G', GROUP is assumed.
  - If GRANTEETYPE is 'R', ROLE is assumed.
  - If GRANTEETYPE does not have the same value, an error is returned (SQLSTATE 56092.

Notes

- The REVOKE statement does not take effect until it is committed, even for the connection that issues the statement.

Example

Revoke the privilege to use the workload CAMPAIGN from user LISA.

```
REVOKE USAGE ON WORKLOAD CAMPAIGN FROM USER LISA
```

REVOKE (XSR object privileges)

This form of the REVOKE statement revokes USAGE privilege on an XSR object.

Invocation

The REVOKE statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if the DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

One of the following authorities is required:

- ACCESSCTRL or SECADM authority or ACCESSCTRL authority on the schema containing the XSR object

Syntax

Description

ON XSROBJECT *xsrobject-name*
This name identifies the XSR object for which the USAGE privilege is revoked. The *xsrobject-name*, including the implicit or explicit schema qualifier, must uniquely identify an existing XSR object at the current server. If no XSR object by this name exists in the specified schema, an error is raised (SQLSTATE 42704).

FROM PUBLIC
Revokes the USAGE privilege from PUBLIC.

BY ALL
Revokes each named privilege from all users who were explicitly granted those privileges, regardless of who granted them. This is the default behavior.

Example

Revoke usage privileges on the XML schema MYSCHEMA from PUBLIC:

```
REVOKE USAGE ON XSROBJECT MYSCHEMA FROM PUBLIC
```

ROLLBACK

The ROLLBACK statement is used to back out of the database changes that were made within a unit of work or a savepoint.

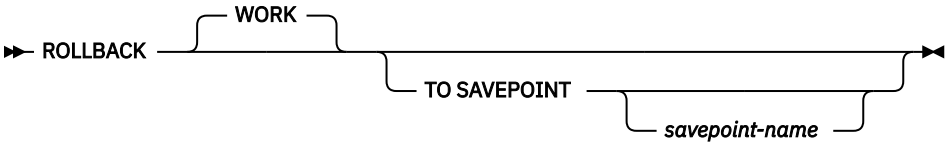
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

The unit of work in which the ROLLBACK statement is executed is terminated and a new unit of work is initiated. All changes made to the database during the unit of work are backed out.

The following statements, however, are not under transaction control, and changes made by them are independent of the ROLLBACK statement:

- SET CONNECTION
- SET ENCRYPTION PASSWORD
- SET EVENT MONITOR STATE
- SET PASSTHRU

  Note: Although the SET PASSTHRU statement is not under transaction control, the passthru session initiated by the statement is under transaction control.
- SET SERVER OPTION
- SET variable
- Assignments to updatable special registers

The generation of sequence and identity values is not under transaction control. Values generated and consumed by the *nextval-expression* or by inserting rows into a table that has an identity column are independent of issuing the ROLLBACK statement. Also, issuing the ROLLBACK statement does not affect the value returned by the *prevval-expression*, nor the IDENTITY_VAL_LOCAL function.

Modification of the values of global variables is not under transaction control. ROLLBACK statements do not affect the values assigned to global variables.

TO SAVEPOINT
Specifies that a partial rollback (ROLLBACK TO SAVEPOINT) is to be performed. If no savepoint is active in the current savepoint level (see the "Rules" section in the description of the SAVEPOINT statement), an error is returned (SQLSTATE 3B502). After a successful rollback, the savepoint continues to exist, but any nested savepoints are released and no longer exist. The nested savepoints, if any, are considered to have been rolled back and then released as part of the rollback to the current savepoint. If a *savepoint-name* is not provided, rollback occurs to the most recently set savepoint within the current savepoint level.

If this clause is omitted, the ROLLBACK statement rolls back the entire transaction. Furthermore, savepoints within the transaction are released.

savepoint-name
Specifies the savepoint that is to be used in the rollback operation. The specified *savepoint-name* cannot begin with 'SYS' (SQLSTATE 42939). After a successful rollback operation, the named savepoint continues to exist. If the savepoint name does not exist, an error (SQLSTATE 3B001) is returned. Data and schema changes made since the savepoint was set are undone.

Notes

- All locks held are released on a ROLLBACK of the unit of work. All open cursors are closed. All LOB locators are freed.
- Executing a ROLLBACK statement does not affect either the SET statements that change special register values or the RELEASE statement.
- If the program terminates abnormally, the unit of work is implicitly rolled back.
- Statement caching is affected by the rollback operation.
- The impact on cursors resulting from a ROLLBACK TO SAVEPOINT depends on the statements within the savepoint
  - If the savepoint contains DDL on which a cursor is dependent, the cursor is marked invalid. Attempts to use such a cursor results in an error (SQLSTATE 57007).
  - Otherwise:
    - If the cursor is referenced in the savepoint, the cursor remains open and is positioned before the next logical row of the result table. (A FETCH must be performed before a positioned UPDATE or DELETE statement is issued.)
    - Otherwise, the cursor is not affected by the ROLLBACK TO SAVEPOINT (it remains open and positioned).
- Dynamic SQL statements prepared in a package bound with the `KEEPDYNAMIC YES` option are kept in the SQL context after a ROLLBACK statement. The statement might be implicitly prepared again, as a result of DDL operations that are rolled back within the unit of work.
- Inactive dynamic SQL statements prepared in a package bound with `KEEPDYNAMIC NO` are removed from the SQL context after a rollback operation. The statement must be prepared again before it can be executed in a new transaction.
- The following dynamic SQL statements may be active during ROLLBACK:
  - ROLLBACK statement
  - CALL statements under which the ROLLBACK statement was executed
- A ROLLBACK TO SAVEPOINT operation will drop any created temporary tables created within the savepoint. If a created temporary table is modified within the savepoint and that table has been defined as not logged, then all rows in the table are deleted.
- A ROLLBACK TO SAVEPOINT operation will drop any declared temporary tables declared within the savepoint. If a declared temporary table is modified within the savepoint and that table has been defined as not logged, then all rows in the table are deleted.

- All locks are retained after a ROLLBACK TO SAVEPOINT statement.
- All LOB locators are preserved following a ROLLBACK TO SAVEPOINT operation.

Example

Delete the alterations made since the last commit point or rollback.

```
ROLLBACK WORK
```

SAVEPOINT

Use the SAVEPOINT statement to set a savepoint within a transaction.

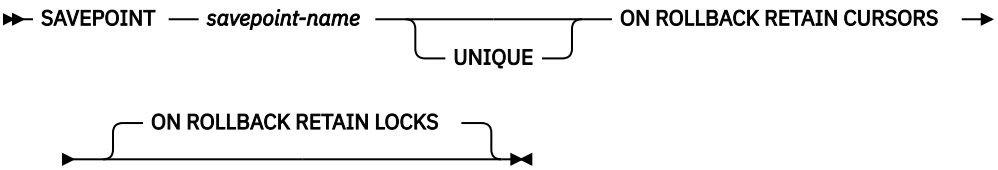
Invocation

This statement can be imbedded in an application program (including a procedure) or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

savepoint-name
Specifies the name of a savepoint. The specified *savepoint-name* cannot begin with 'SYS' (SQLSTATE 42939). If a savepoint by this name has already been defined as UNIQUE within this savepoint level, an error is returned (SQLSTATE 3B501).

UNIQUE
Specifies that the application does not intend to reuse this savepoint name while the savepoint is active within the current savepoint level. If *savepoint-name* already exists within this savepoint level, an error is returned (SQLSTATE 3B501).

ON ROLLBACK RETAIN CURSORS
Specifies system behavior upon rollback to this savepoint with respect to open cursor statements processed after the SAVEPOINT statement. This clause indicates that, whenever possible, the cursors are unaffected by a rollback to savepoint operation. For situations where the cursors are affected by the rollback to savepoint, see "ROLLBACK".

ON ROLLBACK RETAIN LOCKS
Specifies system behavior upon rollback to this savepoint with respect to locks acquired after the setting of the savepoint. Locks acquired since the savepoint are not tracked, and are not rolled back (released) upon rollback to the savepoint.

Rules

- Savepoint-related statements must not be used within trigger definitions (SQLSTATE 42987).
- A new savepoint level starts when one of the following events occurs:
  - A new unit of work (UOW) starts.

  - A procedure defined with the NEW SAVEPOINT LEVEL clause is called.
  - An atomic compound SQL statement starts.
- A savepoint level ends when the event that caused its creation is finished or removed. When a savepoint level ends, all savepoints contained within it are released. Any open cursors, DDL actions, or data modifications are inherited by the parent savepoint level (that is, the savepoint level within which the one that just ended was created), and are subject to any savepoint-related statements issued against the parent savepoint level.
- The following rules apply to actions within a savepoint level:
  - Savepoints can only be referenced within the savepoint level in which they are established. You cannot release, destroy, or roll back to a savepoint established outside of the current savepoint level.
  - All active savepoints established within the current savepoint level are automatically released when the savepoint level ends.
  - The uniqueness of savepoint names is only enforced within the current savepoint level. The names of savepoints that are active in other savepoint levels can be reused in the current savepoint level without affecting those savepoints in other savepoint levels.

Notes

- Once a SAVEPOINT statement has been issued, insert, update, or delete operations on nicknames are not allowed.
- Omitting the UNIQUE clause specifies that *savepoint-name* can be reused within the savepoint level by another savepoint. If a savepoint of the same name already exists within the savepoint level, the existing savepoint is destroyed and a new savepoint with the same name is created at the current point in processing. The new savepoint is considered to be the last savepoint established by the application. Note that the destruction of a savepoint through the reuse of its name by another savepoint simply destroys that one savepoint and does not release any savepoints established after the destroyed savepoint. These subsequent savepoints can only be released by means of the RELEASE SAVEPOINT statement, which releases the named savepoint and all savepoints established after the named savepoint.
- If the UNIQUE clause is specified, *savepoint-name* can only be reused after an existing savepoint with the same name has been released.
- Within a savepoint, if a utility, SQL statement, or database command performs intermittent commits during processing, the savepoint will be implicitly released.
- If the SET INTEGRITY statement is rolled back within the savepoint, dynamically prepared statement names are still valid, although the statement might be implicitly prepared again.
- If inserts are buffered (that is, the application was precompiled with the INSERT BUF option), the buffer will be flushed when SAVEPOINT, ROLLBACK, or RELEASE TO SAVEPOINT statements are issued.

Example

Perform a rollback operation for nested savepoints. First, create a table named DEPARTMENT. Insert a row before starting SAVEPOINT1; insert another row and start SAVEPOINT2; then, insert a third row and start SAVEPOINT3.

```
CREATE TABLE DEPARTMENT (
  DEPTNO   CHAR(6),
  DEPTNAME VARCHAR(20),
  MGRNO    INTEGER)

  INSERT INTO DEPARTMENT VALUES ('A20', 'MARKETING', 301)

  SAVEPOINT SAVEPOINT1 ON ROLLBACK RETAIN CURSORS

  INSERT INTO DEPARTMENT VALUES ('B30', 'FINANCE', 520)

  SAVEPOINT SAVEPOINT2 ON ROLLBACK RETAIN CURSORS

  INSERT INTO DEPARTMENT VALUES ('C40', 'IT SUPPORT', 430)
```

```
    SAVEPOINT SAVEPOINT3 ON ROLLBACK RETAIN CURSORS
    INSERT INTO DEPARTMENT VALUES ('R50', 'RESEARCH', 150)
```

At this point, the DEPARTMENT table exists with rows A20, B30, C40, and R50. If you now issue:

```
  ROLLBACK TO SAVEPOINT SAVEPOINT3
```

row R50 is no longer in the DEPARTMENT table. If you then issue:

```
  ROLLBACK TO SAVEPOINT SAVEPOINT1
```

the DEPARTMENT table still exists, but the rows inserted since SAVEPOINT1 was established (B30 and C40) are no longer in the table.

SELECT

The SELECT statement is a form of query

The SELECT statement can be embedded in an application program or issued interactively.

SELECT INTO

The SELECT INTO statement produces a result table consisting of at most one row, and assigns the values in that row to host variables.

If the table is empty, the statement assigns +100 to SQLCODE and '02000' to SQLSTATE and does not assign values to the host variables. If more than one row satisfies the search condition, statement processing is terminated, and an error occurs (SQLSTATE 21000).

Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- SELECT privilege on the table, view, or nickname
- CONTROL privilege on the table, view, or nickname
- SELECTIN privilege on the schema containing the table, view, or nickname
- DATAACCESS authority on the schema containing the table, view, or nickname
- DATAACCESS authority

For each global variable used as an assignment target, the privileges held by the authorization ID of the statement must include one of the following authorities:

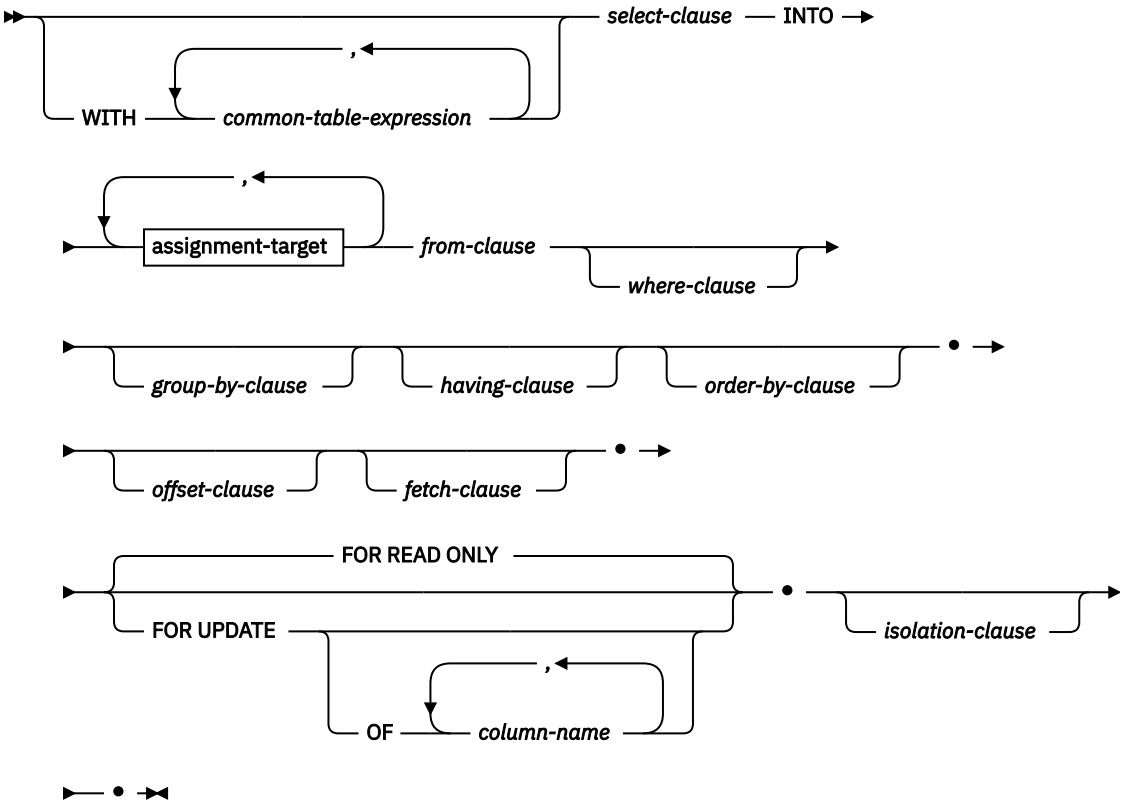
- WRITE privilege on the global variable that is not defined in a module
- EXECUTE privilege on the module of the global variable that is defined in a module
- EXECUTEIN privilege on the schema containing the module of the global variable that is defined in a module
- DATAACCESS authority on the schema containing the module of the global variable that is defined in a module

GROUP privileges are not checked for static SELECT INTO statements.

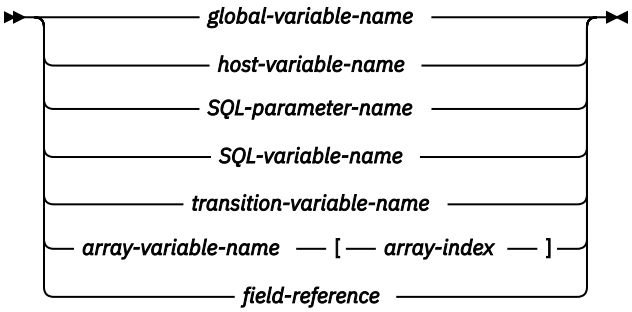
If the target of the SELECT INTO statement is a nickname, privileges on the object at the data source are not considered until the statement is executed at the data source. At this time, the authorization ID

that is used to connect to the data source must have the privileges that are required for the operation on the object at the data source. The authorization ID of the statement can be mapped to a different authorization ID at the data source.

Syntax



assignment-target



Description

For a description of the *common-table-expression*, *select-clause*, *from-clause*, *where-clause*, *group-by-clause*, *having-clause*, *order-by-clause*, *offset-clause*, *fetch-clause*, and *isolation-clause*, see "subselect" in the *SQL Reference Volume 1*.

INTO *assignment-target*

Identifies one or more targets for the assignment of output values.

The first value in the result row is assigned to the first target in the list, the second value to the second target, and so on. Each assignment to an *assignment-target* is made in sequence through the list. If an error occurs on any assignment, no value is assigned to any *assignment-target*.

When the data type of every *assignment-target* is not a row type, then the value 'W' is assigned to the SQLWARN3 field of the SQLCA if the number of *assignment-targets* is less than the number of result column values.

If the data type of an *assignment-target* is a row type, then there must be exactly one *assignment-target* specified (SQLSTATE 428HR), the number of columns must match the number of fields in the row type, and the data types of the columns of the fetched row must be assignable to the corresponding fields of the row type (SQLSTATE 42821).

If the data type of an *assignment-target* is an array element, then there must be exactly one *assignment-target* specified.

global-variable-name
Identifies the global variable that is the assignment target.

host-variable-name
Identifies the host variable that is the assignment target. For LOB output values, the target can be a regular host variable (if it is large enough), a LOB locator variable, or a LOB file reference variable.

SQL-parameter-name
Identifies the parameter that is the assignment target.

SQL-variable-name
Identifies the SQL variable that is the assignment target. SQL variables must be declared before they are used.

transition-variable-name
Identifies the column to be updated in the transition row. A *transition-variable-name* must identify a column in the subject table of a trigger, optionally qualified by a correlation name that identifies the new value.

array-variable-name
Identifies an SQL variable, SQL parameter, or global variable of an array type.

[array-index]
An expression that specifies which element in the array will be the target of the assignment. For an ordinary array, the *array-index* expression must be assignable to INTEGER (SQLSTATE 428H1) and cannot be the null value. Its value must be between 1 and the maximum cardinality defined for the array (SQLSTATE 2202E). For an associative array, the *array-index* expression must be assignable to the index data type of the associative array (SQLSTATE 428H1) and cannot be the null value.

field-reference
Identifies the field within a row type value that is the assignment target. The *field-reference* must be specified as a qualified *field-name* where the qualifier identifies the row value in which the field is defined.

FOR READ ONLY or FOR UPDATE
Indicates the intended use for the selected row. The default is FOR READ ONLY.

FOR READ ONLY
Specifies that the selected row will not be locked for update.

FOR UPDATE
Specifies that the selected row from the underlying table will be locked to facilitate updating the row later on in the transaction, similar to the locking done for the select statement of a cursor which includes the FOR UPDATE clause.

FOR UPDATE must not be specified if the result table of the SELECT INTO statement is read-only (SQLSTATE 42829).

If *column-name* values are listed, these columns must be updatable (SQLSTATE 42829).

Note that listing columns has only documentary effect and does not limit subsequent searched update statements from modifying other columns.

Rules

- Global variables cannot be assigned inside triggers that are not defined using a compound SQL (compiled) statement, functions that are not defined using a compound SQL (compiled) statement, methods, or compound SQL (inlined) statements (SQLSTATE 428GX).

Notes

- **Syntax alternatives:** For consistency with SQL queries:
  - FOR FETCH ONLY can be specified in place of FOR READ ONLY

Examples

- *Example 1:* This C example puts the maximum salary in the EMP table into the host variable MAXSALARY.

```
EXEC SQL SELECT MAX(SALARY)
  INTO :MAXSALARY
  FROM EMP;
```

- *Example 2:* This C example puts the row for employee 528671 (from the EMP table) into host variables.

```
EXEC SQL SELECT * INTO :h1, :h2, :h3, :h4
  FROM EMP
  WHERE EMPNO = '528671';
```

- *Example 3:* This SQLJ example puts the row for employee 528671 (from the EMP table) into host variables. That row will later be updated using a searched update, and should be locked when the query executes.

```
#sql { SELECT * INTO :FIRSTNAME, :LASTNAME, :EMPNO, :SALARY
  FROM EMP
  WHERE EMPNO = '528671'
  FOR UPDATE };
```

- *Example 4:* This C example puts the maximum salary in the EMP table into the global variable GV_MAXSALARY.

```
EXEC SQL SELECT MAX(SALARY)
  INTO GV_MAXSALARY
  FROM EMP;
```

SET COMPILATION ENVIRONMENT

The SET COMPILATION ENVIRONMENT statement changes the current compilation environment in the connection to match the values contained in the compilation environment provided by an event monitor.

The compilation environment contains information like schema, isolation level, query degree or function path under which a SQL statement has been compiled (prepared). It allows you to run and explain SQL statements within a given environment. You can run the **db2caem** command or the **db2support** command with the **-compenv** parameter to specify a file containing a BLOB data type with the compilation environment. Follow these steps to create the file:

1. Set the environment (for example, to **SET CURRENT SCHEMA**, **CHANGE ISOLATION**, **SET CURRENT DEGREE**, or **SET PATH**.
2. Run a SQL statement
3. Export the compilation environment into the `lob` file in directory `lobs`:

```
EXPORT TO compenv.ixf OF IXF LOBS TO lobs
SELECT COMP_ENV_DESC
FROM TABLE (MON_GET_PKG_CACHE_STMT('d', null, null, -1)) AS tf
WHERE STMT_TEXT = '' ;
```

4. Use compilation environment BLOB file for other SQL statements for **db2caem** and **db2support** and specify:

```
-compenv lobs/compenv.ixf.001.lob
```

For more information, see COMPILATION_ENV table function

This statement changes the values of one or more special registers; these changes, in turn, will affect the compilation of any subsequent dynamic SQL statement.

This statement is not under transaction control.

Invocation

The statement can be embedded in an application program. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax

Description

host-variable
A variable of type BLOB containing a compilation environment provided by an event monitor. It cannot be set to null. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815). If the format of the compilation environment is incorrect, an error is returned, and the connection settings remain unmodified (SQLSTATE 51040).

Notes

- To reset the compilation environment to the original default values, terminate and then restart the connection. You can achieve the same effect by issuing this statement within an SQL routine, so that any special register changes are not reflected in the connection upon return from that routine.
- Use the COMPILATION_ENV table function to look at the individual elements that are contained within the compilation environment.

Example

Set the current session's compilation environment to the values contained in a compilation environment that was previously captured by a deadlock event monitor. A deadlock event monitor that is created specifying the WITH DETAILS HISTORY option will capture the compilation environment for dynamic SQL statements. This captured environment is what is accepted as input to the statement.

```
SET COMPILATION ENVIRONMENT = :hv1
```

SET CONNECTION

The SET CONNECTION statement changes the state of a connection from dormant to current, making the specified location the current server.

This statement is not under transaction control.

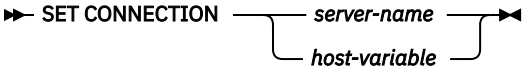
Invocation

Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax



Description

server-name* or *host-variable

Identifies the application server by the specified *server-name* or a *host-variable* which contains the *server-name*.

If a *host-variable* is specified, it must be a character string variable with a length attribute that is not greater than 8, and it must not include an indicator variable. The *server-name* that is contained within the *host-variable* must be left-aligned and must not be delimited by quotation marks.

Note that the *server-name* is a database alias identifying the application server. It must be listed in the application requester's local directory.

The *server-name* or the *host-variable* must identify an existing connection of the application process. If they do not identify an existing connection, an error (SQLSTATE 08003) is raised.

If SET CONNECTION is to the current connection, the states of all connections of the application process are unchanged.

Successful Connection

If the SET CONNECTION statement executes successfully:

- No connection is made. The CURRENT SERVER special register is updated with the specified *server-name*.
- The previously current connection, if any, is placed into the dormant state (assuming a different *server-name* is specified).
- The CURRENT SERVER special register and the SQLCA are updated in the same way as documented under "CONNECT (Type 1)".

Unsuccessful Connection

If the SET CONNECTION statement fails:

- No matter what the reason for failure, the connection state of the application process and the states of its connections are unchanged.
- As with an unsuccessful Type 1 CONNECT, the SQLERRP field of the SQLCA is set to the name of the module that detected the error.

Notes

- The use of type 1 CONNECT statements does not preclude the use of SET CONNECTION, but the statement will always fail (SQLSTATE 08003), unless the SET CONNECTION statement specifies the current connection, because dormant connections cannot exist.
- The SQLRULES(DB2) connection option (see "Options that Govern Distributed Unit of Work Semantics") does not preclude the use of SET CONNECTION, but the statement is unnecessary, because type 2 CONNECT statements can be used instead.

- When a connection is used, made dormant, and then restored to the current state in the same unit of work, that connection reflects its last use by the application process with regard to the status of locks, cursors, and prepared statements.

Example

Execute SQL statements at IBMSTHDB, execute SQL statements at IBMTOKDB, and then execute more SQL statements at IBMSTHDB.

```
EXEC SQL CONNECT TO IBMSTHDB;
/* Execute statements referencing objects at IBMSTHDB */
```

```
EXEC SQL CONNECT TO IBMTOKDB;
/* Execute statements referencing objects at IBMTOKDB */
```

```
EXEC SQL SET CONNECTION IBMSTHDB;
/* Execute statements referencing objects at IBMSTHDB */
```

Note that the first CONNECT statement creates the IBMSTHDB connection, the second CONNECT statement places it in the dormant state, and the SET CONNECTION statement returns it to the current state.

SET CURRENT DECFLOAT ROUNDING MODE

The SET CURRENT DECFLOAT ROUNDING MODE statement verifies that the specified rounding mode is the value that is currently set for the CURRENT DECFLOAT ROUNDING MODE special register.

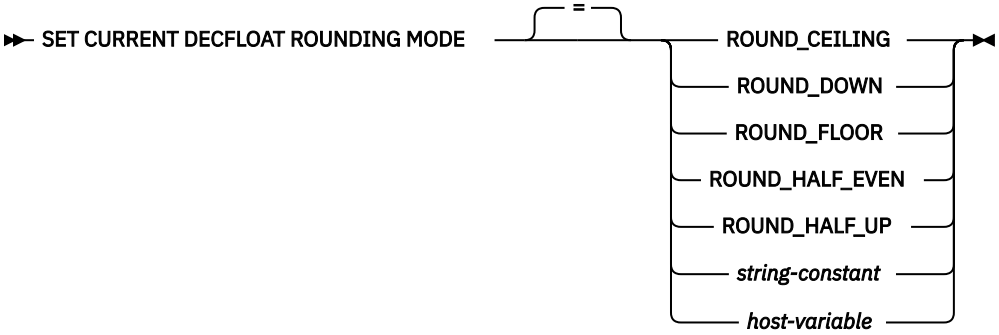
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

ROUND_CEILING
Round the value toward positive infinity. If all of the discarded digits are zero or if the sign is negative, the result is unchanged (except for the removal of the discarded digits). Otherwise, the result coefficient is incremented by 1.

ROUND_DOWN
Round the value toward 0 (truncation). The discarded digits are ignored.

ROUND_FLOOR
Round the value toward negative infinity. If all of the discarded digits are zero or if the sign is positive, the result is unchanged (except for the removal of the discarded digits). Otherwise, the sign is negative and the result coefficient is incremented by 1.

ROUND_HALF_EVEN
Round the value to the nearest value. If the values are equidistant, round the value so that the final digit is even. If the discarded digits represent more than half of the value of a number in the next left position, the result coefficient is incremented by 1. If they represent less than half, the result coefficient is not adjusted (that is, the discarded digits are ignored). Otherwise, the result coefficient is unaltered if its rightmost digit is even, or incremented by 1 if its rightmost digit is odd (to make an even digit).

ROUND_HALF_UP
Round the value to the nearest value. If the values are equidistant, round the value up. If the discarded digits represent half or more than half of the value of a number in the next left position, the result coefficient is incremented by 1. Otherwise, the discarded digits are ignored.

string-constant
A character string constant with a maximum length of 15 bytes, after trailing blanks have been removed. The value must be a left-aligned string that specifies one of the five rounding mode keywords (case insensitive).

host-variable
A variable of type CHAR or VARCHAR. The value of the host variable must be a left-aligned string that specifies one of the five rounding mode keywords (case insensitive). The actual length of the contents of *host-variable* must not be greater than 15 bytes, after trailing blanks have been removed. The value must be padded on the right with blanks when using a fixed-length character host variable. The host variable cannot be set to the null value.

Rules

- The specified rounding mode value must be the same as the value of the CURRENT DECFLOAT ROUNDING MODE special register (SQLSTATE 42815).

Notes

- This statement does not change the value of the CURRENT DECFLOAT ROUNDING MODE special register on a Db2 server. However, when the statement is processed by a Db2 for z/OS server or a Db2 for IBM i server, it can be used to change the value of the CURRENT DECFLOAT ROUNDING MODE special register on that server.

Example

The following statement verifies whether the specified rounding mode value for the client matches the rounding mode value that is currently set on the server.

```
SET CURRENT DECFLOAT ROUNDING MODE = ROUND_CEILING
```

SET CURRENT DEFAULT TRANSFORM GROUP

The SET CURRENT DEFAULT TRANSFORM GROUP statement changes the value of the CURRENT DEFAULT TRANSFORM GROUP special register.

This statement is not under transaction control.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax

Description

group-name
Specifies a one-part name that identifies a transform group defined for all structured types. This name can be referenced in subsequent statements (or until the special register value is changed again using another SET CURRENT DEFAULT TRANSFORM GROUP statement).

The name must be an SQL identifier (either ordinary or delimited). No validation that the *group-name* is defined for any structured type is made when the special register is set. Only when a structured type is specifically referenced is the definition of the named transform group checked for validity.

Rules

- If the value specified does not conform to the rules for a *group-name*, an error is raised (SQLSTATE 42815)
- The TO SQL and FROM SQL functions defined in the *group-name* transform group are used for exchanging user-defined structured type data with a host program.

Usage notes

- The initial value of the CURRENT DEFAULT TRANSFORM GROUP special register is the empty string.

Example

Set the default transform group to MYSTRUCT1. The TO SQL and FROM SQL functions defined in the MYSTRUCT1 transform group will be used for exchanging user-defined structured type variables with the current host program.

```
SET CURRENT DEFAULT TRANSFORM GROUP = MYSTRUCT1
```

SET CURRENT DEGREE

The SET CURRENT DEGREE statement assigns a value to the CURRENT DEGREE special register.

This statement is not under transaction control.

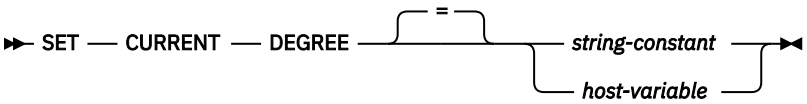
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

The value of CURRENT DEGREE is replaced by the value of the string constant or host variable. The value must be a character string that is not longer than 5 bytes. The value must be the character string representation of an integer between 1 and 32 767 inclusive or 'ANY'.

If the value of CURRENT DEGREE represented as an integer is 1 when an SQL statement is dynamically prepared, the execution of that statement will not use intrapartition parallelism.

If the value of CURRENT DEGREE is a number when an SQL statement is dynamically prepared, the execution of that statement can involve intrapartition parallelism with the specified degree.

If the value of CURRENT DEGREE is 'ANY' when an SQL statement is dynamically prepared, the execution of that statement can involve intrapartition parallelism using a degree determined by the database manager.

host-variable
The *host-variable* must be of data type CHAR or VARCHAR and the length must not exceed 5. If a longer field is provided, an error will be returned (SQLSTATE 42815). If the actual value provided is larger than the replacement value specified, the input must be padded on the right with blanks. Leading blanks are not allowed (SQLSTATE 42815). All input values are treated as being case-insensitive. If a *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

string-constant
The *string-constant* length must not exceed 5.

Notes

- The degree of intrapartition parallelism for static SQL statements can be controlled using the DEGREE option of the PREP or BIND command.
- The actual runtime degree of intrapartition parallelism will be the lower of:
  - Maximum query degree (**max_querydegree**) configuration parameter
  - Application runtime degree
  - SQL statement compilation degree
  - MAXIMUM DEGREE service class option
  - MAXIMUM DEGREE workload option
- The **intra_parallel** database manager configuration parameter must be on to use intrapartition parallelism. If it is set to off, the value of this register will be ignored and the statement will not use intrapartition parallelism for the purpose of optimization (SQLSTATE 01623).
- The value in the CURRENT DEGREE special register and the **intra_parallel** setting can be overridden in a workload by setting the MAXIMUM DEGREE workload attribute.
- If the DB2_WORKLOAD system environment variables is set to ANALYTICS and MAXIMUM DEGREE for the workload is set to DEFAULT, the value of the **intra_parallel** setting for the workload is overridden to ON.
- Some SQL statements cannot use intrapartition parallelism.

Examples

- *Example 1:* The following statement sets the CURRENT DEGREE to inhibit intrapartition parallelism.

```
SET CURRENT DEGREE = '1'
```

- *Example 2:* The following statement sets the CURRENT DEGREE to allow intrapartition parallelism.

```
SET CURRENT DEGREE = 'ANY'
```

SET CURRENT EXPLAIN MODE

The SET CURRENT EXPLAIN MODE statement changes the value of the CURRENT EXPLAIN MODE special register. It is not under transaction control.

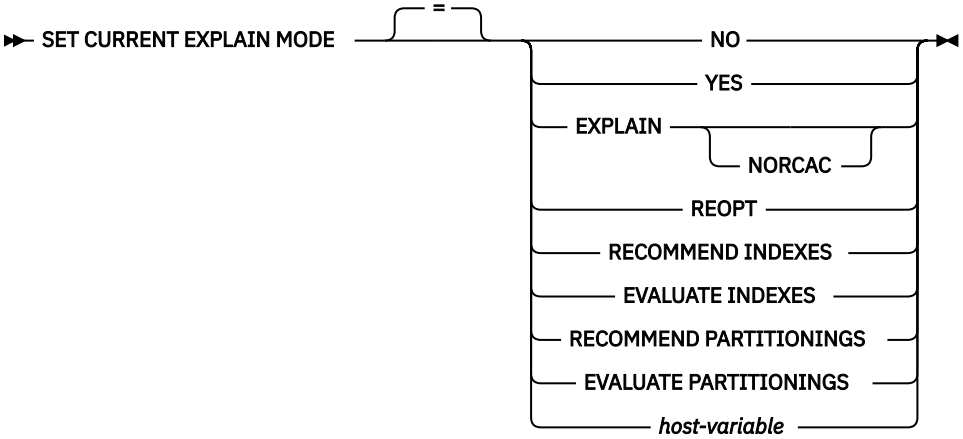
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

NO
Disables the Explain facility. No Explain information is captured. NO is the initial value of the special register.

YES
Enables the Explain facility and causes Explain information to be inserted into the Explain tables for eligible dynamic SQL statements. All dynamic SQL statements are compiled and executed normally.

EXPLAIN
Enables the Explain facility and causes Explain information to be captured for any eligible dynamic SQL statement that is prepared. However, dynamic statements are not executed.

EXPLAIN NORCAC
Enables the Explain facility and causes Explain information to be captured for any eligible dynamic SQL statement that is prepared as if row or column access control (RCAC) was not activated. Dynamic statements are not executed. When this explain mode is set, explain facility would explain the plan as if RCAC was not present.

REOPT
Enables the Explain facility and causes Explain information to be captured for a static or dynamic SQL statement during statement reoptimization at execution time; that is, when actual values for the host variables, special registers, global variables, or parameter markers are available.

RECOMMEND INDEXES
Enables the SQL compiler to recommend indexes. All queries that are executed in this explain mode will populate the ADVISE_INDEX table with recommended indexes. In addition, Explain information will be captured in the Explain tables to reveal how the recommended indexes are used, but the statements are neither compiled nor executed.

EVALUATE INDEXES
Enables the SQL compiler to evaluate virtual recommended indexes for dynamic queries. Queries executed in this explain mode will be compiled and optimized using fabricated statistics based on the virtual indexes. The statements are not executed. The indexes to be evaluated are read from the ADVISE_INDEX table if the USE_INDEX column contains "Y". Existing non-unique indexes can also be ignored by setting the USE_INDEX column to "I" and the EXISTS column to "Y". If a combination of USE_INDEX="I" and EXISTS="N" is given then index evaluation for the query will continue normally but the index in question will not be ignored.

RECOMMEND PARTITIONINGS
Specifies that the compiler is to recommend the best database partition for each table that is accessed by a specific query. The best database partitions are then written to an ADVISE_PARTITION table. The query is not executed.

EVALUATE PARTITIONINGS
Specifies that the compiler is to obtain the estimated performance of a query using the virtual database partitions specified in the ADVISE_PARTITION table.

host-variable
The *host-variable* must be of data type CHAR or VARCHAR and the length must not exceed 254. If a longer field is provided, an error will be returned (SQLSTATE 42815). The value specified must be NO, YES, EXPLAIN, RECOMMEND INDEXES, or EVALUATE INDEXES. If the actual value provided is larger than the replacement value specified, the input must be padded on the right with blanks. Leading blanks are not allowed (SQLSTATE 42815). All input values are treated as being case-insensitive. If a *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

Notes

- The Explain facility uses the following IDs as the schema when qualifying Explain tables that it is populating:
  - The session authorization ID for dynamic SQL
  - The statement authorization ID for static SQL

  The schema can be associated with a set of Explain tables, or aliases that point to a set of Explain tables under a different schema. If no Explain tables are found under the schema, the Explain facility checks for Explain tables under the SYSTOOLS schema and attempts to use those tables.
- Explain information for static SQL statements can be captured by using the **EXPLAIN** option of the **PREP** or **BIND** command. If the ALL value of the **EXPLAIN** option is specified, and the CURRENT EXPLAIN MODE register value is NO, explain information will be captured for dynamic SQL statements at run time. If the value of the CURRENT EXPLAIN MODE register is not NO, the value of the **EXPLAIN** bind option is ignored.
- RECOMMEND INDEXES and EVALUATE INDEXES are special modes which can only be set with the SET CURRENT EXPLAIN MODE statement. These modes cannot be set using **PREP** or **BIND** options, and they do not work with the SET CURRENT EXPLAIN SNAPSHOT statement.
- If the Explain facility is activated, the current authorization ID must have INSERT privilege for the Explain tables, or an error (SQLSTATE 42501) is raised.

- When SQL statements are explained from a routine, the routine must be defined with an SQL data access indicator of MODIFIES SQL DATA (SQLSTATE 42985).
- If the special register is set to REOPT, and the SQL statement does not qualify for reoptimization at execution time (that is, if the statement does not have input variables, or if the **REOPT** bind option is set to NONE), then no Explain information will be captured. If the **REOPT** bind option is set to ONCE, Explain information will be captured only once when the statement is initially reoptimized. After the statement is cached, no further Explain information will be acquired for this statement on subsequent executions.
- If the Explain facility is enabled, the **REOPT** bind option is set to ONCE, and you attempt to execute an SQL statement that is already cached, the statement will be compiled and reoptimized with the current values of the input variables, and the Explain tables will be populated accordingly. The newly generated access plan for this statement will not be cached or executed. Other applications that are concurrently executing this cached statement will continue to execute, and new requests to execute this statement will pick up the already cached access plan.
- A value of REOPT for the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special registers will override the value of the **EXPLAIN** and **EXPLSNAP** bind options at bind time if a static or dynamic SQL statement has input variables, and the **REOPT** bind option is set to ONCE or ALWAYS.
- Row and column level access control (RCAC) defined on the EXPLAIN tables is enforced for user access to these tables just like any other regular tables. However, row and column level access control on the EXPLAIN tables is not enforced when the database itself is populating those EXPLAIN tables. This is considered internal housekeeping and is not subject to RCAC, much like internal SQL.

Example

The following statement sets the CURRENT EXPLAIN MODE special register, so that Explain information will be captured for any subsequent eligible dynamic SQL statements and the statement will not be executed.

```
SET CURRENT EXPLAIN MODE = EXPLAIN
```

SET CURRENT EXPLAIN SNAPSHOT

The SET CURRENT EXPLAIN SNAPSHOT statement changes the value of the CURRENT EXPLAIN SNAPSHOT special register.

This statement is not under transaction control.

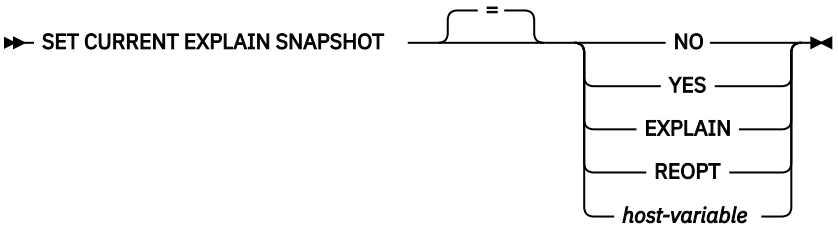
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

NO
Disables the Explain snapshot facility. No snapshot is taken. NO is the initial value of the special register.

YES
Enables the Explain snapshot facility, creating a snapshot of the internal representation for each eligible dynamic SQL statement. This information is inserted in the SNAPSHOT column of the EXPLAIN_STATEMENT table.

EXPLAIN
Enables the Explain snapshot facility, creating a snapshot of the internal representation for each eligible dynamic SQL statement that is prepared. However, dynamic statements are not executed.

REOPT
Enables the Explain facility and causes Explain information to be captured for a static or dynamic SQL statement during statement reoptimization at execution time; that is, when actual values for the host variables, special registers, global variables, or parameter markers are available.

host-variable
The *host-variable* must be of data type CHAR or VARCHAR and the length of its contents must not exceed 8. If a longer field is provided, an error will be returned (SQLSTATE 42815). The value contained in this register must be either NO, YES, or EXPLAIN. If the actual value provided is larger than the replacement value specified, the input must be padded on the right with blanks. Leading blanks are not allowed (SQLSTATE 42815). All input values are treated as being case-insensitive. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

Notes

- The Explain facility uses the following IDs as the schema when qualifying Explain tables that it is populating:
  - The session authorization ID for dynamic SQL
  - The statement authorization ID for static SQL

  The schema can be associated with a set of Explain tables, or aliases that point to a set of Explain tables under a different schema. If no Explain tables are found under the schema, the Explain facility checks for Explain tables under the SYSTOOLS schema and attempts to use those tables.
- Explain snapshots for static SQL statements can be captured by using the EXPLSNAP option of the PREP or BIND command. If the ALL value of the EXPLSNAP option is specified, and the CURRENT EXPLAIN SNAPSHOT register value is NO, Explain snapshots will be captured for dynamic SQL statements at run time. If the value of the CURRENT EXPLAIN SNAPSHOT register is not NO, the EXPLSNAP option is ignored.
- If the Explain snapshot facility is activated, the current authorization ID must have INSERT privilege for the Explain tables or an error (SQLSTATE 42501) is raised.
- When SQL statements are explained from a routine, the routine must be defined with an SQL data access indicator of MODIFIES SQL DATA (SQLSTATE 42985).
- If the special register is set to REOPT, and the SQL statement does not qualify for reoptimization at execution time (that is, if the statement does not have input variables, or if the REOPT bind option is set to NONE), then no Explain information will be captured. If the REOPT bind option is set to ONCE, Explain snapshot information will be captured only once when the statement is initially reoptimized. After the statement is cached, no further Explain information will be acquired for this statement on subsequent executions.
- If the Explain facility is enabled, the REOPT bind option is set to ONCE, and you attempt to execute a reoptimizable SQL statement that is already cached, the statement will be compiled and reoptimized with the current values of the input variables, and the Explain snapshot will be captured accordingly. The newly generated access plan for this statement will not be cached or executed. Other applications

that are concurrently executing this cached statement will continue to execute, and new requests to execute this statement will pick up the already cached access plan.

- The value REOPT for the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special registers will override the value of the EXPLAIN and EXPLSNAP bind options at bind time if a static or dynamic SQL statement has input variables, and the REOPT bind option is set to ONCE or ALWAYS.

Examples

- *Example 1:* The following statement sets the CURRENT EXPLAIN SNAPSHOT special register, so that an Explain snapshot will be taken for any subsequent eligible dynamic SQL statements and the statement will be executed.

```
SET CURRENT EXPLAIN SNAPSHOT = YES
```

- *Example 2:* The following example retrieves the current value of the CURRENT EXPLAIN SNAPSHOT special register into the host variable called SNAP.

```
EXEC SQL VALUES (CURRENT EXPLAIN SNAPSHOT) INTO :SNAP;
```

SET CURRENT FEDERATED ASYNCHRONY

The SET CURRENT FEDERATED ASYNCHRONY statement assigns a value to the CURRENT FEDERATED ASYNCHRONY special register.

This statement is not under transaction control.

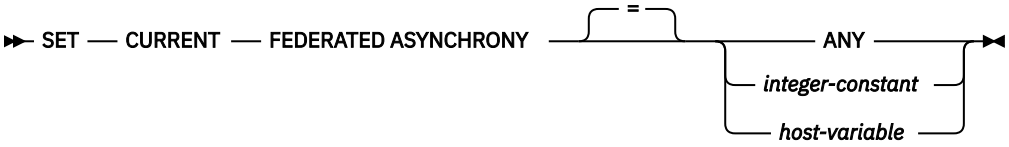
Invocation

The statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

ANY
Specifies a CURRENT FEDERATED ASYNCHRONY value of -1, which means that the execution of statements can involve asynchrony using a degree that is determined by the database manager.

integer-constant
Specifies an integer value between 0 and 32 767, inclusive. The execution of statements can involve asynchrony using the specified degree. If the value is 0 when an SQL statement is dynamically prepared, the execution of that statement will not use asynchrony.

host-variable
A variable of type INTEGER. The value must be between 0 and 32 767, inclusive, or -1 (representing ANY). If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

Notes

- The degree of asynchrony for static SQL statements can be controlled using the FEDERATED_ASYNCHRONY option of the PREP or BIND command.
- The initial value of the CURRENT FEDERATED ASYNCHRONY special register is determined by the **federated_async** database manager configuration parameter if the dynamic statement is issued through the command line processor (CLP). The initial value is determined by the FEDERATED_ASYNCHRONY bind option if the dynamic statement is part of an application that is being bound.

Examples

- *Example 1:* The following statement disables asynchrony by setting the value of the CURRENT FEDERATED ASYNCHRONY special register to 0.

```
SET CURRENT FEDERATED ASYNCHRONY = 0
```

- *Example 2:* The following statement sets the degree of asynchrony to 5.

```
SET CURRENT FEDERATED ASYNCHRONY 5
```

- *Example 3:* The following statement sets the value of the CURRENT FEDERATED ASYNCHRONY special register to -1, which specifies that the database manager is to determine the degree of asynchrony.

```
SET CURRENT FEDERATED ASYNCHRONY ANY
```

SET CURRENT IMPLICIT XMLPARSE OPTION

The SET CURRENT IMPLICIT XMLPARSE OPTION statement changes the value of the CURRENT IMPLICIT XMLPARSE OPTION special register.

This statement is not under transaction control.

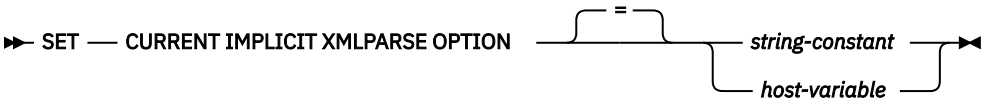
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

string-constant
A character string constant. The value must be a left-aligned string that is either 'PRESERVE WHITESPACE' or 'STRIP WHITESPACE' (case insensitive) with no additional blank characters between the keywords.

host-variable
A variable of type CHAR or VARCHAR. The value of the host variable must be a left-aligned string that is either 'PRESERVE WHITESPACE' or 'STRIP WHITESPACE' (case insensitive) with no additional blank

characters between the keywords. The value must be padded on the right with blanks when using a fixed-length character *host-variable*. The host variable cannot be set to null.

Notes

- The initial value of the CURRENT IMPLICIT XMLPARSE OPTION special register is 'STRIP WHITESPACE'.
- Both dynamic and static SQL statements are affected by this special register.

Example

Set the value of the CURRENT IMPLICIT XMLPARSE OPTION special register to 'PRESERVE WHITESPACE'.

```
SET CURRENT IMPLICIT XMLPARSE OPTION = 'PRESERVE WHITESPACE'
```

SET CURRENT ISOLATION

The SET CURRENT ISOLATION statement assigns a value to the CURRENT ISOLATION special register.

This statement is not under transaction control.

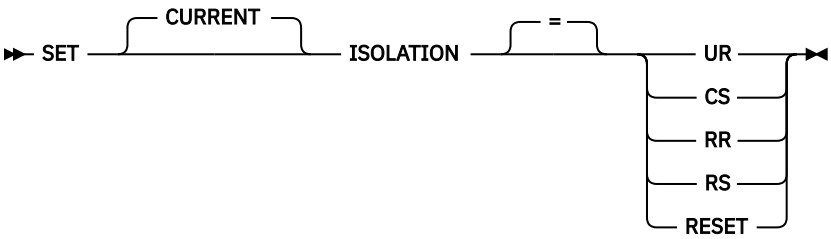
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

The value of the CURRENT ISOLATION special register is replaced by the specified value or set to blanks if RESET is specified.

Notes

- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products.
  - TO can be specified in place of the equal sign (=)
  - DIRTY READ can be specified in place of UR
  - READ UNCOMMITTED can be specified in place of UR
  - READ COMMITTED is recognized and upgraded to CS
  - CURSOR STABILITY can be specified in place of CS
  - REPEATABLE READ can be specified in place of RR
  - SERIALIZABLE can be specified in place of RR

SET CURRENT LOCALE LC_MESSAGES

The SET CURRENT LOCALE LC_MESSAGES statement changes the value of the CURRENT LOCALE LC_MESSAGES special register.

This statement is not under transaction control.

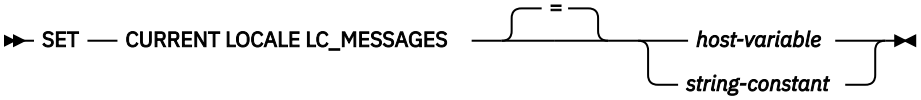
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

The CURRENT LOCALE LC_MESSAGES special register identifies the locale that is used by EVMON_UPGRADE_TABLES, as well as monitoring routines in the **monreport** module. EVMON_UPGRADE_TABLES and the monitoring routines use the value of CURRENT LOCALE LC_MESSAGES to determine in which language the result set text output should be returned. User-defined routines that are coded to return messages could also use the value of CURRENT LOCALE LC_MESSAGES to determine what language to use for message text.

host-variable
A variable of type CHAR or VARCHAR. It cannot be set to null.

string-constant
A character string constant.

Notes

- ***Initial value:*** The initial value of the CURRENT LOCALE LC_MESSAGES special register is `'en_US'`.
- ***Language availability:*** If the language for the locale is not available to the database manager, messages will be returned in English.
- ***Code page compatibility:*** The language for the locale specified must be supported by the code page of the output parameter or returns type of a routine that uses the special register to determine what language to return message text information in. If the database is not a Unicode database (and the routine was not created with PARAMETER CCSID UNICODE) and some characters in the language for the locale cannot be represented in the database code page, substitution characters will be returned as a result of code page conversion.
- ***Potential future use:*** In a future release, the value of the CURRENT LOCALE LC_MESSAGES special register might be used for other areas of the database environment that involve messages.
- ***Valid locales and naming:*** For information about valid locales and their naming, see "Locale names for SQL and XQuery" in the *Globalization Guide*.

Examples

- *Example 1:* The following statement sets the CURRENT LOCALE LC_MESSAGES special register to the English (Canada) locale using the latest version of Common Locale Data Repository (CLDR) available in the database manager.

```
SET CURRENT LOCALE LC_MESSAGES = 'en_CA'
```

- *Example 2:* The following statement sets the CURRENT LOCALE LC_MESSAGES special register to the French (France) locale using Common Locale Data Repository (CLDR) version 1.5. The CONNECTION routine in the **monreport** module is then invoked to have its output returned in French.

```
SET CURRENT LOCALE LC_MESSAGES = 'CLDR 1.5:fr_FR'
CALL MONREPORT.CONNECTION
```

- *Example 3:* Assume that the user-defined procedure XYZ.STORELOCATOR takes a zip code or postal code input. It returns a result set of stores of the XYZ company within a 30 minute drive from the zip code or postal code given as input. If the zip code or postal code is not in the correct format, an error message is returned that indicates what the problem is with the format. The procedure is coded to be able to return the error message in the language determined from the value of the CURRENT LOCALE LC_MESSAGES special register. The following statement sets the CURRENT LOCALE LC_MESSAGES special register to the Spanish (Mexico) locale. The store locator user-defined procedure is then invoked and any error messages will be returned in Spanish.

```
SET CURRENT LOCALE LC_MESSAGES = 'es_MX'
CALL XYZ.STORELOCATOR(:ZIP, :STATUSMSG)
```

SET CURRENT LOCALE LC_TIME

The SET CURRENT LOCALE LC_TIME statement changes the value of the CURRENT LOCALE LC_TIME special register. It is not under transaction control.

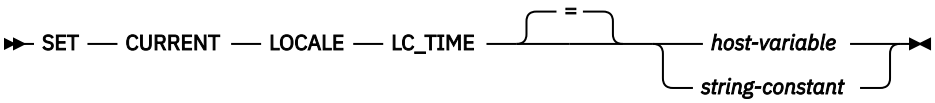
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

The CURRENT LOCALE LC_TIME special register is used by the DAYNAME, MONTHNAME, NEXT_DAY, ROUND, ROUND_TIMESTAMP, TIMESTAMP_FORMAT, TRUNCATE, TRUNC_TIMESTAMP and VARCHAR_FORMAT functions when the *locale-name* argument is not explicitly specified.

host-variable
A variable of type CHAR or VARCHAR. It cannot be set to null.

string-constant
A character string constant.

Notes

- ***Initial Value:*** The initial value of the CURRENT LOCALE LC_TIME special register is 'en_US'.
- ***Potential future use:*** In a future release the value of the CURRENT LOCALE LC_TIME special register might be used by other scalar functions and for other areas of the database environment that involve datetime values.
- ***Valid locales and naming:*** For information on valid locales and their naming,, see "Locale names for SQL and XQuery" in the *Globalization Guide* .

Examples

- *Example 1:* The following statement sets the CURRENT LOCALE LC_TIME special register to the English (Canada) locale using the latest version of Common Locale Data Repository (CLDR) available in the database manager.

```
SET CURRENT LOCALE LC_TIME = 'en_CA'
```

- *Example 2:* The following statement sets the CURRENT LOCALE LC_TIME special register to the French (France) locale using Common Locale Data Repository (CLDR) version 1.8.1. The MONTHNAME scalar function is then invoked with a single argument of '2008-11-10-00.00.00.000000'.

```
SET CURRENT LOCALE LC_TIME = 'CLDR181_fr_FR'
VALUES MONTHNAME( '2008-11-10-00.00.00.000000' )
```

  returns:

```
'novembre'
```

SET CURRENT LOCK TIMEOUT

The SET CURRENT LOCK TIMEOUT statement changes the value of the CURRENT LOCK TIMEOUT special register.

This statement is not under transaction control.

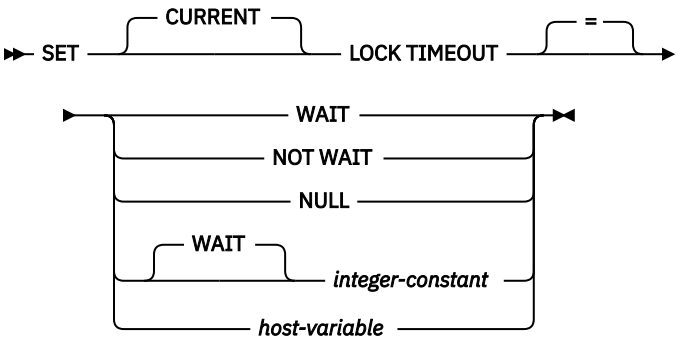
Invocation

The statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

The specified value must be an integer between -1 and 32767, inclusive (SQLSTATE 428B7), or the null value.

WAIT
Specifies a CURRENT LOCK TIMEOUT value of -1, which means that the database manager is to wait until a lock is released, or a deadlock is detected (SQLSTATE 40001 or 57033).

NOT WAIT
Specifies a CURRENT LOCK TIMEOUT value of 0, which means that the database manager is not to wait for locks that cannot be obtained, and an error (SQLSTATE 40001 or 57033) will be returned.

NULL
Specifies that the CURRENT LOCK TIMEOUT value is to be unset, and that the value of the **locktimeout** database configuration parameter is to be used when waiting for a lock. The value that is returned for the special register will change as the value of **locktimeout** changes.

WAIT *integer-constant*
Specifies an integer value between -1 and 32767. A value of -1 is equivalent to specifying the WAIT keyword without an integer value. A value of 0 is equivalent to specifying the NOT WAIT clause. If the value is between 1 and 32767, the database manager will wait that number of seconds (if a lock cannot be obtained) before an error (SQLSTATE 40001 or 57033) is returned.

host-variable
A variable of type INTEGER. The value must be between -1 and 32767. If *host-variable* has an associated indicator variable, and the value of that indicator variable specifies a null value, the CURRENT LOCK TIMEOUT value is unset. This is equivalent to specifying the NULL keyword.

Notes

- An updated value of the special register takes effect immediately upon successful execution of this statement. Because the special register value that is to be used during statement execution is fixed at the beginning of statement execution, an updated value of the CURRENT LOCK TIMEOUT special register will only be returned by statements that start execution after the SET LOCK TIMEOUT statement has completed successfully.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with Informix database products. These alternatives are non-standard and should not be used.
  - MODE can be specified in place of TIMEOUT.
  - TO can be specified in place of the equals (=) operator.

Examples

- *Example 1:* Set the lock timeout value to wait for 30 seconds before returning an error.

```
SET CURRENT LOCK TIMEOUT 30
```

- *Example 2:* Unset the lock timeout value, so that the **locktimeout** database configuration parameter value will be used instead.

```
SET CURRENT LOCK TIMEOUT NULL
```

SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION

The SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION statement changes the value of the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register.

This statement is not under transaction control.

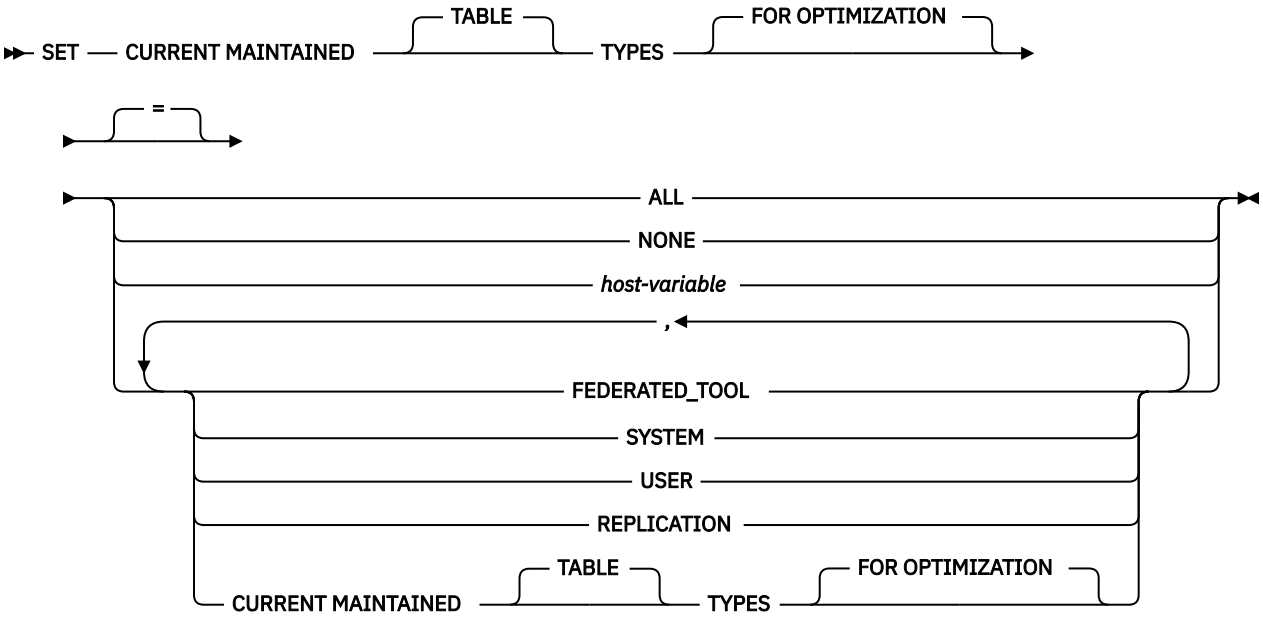
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

ALL
Specifies that all possible types of maintained tables controlled by this special register, now and in the future, are to be considered when optimizing the processing of dynamic SQL queries.

NONE
Specifies that none of the object types that are controlled by this special register are to be considered when optimizing the processing of dynamic SQL queries.

FEDERATED_TOOL
Specifies that refresh-deferred materialized query tables that are maintained by a federated tool can be considered to optimize the processing of dynamic SQL queries, provided the value of the CURRENT QUERY OPTIMIZATION special register is 2 or greater than 5.

SYSTEM
Specifies that system-maintained refresh-deferred materialized query tables can be considered to optimize the processing of dynamic SQL queries. (Immediate materialized query tables are always available.)

USER
Specifies that user-maintained refresh-deferred materialized query tables can be considered to optimize the processing of dynamic SQL queries.

REPLICATION
Specifies that shadow tables can be considered to help optimize the processing of dynamic SQL queries.

CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
The value of the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register before this statement executes.

host-variable
A variable of type CHAR or VARCHAR. The length of the contents of the host variable must not exceed 254 bytes (SQLSTATE 42815). It cannot be set to null. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

The characters of *host-variable* must be left-aligned. The contents of *host-variable* must be a string that is a comma-separated list of keywords matching what can be specified as keywords for the special register. These keywords must be specified in the exact case intended, because there is no conversion to uppercase characters. The value must be padded on the right with blanks if its length is less than that of the host variable.

Notes

- The initial value of the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register is determined by the **dft_mttb_types** database configuration parameter, which has a default value of SYSTEM.
- You must set the CURRENT REFRESH AGE special register to a value other than 0 for the specified table types to be considered during optimization of the processing of dynamic SQL queries.
- If the value of the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register includes anything other than REPLICATION or NONE, the value of the CURRENT REFRESH AGE special register must be either 0 or 99999999999999 (ANY).

Examples

- *Example 1:* Set the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register.

```
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION = SYSTEM, USER
```

- *Example 2:* Retrieve the current value of the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register into a host variable called CURMAINTYPES.

```
EXEC SQL VALUES (CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION)
  INTO :CURMAINTYPES
```

- *Example 3:* Set the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register to have no value.

```
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION = NONE
```

SET CURRENT MDC ROLLOUT MODE

The SET CURRENT MDC ROLLOUT MODE statement assigns a value to the CURRENT MDC ROLLOUT MODE special register. The value specifies the type of rollout cleanup that is to be performed on qualifying DELETE statements for multidimensional clustering (MDC) tables.

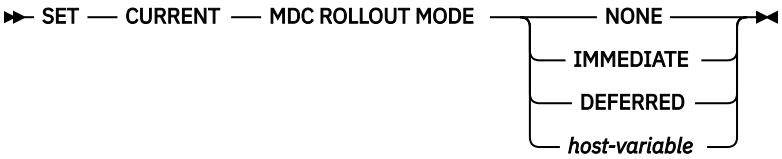
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

NONE
Specifies that MDC rollout optimization during delete operations is not to be used. The DELETE statement is processed in the same way as a DELETE statement that does not qualify for rollout.

IMMEDIATE
Specifies that MDC rollout optimization is to be used if the DELETE statement qualifies. If the table has RID indexes, the indexes are updated immediately during delete processing. The deleted blocks are available for reuse after the transaction commits.

DEFERRED
Specifies that MDC rollout optimization is to be used if the DELETE statement qualifies. If the table has RID indexes, index updates are deferred until after the transactions commits. With this option, delete processing is faster and uses less log space, but the deleted blocks are not available for reuse until after the index updates are complete.

host-variable
A variable of type VARCHAR. The length of *host-variable* must be less than or equal to 17 bytes (SQLSTATE 42815). The value of the host variable must be a left-aligned string that is one of 'NONE', 'IMMEDIATE', or 'DEFERRED' (case insensitive). If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

Notes

- Subsequent DELETE statements that are eligible for rollout processing respect the setting of the CURRENT MDC ROLLOUT MODE special register. Currently executing sections are not affected by a change to this special register.
- The effects of executing the SET CURRENT MDC ROLLOUT MODE statement are not rolled back if the unit of work in which the statement is executed is rolled back.
- After you run a SET CURRENT MDC ROLLOUT MODE statement, the behavior of MDC table rollout changes. The behavior of MDC table rollouts returns to the configuration set by the registry variable **DB2_MDC_ROLLOUT** when one of the following situations occur:
  - The connection/session to the database is terminated.
  - A CONNECT RESET is entered.
  - A SET CURRENT MDC ROLLOUT MODE NONE is entered.
- The DEFERRED mode is not supported on a data partitioned MDC table with partitioned RID indexes. Only the NONE and IMMEDIATE modes are supported. The cleanup rollout type will be IMMEDIATE if the **DB2_MDC_ROLLOUT** registry variable is set to DEFER, or if the CURRENT MDC ROLLOUT MODE special register is set to DEFERRED to override the **DB2_MDC_ROLLOUT** setting.

  If only nonpartitioned RID indexes exist on the MDC table, deferred index cleanup rollout is supported.

Example

Specify deferred cleanup behavior for the next DELETE statement that qualifies for rollout processing.

```
SET CURRENT MDC ROLLOUT MODE IMMEDIATE
```

SET CURRENT OPTIMIZATION PROFILE

The SET CURRENT OPTIMIZATION PROFILE statement assigns a value to the CURRENT OPTIMIZATION PROFILE special register. The value specifies the optimization profile the optimizer should use when preparing dynamic DML statements.

This statement is not under transaction control.

When the statement is evaluated, the name of the optimization profile is checked for validity, but the profile is not processed until the optimizer encounters a dynamic DML statement.

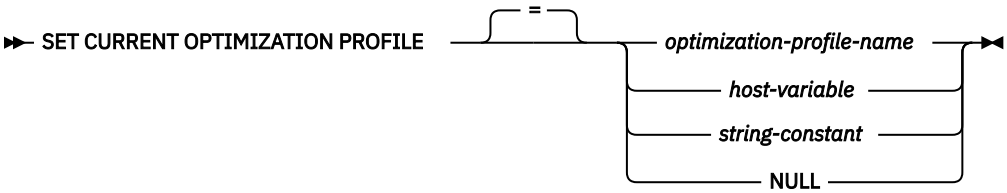
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

optimization-profile-name

The two-part name of the optimization profile. The name can be specified with a literal, host variable, or special register. The name specified is the name entered into the CURRENT OPTIMIZATION PROFILE special register.

If the specified optimization-profile-name is unqualified, the value of the CURRENT DEFAULT SCHEMA register is used as the implicit qualifier. The default value of the special register is null.

host-variable

A variable of type CHAR or VARCHAR that includes the name of the optimization profile. A host variable that includes a null indicator indicates that the value of the OPTPROFILE bind option is to be used if that value is specified for the current package. A host variable of zero length, or of white space only, indicates that no optimization profile is to be used.

The host variable must meet the following characteristics:

- The content of the string is a single or two-part identifier (separated by a period), with no leading blanks.
- The identifier or identifiers can be delimited or non-delimited.
- The content of the string is not folded to upper case.
- Lower case and special characters cannot be used in non-delimited strings.
- If the first character is a double quotation mark, a closing double quotation mark must either precede a period or be the last non-blank character in the string.
- If the first character following a period is a double quotation mark, then a double quotation mark must be the last non-blank character in the string.
- If the identifier is delimited, then to include double quotation marks in the identifier, specify the character twice.

- Any period that is not inside a delimited identifier is treated as a separator, and only one period separator can exist in the string.

string-constant
Specifies a constant as a character string that is the name of the optimization profile. The content of a string constant must meet the same characteristics as a host variable.

NULL
Sets the CURRENT OPTIMIZATION PROFILE register to null.

Table 153 on page 1835 provides examples of string literals and identifiers that might be used to assign the register as per the optimization profile naming rules. The value in the SCHEMA and NAME column represent an optimization profile name as it might appear in the OPT_PROFILE table. The valid string literals column shows string literals that match the optimization profile named by the corresponding SCHEMA and NAME column values. The valid identifiers column shows identifiers that would identify that same optimization profile.

Table 153. Examples of string literals and identifiers

SCHEMA	NAME	Valid string literals	Valid identifiers
SIMMEN	BIG_PROF	'BIG_PROF' 'SIMMEN.BIG_PROF' '"BIG_PROF"' '"SIMMEN"."BIG_PROF"'	BIG_PROF SIMMEN.BIG_PROF "BIG_PROF" "SIMMEN"."BIG_PROF"
SIMMEN	low_profile	'"low_profile"' 'SIMMEN."low_profile"' '"SIMMEN"."low_profile"'	"low_profile" SIMMEN."low_profile" "SIMMEN"."low_profile"
eliaz	DBA3	'DBA3' '"DBA3"' '"eliaz".DBA3' '"eliaz"."DBA3"'	DBA3 "eliaz".DBA3 "eliaz"."DBA3"
SNOW	PROFILE1.0	'"PROFILE1.0"' 'SNOW."PROFILE1.0"' '"SNOW"."PROFILE1.0"'	"PROFILE1.0" SNOW."PROFILE1.0" "SNOW"."PROFILE1.0"

Notes

- If the value of the register specifies the name of an existing optimization profile, the specified optimization profile is used when preparing subsequent dynamic DML statements.
- If the value of the register is null, the optimization profile specified by the OPTPROFILE bind option, if any, is used when preparing subsequent dynamic DML statements.
- If the value of the register is null, and the OPTPROFILE bind option is not set, no optimization profile is used when preparing subsequent dynamic DML statements.
- If the value of the register is the empty string, then no optimization profile is used when preparing subsequent dynamic DML statements, regardless of whether the OPTPROFILE bind option is set.
- Subsequent changes to CURRENT DEFAULT SCHEMA do not have any effect on the optimization profile. The CURRENT OPTIMIZATION PROFILE register value is set with the two part name that is in effect at the time SET CURRENT OPTIMIZATION PROFILE statement is evaluated. Only another SET CURRENT OPTIMIZATION PROFILE statement can change the optimization profile that is used.

Examples

- *Example 1:* The optimization profile RICK.FOO is used for statements 1, 2, and 3. TOM.FOO is used for statement 4.

```
SET CURRENT SCHEMA = 'RICK'
SET CURRENT OPTIMIZATION PROFILE = 'FOO'
   statement 1
   statement 2
SET CURRENT SCHEMA = 'TOM'
   statement 3
SET CURRENT OPTIMIZATION PROFILE = 'FOO'
   statement 4
```

- *Example 2:* An application with the following statements was bound with the options OPTPROFILE("Foo") and QUALIFIER("John"). The optimization profile KAAREL.BAR is used for statement 1 and optimization profile "John"."Foo" is used for statement 2.

```
SET CURRENT SCHEMA = 'KAAREL'
SET CURRENT OPTIMIZATION PROFILE = 'BAR'
   statement 1
SET CURRENT SCHEMA = "Tom"
SET CURRENT OPTIMIZATION PROFILE NULL
   statement 2
```

- *Example 3:* The empty string is a special value that indicates that no optimization profile is to be used. Optimization profile "Hamid"."Foo" is used for statement 1 and no optimization profile is used for statement 2.

```
SET CURRENT OPTIMIZATION PROFILE = '"Hamid"."Foo"'
   statement 1
SET CURRENT OPTIMIZATION PROFILE = ''
   statement 2
```

SET CURRENT PACKAGE PATH

The SET CURRENT PACKAGE PATH statement assigns a value to the CURRENT PACKAGE PATH special register.

This statement is not under transaction control.

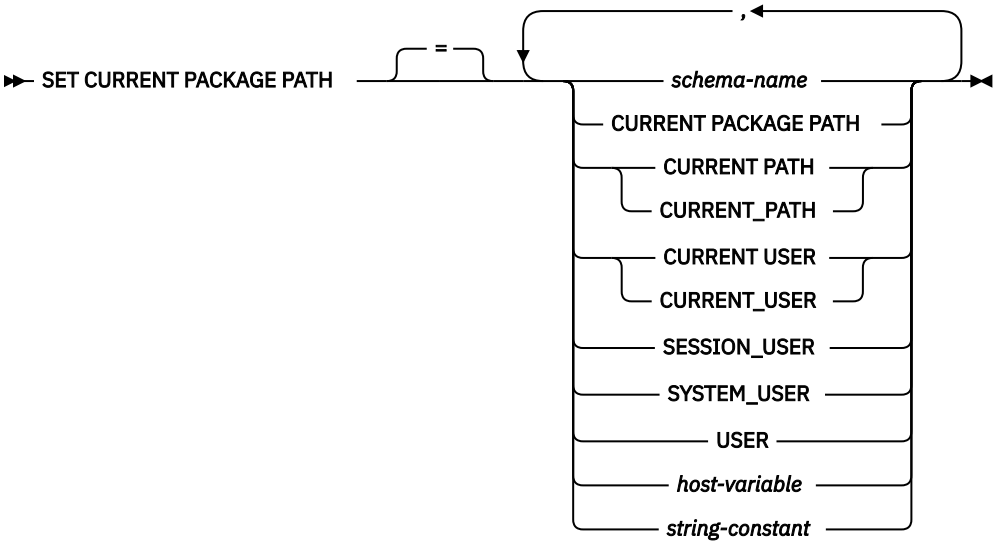
Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax



Description

schema-name
Identifies a schema. The name must not be a delimited identifier that is empty or that contains only blanks (SQLSTATE 42815).

CURRENT PACKAGE PATH
The value of the CURRENT PACKAGE PATH special register before this statement executes.

CURRENT PATH
The value of the CURRENT PATH special register.

CURRENT USER
The value of the CURRENT USER special register.

SESSION_USER
The value of the SESSION_USER special register.

SYSTEM_USER
The value of the SYSTEM_USER special register.

USER
The value of the USER special register.

host-variable
Contains one or more schema names, separated by commas. The host variable must:

- Be a character-string variable (CHAR or VARCHAR). The actual length of the contents of the host variable must not exceed the length of the CURRENT PACKAGE PATH special register.
- Not be the null value. If an indicator variable is provided, its value must not indicate a null value.
- Contain an empty or blank string, or one or more schema names separated by commas.
- Be padded on the right with blanks if the actual length of the host variable is greater than the content.
- Not contain CURRENT PACKAGE PATH, CURRENT PATH, CURRENT_PATH, CURRENT USER, CURRENT_USER, SESSION_USER, SYSTEM_USER, PATH, or USER.
- Not contain a delimited identifier that is empty or that contains only blanks.

string-constant
Specifies a character string constant that contains zero, one, or more schema names that are separated by commas. The string constant must:

- Have a length that does not exceed the maximum length of the CURRENT PACKAGE PATH special register.
- Not contain CURRENT PACKAGE PATH, CURRENT PATH, CURRENT_PATH, CURRENT USER, CURRENT_USER, SESSION_USER, SYSTEM_USER, PATH, or USER.
- Not contain a delimited identifier that is empty or that contains only blanks.

Rules

- If the same schema appears more than once in the list, the first occurrence of the schema is used (SQLSTATE 01625).
- The number of schemas that can be specified is limited by the total length of the CURRENT PACKAGE PATH special register. The special register string is built by taking each specified schema name and removing trailing blanks, delimiting the name with double quotation marks, and separating the schema names with commas. The length of the resulting list cannot exceed the maximum length of the special register (SQLSTATE 0E000).
- A schema name that does not conform to the rules for an ordinary identifier (for example, a schema name that contains lowercase characters or characters that cannot be specified in an ordinary identifier), must be specified as a delimited schema name, and must not be specified within a host variable or string constant.
- To indicate that the current value of a special register (specified as a single keyword) is to be used in the package path, specify the name of the special register as a keyword. If the name of the special register is specified as a delimited identifier instead (for example, "USER"), it is interpreted as a schema name of that value ('USER').
- The following rules are used to determine whether a value specified in a SET CURRENT PACKAGE PATH statement is a variable or a schema name:
  - If *name* is the same as a parameter or SQL variable in the SQL procedure, *name* is interpreted as a parameter or SQL variable, and the value in *name* is assigned to the package path.
  - If *name* is not the same as a parameter or SQL variable in the SQL procedure, *name* is interpreted as a schema name, and the value in *name* is assigned to the package path.

Notes

- ***Transaction considerations:*** The SET CURRENT PACKAGE PATH statement is not a commitable operation. ROLLBACK has no effect on the CURRENT PACKAGE PATH special register.
- ***Existence checking of schemas:*** No validation that the specified schemas exist is made at the time that the CURRENT PACKAGE PATH special register is set. For example, a schema that is misspelled is not detected, which could affect the way subsequent SQL operates. At package execution time, authorization to a matching package is checked, and if this authorization check fails, an error is returned (SQLSTATE 42501).
- ***Contents of host variable or string constant:*** The contents of a host variable or a string constant are interpreted as a list of schema names. If multiple schema names are specified, they must be separated by commas. Each schema name in the list must conform to the rules for forming an ordinary identifier, or be specified as a delimited identifier. The contents of the host variable or string constant are not folded to uppercase.
- ***Restrictions specific to embedded SQL for COBOL applications:*** A maximum of ten literal (non-host variable) values can appear on the right side of a SET CURRENT PACKAGE PATH statement. Such values can have a maximum length of 130 (non-delimited) or 128 (delimited).

Examples

- *Example 1:* Set the CURRENT PACKAGE PATH special register to the following list of schemas: MYPKGS, 'ABC E', SYSIBM

```
SET CURRENT PACKAGE PATH = MYPKGS, 'ABC E', SYSIBM
```

The following statement sets a host variable to the value of the resulting list:

```
SET :hvpklist = CURRENT PACKAGE PATH
```

The value of the host variable is: "MYPKGS", "ABC E", "SYSIBM".

- *Example 2:* Set the CURRENT PACKAGE PATH special register to the following list of schemas: "SCH4","SCH5", where :hvar1 contains 'SCH4,SCH5'.

```
SET CURRENT PACKAGE PATH :hvar1
```

The value of the CURRENT PACKAGE PATH special register after this statement executes is: "SCH4","SCH5".

- *Example 3:* Set the CURRENT PACKAGE PATH special register to the following list of schemas: "SCH1","SCH#2","SCH3","SCH4","SCH5", where :hvar1 contains 'SCH4,SCH5'.

```
SET CURRENT PACKAGE PATH = SCH1,'SCH#2',"SCH3",:hvar1
```

The value of the CURRENT PACKAGE PATH special register after this statement executes is: "SCH1","SCH#2","SCH3","SCH4","SCH5".

- *Example 4:* Clear the CURRENT PACKAGE PATH special register.

```
SET CURRENT PACKAGE PATH = ''
```

- *Example 5:* Temporarily append the "SCH_PROD" schema (contained in the :prodschema host variable) and the "SCH_PROD2" schema (contained in the :prod2schema host variable) to the end of the CURRENT PACKAGE PATH special register for execution of the SUMMARIZE procedure. Then, switch the CURRENT PACKAGE PATH special register back to its previous value.

```
SET :oldCPP = CURRENT PACKAGE PATH

SET CURRENT PACKAGE PATH = CURRENT PACKAGE PATH,:prodschema,:prod2schema

CALL SUMMARIZE(:V1,:V2)

SET CURRENT PACKAGE PATH = :oldCPP
```

- *Example 6:* Set the CURRENT PACKAGE PATH special register to a list of delimited schema names: "MY.SCHEMA" (imbedded period), "OLD SCHEMA" (imbedded blank). Use a single host variable containing both delimited identifiers:

```
hv = '"MY.SCHEMA", "OLD SCHEMA"'

SET CURRENT PACKAGE PATH = :hv
```

or use a single string constant containing both delimited identifiers:

```
SET CURRENT PACKAGE PATH = '"MY.SCHEMA", "OLD SCHEMA"'
```

or use a list of delimited schemas:

```
SET CURRENT PACKAGE PATH = 'MY.SCHEMA', 'OLD SCHEMA'
```

SET CURRENT PACKAGESET

The SET CURRENT PACKAGESET statement sets the schema name (collection identifier) that will be used to select the package to use for subsequent SQL statements.

This statement is not under transaction control.

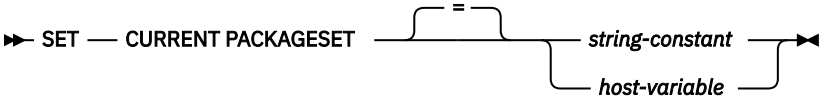
Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared. This statement is not supported in REXX.

Authorization

None required.

Syntax



Description

string-constant
A character string constant. If the value exceeds 128 bytes, only the first 128 bytes are used.

host-variable
A variable of type CHAR or VARCHAR. It cannot be set to null. If the value exceeds 128 bytes, only the first 128 bytes are used.

Notes

- This statement allows an application to specify the schema name used when selecting a package for an executable SQL statement. The statement is processed at the client and does not flow to the application server.
- The COLLECTION bind option can be used to create a package with a specified schema name.
- Unlike Db2 for z/OS, the SET CURRENT PACKAGESET statement is implemented without support for a special register called CURRENT PACKAGESET.

Examples

- *Example 1*: Assume an application called TRYIT is precompiled by user ID PRODUSA, making "PRODUSA" the default schema name in the bind file. The application is then bound twice with different bind options. The following command line processor commands were used:

```
CONNECT TO SAMPLE USER PRODUSA
BIND TRYIT.BND DATETIME USA
CONNECT TO SAMPLE USER PRODEUR
BIND TRYIT.BND DATETIME EUR COLLECTION 'PRODEUR'
```

  This creates two packages called TRYIT. The first bind command created the package in the schema named "PRODUSA". The second bind command created the package in the schema named "PRODEUR" based on the COLLECTION option.

- *Example 2*: Assume the application TRYIT contains the following statements:

```
EXEC SQL CONNECT TO SAMPLE;
.
.
EXEC SQL SELECT HIREDATE INTO :HD FROM EMPLOYEE WHERE EMPNO='000010'; 1
.
.
EXEC SQL SET CURRENT PACKAGESET 'PRODEUR';                            2
.
.
EXEC SQL SELECT HIREDATE INTO :HD FROM EMPLOYEE WHERE EMPNO='000010'; 3
```

1
This statement will run using the PRODUSA.TRYIT package because it is the default package for the application. The date is therefore returned in USA format.

2
This statement sets the schema name to "PRODEUR" for package selection.

3

This statement will run using the PRODEUR.TRYIT package as a result of the SET CURRENT PACKAGESET statement. The date is therefore returned in EUR format.

SET CURRENT QUERY OPTIMIZATION

The SET CURRENT QUERY OPTIMIZATION statement assigns a value to the CURRENT QUERY OPTIMIZATION special register. The value specifies the current class of optimization techniques enabled when preparing dynamic SQL statements.

This statement is not under transaction control.

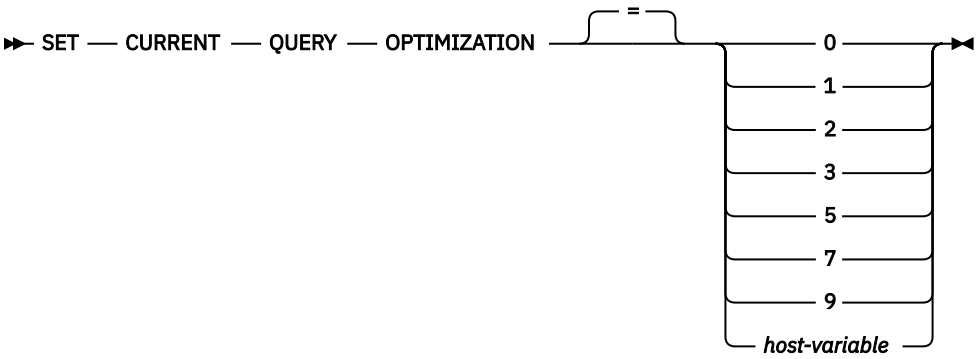
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

optimization-class

optimization-class can be specified either as an integer constant or as the name of a host variable that will contain the appropriate value at run time. An overview of the classes follows.

0

Specifies that a minimal amount of optimization is performed to generate an access plan. This class is most suitable for simple dynamic SQL access to well-indexed tables.

1

Specifies that optimization roughly comparable to Db2 Version 1 is performed to generate an access plan.

2

Specifies a level of optimization higher than that of Db2 Version 1, but at significantly less optimization cost than levels 3 and higher, especially for very complex queries.

3

Specifies that a moderate amount of optimization is performed to generate an access plan.

5

Specifies a significant amount of optimization is performed to generate an access plan. For complex dynamic SQL queries, heuristic rules are used to limit the amount of time spent selecting an access plan. Where possible, queries will use materialized query tables instead of the underlying base tables.

7
Specifies a significant amount of optimization is performed to generate an access plan. Similar to 5 but without the heuristic rules.

9
Specifies a maximal amount of optimization is performed to generate an access plan. This can greatly expand the number of possible access plans that are evaluated. This class should be used to determine if a better access plan can be generated for very complex and very long-running queries using large tables. Explain and performance measurements can be used to verify that a better plan has been generated.

host-variable
The data type is INTEGER. The value must be in the range 0 to 9 (SQLSTATE 42815) but should be 0, 1, 2, 3, 5, 7, or 9. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

Notes

- When the CURRENT QUERY OPTIMIZATION register is set to a particular value, a set of query rewrite rules are enabled, and certain optimization variables take on particular values. This class of optimization techniques is then used during preparation of dynamic SQL statements.
- In general, changing the optimization class impacts the execution time of the application, the compilation time, and resources required. Most statements will be adequately optimized using the default query optimization class. Lower query optimization classes, especially classes 1 and 2, may be appropriate for dynamic SQL statements for which the resources consumed by the dynamic *PREPARE* are a significant portion of those required to execute the query. Higher optimization classes should be chosen only after considering the additional resources that may be consumed and verifying that a better access plan has been generated.
- Query optimization classes must be in the range 0 to 9. Classes outside this range will return an error (SQLSTATE 42815). Unsupported classes within this range will return a warning (SQLSTATE 01608) and will be replaced with the next lowest query optimization class. For example, a query optimization class of 6 will be replaced by 5.
- Dynamically prepared statements use the class of optimization that was set by the most recently executed SET CURRENT QUERY OPTIMIZATION statement. In cases where a SET CURRENT QUERY OPTIMIZATION statement has not yet been executed, the query optimization class is determined by the value of the **dft_queryopt** database configuration parameter.
- Statically bound statements do not use the CURRENT QUERY OPTIMIZATION special register; therefore this statement has no effect on them. The QUERYOPT option is used during preprocessing or binding to specify the required class of optimization for statically bound statements. If QUERYOPT is not specified then, the default value specified by the **dft_queryopt** database configuration parameter is used.
- The results of executing the SET CURRENT QUERY OPTIMIZATION statement are not rolled back if the unit of work in which it is executed is rolled back.

Examples

- *Example 1:* This example shows how the highest degree of optimization can be selected.

```
SET CURRENT QUERY OPTIMIZATION 9
```

- *Example 2:* The following example shows how the CURRENT QUERY OPTIMIZATION special register can be used within a query.

  Using the SYSCAT.PACKAGES catalog view, find all plans that were bound with the same setting as the current value of the CURRENT QUERY OPTIMIZATION special register.

```
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT PKGNAME, PKGSCHEMA FROM SYSCAT.PACKAGES
  WHERE QUERYOPT = CURRENT QUERY OPTIMIZATION
```

SET CURRENT REFRESH AGE

The SET CURRENT REFRESH AGE statement changes the value of the CURRENT REFRESH AGE special register.

This statement is not under transaction control.

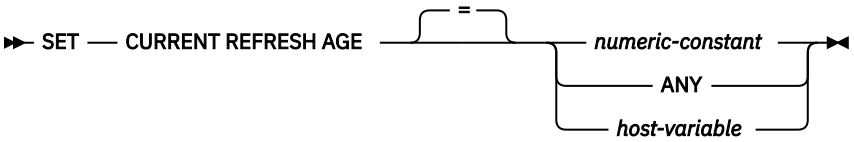
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

numeric-constant
A DECIMAL(20,6) value that represents a time stamp duration.The value must be 0 - 99999999999999 or a valid time stamp within that range. The valid format for the range is *yyyymmddhhmmss.nnnnnn*, where:

- *yyyy* is the number of years and can have a value of 0 - 9999.
- *mm* is the number of months and can have a value of 0 - 11.
- *dd* is the number of days and can have a value of 0 - 30.
- *hh* is the number of hours and can have a value of 0 - 23.
- *mm* is the number of minutes and can have a value of 0 - 59.
- *ss* is the number of seconds and can have a value of 0 - 59.
- *nnnnnn* is the number of fractional seconds. The fractional seconds portion of the value is ignored and therefore can be any value.

You do not have to include the leading zeros for the entire value or the trailing fractional seconds. However, individual elements that have another element to the left must include the zeros. For example, to represent 1 hour, 7 minutes, and 5 seconds, use 10705.

If materialized query tables that are affected by the CURRENT REFRESH AGE special register are maintained by USER, SYSTEM, or FEDERATED_TOOL, the only valid numeric values are 0 and 99999999999999. For further details, see the "Notes" section.

ANY
A short form 99999999999999. See the description of the *numeric-constant* parameter.

host-variable
A variable of type DECIMAL(20,6) or another type that is assignable to DECIMAL(20,6). You cannot set the *host-variable* parameter to null. If the host variable has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815). The *host-variable* parameter must conform to the same constraints as the *numeric-constant* parameter.

Notes

- The initial value of the CURRENT REFRESH AGE special register is determined by the **dft_refresh_age** database configuration parameter, which has a default value of 0.
- Be careful when setting the CURRENT REFRESH AGE special register to a value other than 0. A table type that you specify for the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register might not represent the values of the underlying base table. If such a table is used to optimize the processing of a query, the query result might not accurately represent the data in the underlying table. This might be reasonable if you know that the underlying data has not changed, or if you are willing to accept a degree of error in the results, based on your knowledge of the cached data.
- The CURRENT REFRESH AGE special register value of 99999999999999 cannot be used in time stamp arithmetic operations, because the result would be outside the valid range for dates (SQLSTATE 22008).
- The CURRENT REFRESH AGE special register affects the materialized query tables that you define as REFRESH DEFERRED MAINTAINED BY USER, REFRESH DEFERRED MAINTAINED BY REPLICATION, and REFRESH DEFERRED MAINTAINED BY SYSTEM. The CURRENT REFRESH AGE special register affects these materialized query tables as follows:
  - If the value of the CURRENT REFRESH AGE special register is 0, the materialized query tables are not used to optimize the processing of a query.
  - If the value of the CURRENT REFRESH AGE special register is 99999999999999, the materialized query tables can be used to help optimize the processing of a query if both the following criteria are met:
    - The value of the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register includes these tables.
    - The value of the CURRENT QUERY OPTIMIZATION special register is 2 or a value that is greater than or equal to 5.
  - If the value of the CURRENT REFRESH AGE special register is a value other than 0 or 99999999999999, only shadow tables are affected by this special register setting can be used to optimize the processing of a query, but only if both the following criteria are met:
    - The value of the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register is REPLICATION.
    - The value of the CURRENT QUERY OPTIMIZATION special register is 2 or a value that is greater than or equal to 5.
- The CURRENT REFRESH AGE special register has no effect on whether REFRESH IMMEDIATE MAINTAINED BY SYSTEM materialized query tables or REFRESH DEFERRED MAINTAINED BY FEDERATED_TOOL materialized query tables are used to optimize the processing of a query.

  REFRESH IMMEDIATE MAINTAINED BY SYSTEM materialized query tables can always be used to optimize the processing of a query if the value of the CURRENT QUERY OPTIMIZATION special register is 2 or a value that is greater than or equal to 5.

  REFRESH DEFERRED MAINTAINED BY FEDERATED_TOOL materialized query tables are used for optimization if both the following criteria are met:
  - The value of the CURRENT QUERY OPTIMIZATION special register is 2 or a value that is greater than or equal to 5.
  - The value of the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register is ALL or includes FEDERATED_TOOL.

Examples

- *Example 1:* The following statement sets the CURRENT REFRESH AGE special register:

```
SET CURRENT REFRESH AGE ANY
```

- *Example 2:* The following command retrieves the value of the CURRENT REFRESH AGE special register into a host variable called CURMAXAGE.

```
EXEC SQL VALUES (CURRENT REFRESH AGE) INTO  :CURMAXAGE;
```

SET CURRENT SQL_CCFLAGS

The SET CURRENT SQL_CCFLAGS statement changes the value of the CURRENT SQL_CCFLAGS special register.

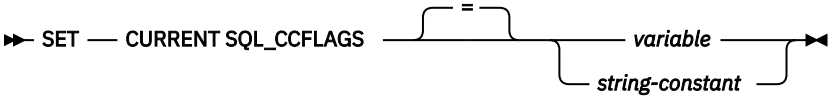
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

variable
Specifies a variable that contains one or more name and value pairs that are separated by commas.

The variable must have the following characteristics (SQLSTATE 42815):

- The data type must be CHAR or VARCHAR. The actual length of the contents of the variable must not exceed the maximum length of the special register.
- It must be a string of blanks, an empty string, or include one or more name and value pairs where the name is separated from the value by the colon character. The name must be a valid ordinary identifier. The value associated with a name must be a BOOLEAN constant, an INTEGER constant, or the keyword NULL.
- It must be padded on the right with blanks if using a fixed-length character variable.
- It can include extra blanks at the beginning or ending of the string, around the comma character, or around the colon character. The blanks are ignored.
- It must not be the null value.

string-constant
Specifies a character string constant that contains one or more name and value pairs that are separated by commas.

The string constant must have the following characteristics (SQLSTATE 42815):

- It must be a character string constant. The length of the constant must not exceed the maximum length of the special register.
- It must be a string of blanks, an empty string or include one or more name and value pairs where the name is separated from the value by the colon character. The name must be a valid ordinary identifier. The value associated with a name must be a BOOLEAN constant, an INTEGER constant, or the keyword NULL.
- It can include extra blanks at the beginning or ending of the string, around the comma character, or around the colon character. The blanks are ignored.

Notes

- If a duplicate name appears in the content for the CURRENT SQL_FLAGS special register, then only the last (furthest to the right) value is used. The special register value will include only a single occurrence of the duplicated name with the value that is used. Concatenating a duplicated name with a different value to the CURRENT SQL_CCFLAGS value can be used to override some conditional compilation values while retaining other values.
- When the CURRENT SQL_CCFLAGS is retrieved, the returned string includes the unique name and value pairs in uppercase characters with multiple pairs separated by a comma and a blank. The pairs are in the order they were specified, with a duplicate name appearing only where it first occurred, but reflecting the value from where it last occurred.
- The CURRENT SQL_CCFLAGS special register can be set to the default defined for the database by retrieving the VALUE column from SYSIBMADM.DBCFG where NAME='sql_ccflags' into a variable and then assigning that variable to the special register.
- ***Transaction considerations:*** The SET SQL_CCFLAGS statement is not a committable operation. ROLLBACK has no effect on CURRENT SQL_CCFLAGS.

Examples

- *Example 1*: Define a conditional compilation value for the session to indicate that the server is Db2 9.7 and that debug is false.

```
SET CURRENT SQL_CCFLAGS 'db2v97:true, debug:false'
```

- *Example 2*: Extend the existing CURRENT SQL_CCFLAGS to set debug to true and define the tracing level.

```
BEGIN
  DECLARE LIST VARCHAR(1024);
  SET LIST = CASE WHEN (CURRENT SQL_CCFLAGS = ' ')
              THEN 'tracelvl:3,debug:true'
              ELSE CURRENT SQL_CCFLAGS
                   concat ',tracelvl:3,debug:true'
              END;
  SET CURRENT SQL_CCFLAGS = LIST;
END
```

A CASE expression is used in the assignment to handle the possibility that the CURRENT SQL_CCFLAGS special register does not include any conditional compilation values, resulting in a leading comma in the value of the variable LIST.

A query of the CURRENT SQL_CCFLAGS special register after the execution of the statement in Example 1 and the compound statement in this example would return:

```
DB2V97:TRUE, DEBUG:TRUE, TRACELVL:3
```

Even though the conditional compilation value for DEBUG appeared twice in the variable LIST, it appears only once in the special register value where it would have first appeared.

SET CURRENT TEMPORAL BUSINESS_TIME

The SET CURRENT TEMPORAL BUSINESS_TIME statement changes the value of the CURRENT TEMPORAL BUSINESS_TIME special register.

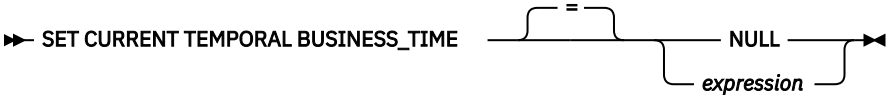
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

NULL
Specifies the null value.

expression
Each expression can contain any of the following supported operands (SQLSTATE 428HY):

- Constant
- Special register
- Variable. For details, refer to "References to variables" in the "Identifiers" topic, in *SQL Reference Volume 1* .
- Built-in scalar function whose arguments are supported operands. User-defined functions and non-deterministic functions are not supported in this context.
- CAST specification where the cast operand is a supported operand
- Expression using arithmetic operator and operands

Notes

- ***Transaction considerations:*** The SET CURRENT TEMPORAL BUSINESS_TIME statement is not a committable operation. ROLLBACK has no effect on CURRENT TEMPORAL BUSINESS_TIME.
- ***Effects on other special registers:*** The setting of the CURRENT TEMPORAL BUSINESS_TIME special register does not have any effect on the values of other special registers, specifically the CURRENT DATE and CURRENT TIMESTAMP special registers.

Examples

- *Example 1:* Set the CURRENT TEMPORAL BUSINESS_TIME special register to the previous month.

```
SET CURRENT TEMPORAL BUSINESS_TIME = CURRENT TIMESTAMP - 1 MONTH
```

- *Example 2:* Set the CURRENT TEMPORAL BUSINESS_TIME special register to the null value.

```
SET CURRENT TEMPORAL BUSINESS_TIME = NULL
```

SET CURRENT TEMPORAL SYSTEM_TIME

The SET CURRENT TEMPORAL SYSTEM_TIME statement changes the value of the CURRENT TEMPORAL SYSTEM_TIME special register.

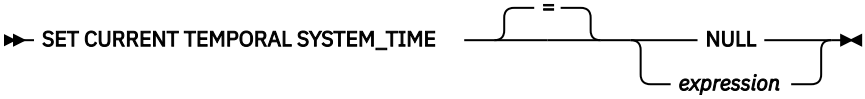
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

NULL
Specifies the null value.

expression
Each expression can contain any of the following supported operands (SQLSTATE 428HY):

- Constant
- Special register
- Variable (host variable, SQL variable, SQL parameter, transition variable, global variable)
- Built-in scalar function whose arguments are supported operands. User-defined functions and non-deterministic functions are not supported in this context.
- CAST specification where the cast operand is a supported operand
- Expression using arithmetic operator and operands

Notes

- ***Transaction considerations:*** The SET CURRENT TEMPORAL SYSTEM_TIME statement is not a committable operation. ROLLBACK has no effect on CURRENT TEMPORAL SYSTEM_TIME.
- ***Effects on other special registers:*** The setting of the CURRENT TEMPORAL SYSTEM_TIME special register does not have any effect on the values of other special registers, specifically the CURRENT DATE and CURRENT TIMESTAMP special registers.

Examples

- *Example 1:* Set the CURRENT TEMPORAL SYSTEM_TIME special register to the previous month.

```
SET CURRENT TEMPORAL SYSTEM_TIME = CURRENT TIMESTAMP - 1 MONTH
```

- *Example 2:* Set the CURRENT TEMPORAL SYSTEM_TIME special register to the null value.

```
SET CURRENT TEMPORAL SYSTEM_TIME = NULL
```

SET ENCRYPTION PASSWORD

The SET ENCRYPTION PASSWORD statement sets the password to be used by the ENCRYPT, DECRYPT_BIN and DECRYPT_CHAR functions. The password is not tied to database authentication, and is used for data encryption and decryption only.

This statement is not under transaction control.

Important: The SET ENCRYPTION PASSWORD statement is deprecated and might not appear in future releases.

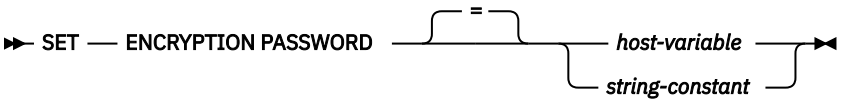
Invocation

The statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

The encryption password can be used by the ENCRYPT, DECRYPT_BIN, and DECRYPT_CHAR built-in functions for password-based encryption. The length of the password must be between 6 and 127 bytes and all characters must be specified in the exact case intended, because there is no automatic conversion to uppercase characters. To maintain the best level of security on your system, it is recommended that you use a host variable or dynamic parameter markers to specify the password, rather than using a literal string in your SET ENCRYPTION PASSWORD statement.

host-variable
A variable of type CHAR or VARCHAR. The length of the *host-variable* must be between 6 and 127 bytes (SQLSTATE 428FC). It cannot be set to null. All characters are specified in the exact case intended, as there is no conversion to uppercase characters.

string-constant
A character string constant. The length must be between 6 and 127 bytes (SQLSTATE 428FC).

Notes

- The initial value of the ENCRYPTION PASSWORD is the empty string.
- The *host-variable* or *string-constant* is transmitted to the database server using normal database mechanisms.

Example

The following example shows how you can set the ENCRYPTION PASSWORD special register in an embedded SQL application using parameter markers. It is strongly recommended that this special register is always set up using parameter markers in your applications.

```
EXEC SQL BEGIN DECLARE SECTION;
      char hostVarSetEncPassStmt[200];
      char hostVarPassword[128];
EXEC SQL END DECLARE SECTION;

/* prepare the statement with a parameter marker */
strcpy(hostVarSetEncPassStmt, "SET ENCRYPTION PASSWORD = ?");
EXEC SQL PREPARE hostVarSetEncPassStmt FROM :hostVarSetEncPassStmt;

/* execute the statement for hostVarPassword = 'Gre89Ea' */
strcpy(hostVarPassword, "Gre89Ea");
EXEC SQL EXECUTE hostVarSetEncPassStmt USING :hostVarPassword;
```

SET EVENT MONITOR STATE

The SET EVENT MONITOR STATE statement activates or deactivates an event monitor. The current state of an event monitor (active or inactive) is determined by using the EVENT_MON_STATE built-in function.

The SET EVENT MONITOR STATE statement is not under transaction control.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include DBADM or SQLADM authority.

Syntax

Description

event-monitor-name
Identifies the event monitor to activate or deactivate. The name must identify an event monitor that exists in the catalog (SQLSTATE 42704).

new-state
new-state can be specified either as an integer constant or as the name of a host variable that will contain the appropriate value at run time. The following values can be specified:

0
Indicates that the specified event monitor should be deactivated.

1
Indicates that the specified event monitor should be activated. The event monitor should not already be active; otherwise a warning (SQLSTATE 01598) is issued.

host-variable
The data type is INTEGER. The value specified must be 0 or 1 (SQLSTATE 42815). If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

Rules

- Although an unlimited number of event monitors may be defined, a maximum of 128 event monitors can be active simultaneously on each database partition. In a multiple partition database environment, a maximum of 32 GLOBAL event monitors can be active simultaneously on each database.
- In order to activate an event monitor, the transaction in which the event monitor was created must have been committed (SQLSTATE 55033). This rule prevents (in one unit of work) creating an event monitor, activating the monitor, then rolling back the transaction.
- If the number or size of the event monitor files exceeds the values specified for MAXFILES or MAXFILESIZE on the CREATE EVENT MONITOR statement, an error (SQLSTATE 54031) is raised.

- If the target path of the event monitor (that was specified on the CREATE EVENT MONITOR statement) is already in use by another event monitor, an error (SQLSTATE 51026) is raised.

Notes

- Activating a non-WLM event monitor performs a reset of any counters associated with it. The reset of counters does not occur when activating WLM, locking, and unit of work event monitors.
- When a WRITE TO TABLE event monitor is started using SET EVENT MONITOR STATE, it updates the EVMON_ACTIVATES column of the SYSCAT.EVENTMONITORS catalog view. If the unit of work in which the set operation was performed is rolled back for any reason, that catalog update is lost. When the event monitor is restarted, it will reuse the EVMON_ACTIVATES value that was rolled back.
- If the database partition on which the event monitor is to run is not active, event monitor activation occurs when that database partition next activates.
- After an event monitor is activated, it behaves like an autostart event monitor until that event monitor is explicitly deactivated or the instance is recycled. That is, if an event monitor is active when a database partition is deactivated, and that database partition is subsequently reactivated, the event monitor is also explicitly reactivated.
- If an activity event monitor is active when the database deactivates, any backlogged activity records in the queue are discarded. To ensure that you obtain all activity event monitor records and that none are discarded, explicitly deactivate the activity event monitor first before deactivating the database. When an activity event monitor is explicitly deactivated, all backlogged activity records in the queue are processed before the event monitor deactivates.

Examples

- *Example 1:* Activate an event monitor named SMITHPAY.

```
SET EVENT MONITOR SMITHPAY STATE = 1
```

- *Example 2:* Assume that MYSAMPLE is a multiple partition database with two database partitions, 0 and 2. Partition 2 is not yet active.

```
On database partition 0:

   CONNECT TO MYSAMPLE;
   CREATE EVENT MONITOR MYEVMON ON DBPARTITIONNUM 2;
   SET EVENT MONITOR MYEVMON STATE 1;
```

  MYEVMON automatically activates whenever MYSAMPLE activates on database partition 2. This occurs until **SET EVENT MONITOR** MYEVMON **STATE** 0 is issued, or partition 2 is stopped.

SET INTEGRITY

The SET INTEGRITY statement is used to set the integrity pending state on tables, place tables into full access state, and prune the contents of one or more staging tables.

The following operations can be performed with the SET INTEGRITY statement:

- Bring one or more tables out of set integrity pending state (previously known as "check pending state") by performing required integrity processing on those tables.
- Bring one or more tables out of set integrity pending state without performing required integrity processing on those tables.
- Place one or more tables in set integrity pending state.
- Place one or more tables into full access state.
- Prune the contents of one or more staging tables.

When the statement is used to perform integrity processing for a table after it has been loaded or attached, the system can incrementally process the table by checking only the appended portion for constraints violations. If the subject table is a materialized query table or a staging table, and load, attach,

or detach operations are performed on its underlying tables, the system can incrementally refresh the materialized query table or incrementally propagate to the staging table with only the delta portions of its underlying tables. However, there are some situations in which the system will not be able to perform such optimizations and will instead perform full integrity processing to ensure data integrity. Full integrity processing is done by checking the entire table for constraints violations, recomputing a materialized query table's definition, or marking a staging table as inconsistent. The latter implies that a full refresh of its associated materialized query table is required. There is also a situation in which you might want to explicitly request incremental processing by specifying the INCREMENTAL option.

The SET INTEGRITY statement is under transaction control.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges required to execute the SET INTEGRITY statement depend on the purpose, as outlined in the following list.

- Bringing tables out of set integrity pending state and performing the required integrity processing.

  The privileges held by the authorization ID of the statement must include at least one of the following:

  – CONTROL privilege on:

    - The tables on which integrity processing is performed and, if exception tables are provided for one or more of those tables, INSERT privilege on the exception tables or INSERTIN privilege on the schema containing the exception tables
    - All descendent foreign key tables, descendent immediate materialized query tables, and descendent immediate staging tables that will implicitly be placed in set integrity pending state by the statement

  – LOAD authority on the database or LOAD authority on the schema containing the table (with conditions). The following conditions must all be met before LOAD authority can be considered as providing valid privileges:

    - The required integrity processing does not involve the following actions:
      - Refreshing a materialized query table
      - Propagating to a staging table
      - Updating a generated or identity column
    - If exception tables are provided for one or more tables, the required access is granted for the duration of the integrity processing to the tables on which integrity processing is performed, and to the associated exception tables. That is:
      - SELECT and DELETE privilege on each table on which integrity processing is performed, or SELECTIN and DELETEIN privilege on the schema containing the table on which integrity processing is performed and
      - INSERT privilege on the exception tables or INSERTIN privilege on the schema containing the exception tables

  – DATAACCESS authority on the relevant schema

  – DATAACCESS authority

- Bringing tables out of set integrity pending state without performing the required integrity processing.

  The privileges held by the authorization ID of the statement must include at least one of the following:

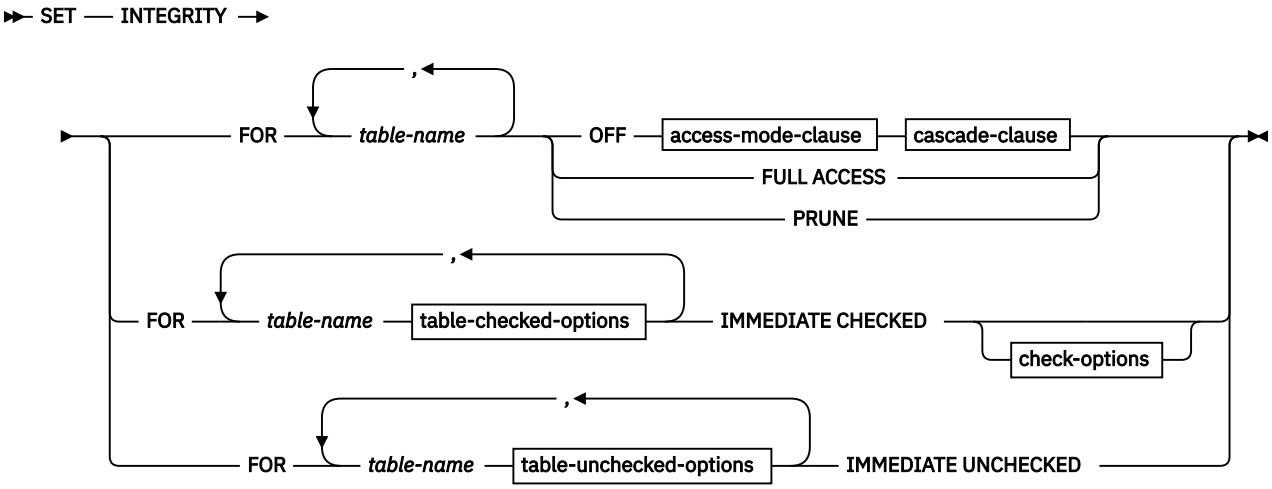
  - CONTROL privilege on the tables that are being processed; CONTROL privilege on each descendent foreign key table, descendent immediate materialized query table, and descendent immediate staging table that will implicitly be placed in set integrity pending state by the statement
  - LOAD authority on the database or LOAD authority on the relevant schema
  - DATAACCESS authority or DATAACCESS authority on the relevant schema
  - DBADM authority or SCHEMAADM authority on the relevant schema
- Placing tables in set integrity pending state.

  The privileges held by the authorization ID of the statement must include at least one of the following:
  - CONTROL privilege on:
    - The specified tables, and
    - The descendent foreign key tables that will be placed in set integrity pending state by the statement, and
    - The descendent immediate materialized query tables that will be placed in set integrity pending state by the statement, and
    - The descendent immediate staging tables that will be placed in set integrity pending state by the statement
  - LOAD authority on the database or LOAD authority on the relevant schema
  - DATAACCESS authority or DATAACCESS authority on the relevant schema
  - DBADM authority or SCHEMAADM authority on the relevant schema
- Place a table into the full access state.

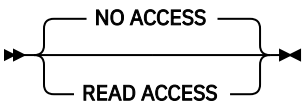
  The privileges held by the authorization ID of the statement must include at least one of the following:
  - CONTROL privilege on the tables that are placed into the full access state
  - LOAD authority on the database or LOAD authority on the relevant schema
  - DATAACCESS authority or DATAACCESS authority on the relevant schema
  - DBADM authority or SCHEMAADM authority on the relevant schema
- Prune a staging table.

  The privileges held by the authorization ID of the statement must include at least one of the following:
  - CONTROL privilege on the table being pruned
  - DATAACCESS authority or DATAACCESS authority on the relevant schema

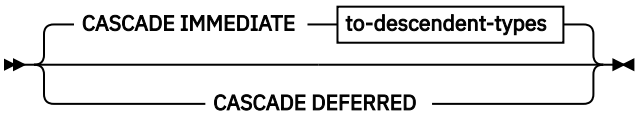
Syntax



access-mode-clause



cascade-clause



to-descendent-types

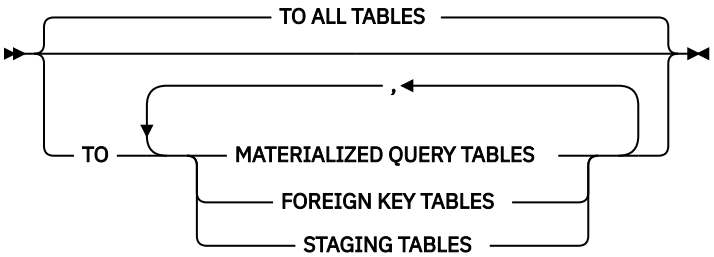
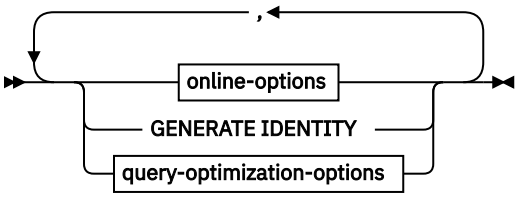
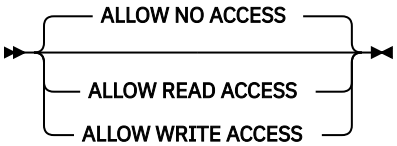


table-checked-options

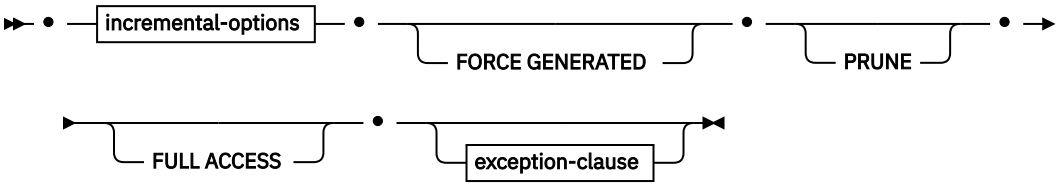


online-options

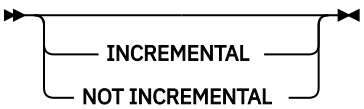


query-optimization-options

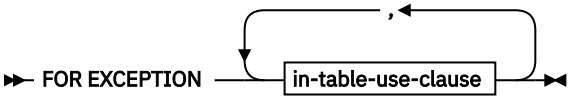
check-options



incremental-options

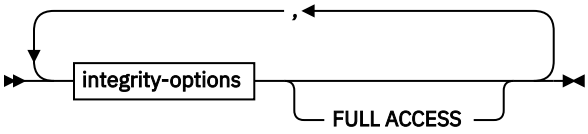


exception-clause

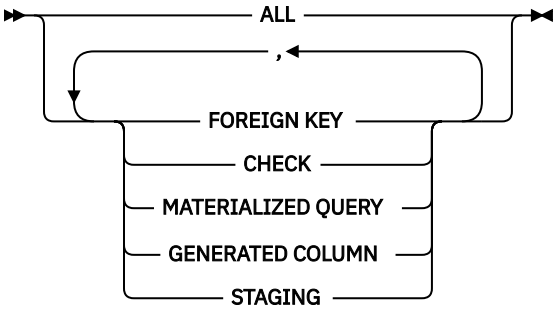


in-table-use-clause

table-unchecked-options



integrity-options



Description

FOR *table-name*
Identifies one or more tables for integrity processing. It must be a table described in the catalog and must not be a view, catalog table, or typed table.

OFF
Specifies that the tables are placed in set integrity pending state. Only very limited activity is allowed on a table that is in set integrity pending state.

access-mode-clause
Specifies the readability of the table while it is in set integrity pending state.

NO ACCESS
Specifies that the table is to be put in set integrity pending no access state, which does not allow read or write access to the table.

READ ACCESS
Specifies that the table is to be put in set integrity pending read access state, which allows read access to the non-appended portion of the table. This option is not allowed on a table that is in set integrity pending no access state (SQLSTATE 428FH).

cascade-clause
Specifies whether the set integrity pending state of the table referenced in the SET INTEGRITY statement is to be immediately cascaded to descendent tables.

CASCADE IMMEDIATE
Specifies that the set integrity pending state is to be immediately extended to descendent tables.

to-descendent-types
Specifies the type of descendent tables to which the set integrity pending state is immediately cascaded.

TO ALL TABLES
Specifies that the set integrity pending state is to be immediately cascaded to all descendent tables of the tables in the invocation list. Descendent tables include all descendent foreign key tables, immediate staging tables, and immediate materialized query tables that are descendants of the tables in the invocation list, or descendants of descendent foreign key tables.

Specifying TO ALL TABLES is equivalent to specifying TO FOREIGN KEY TABLES, TO MATERIALIZED QUERY TABLES, and TO STAGING TABLES, all in the same statement.

TO MATERIALIZED QUERY TABLES
If only TO MATERIALIZED QUERY TABLES is specified, the set integrity pending state is to be immediately cascaded only to descendent immediate materialized query tables. Other descendent tables might later be put in set integrity pending state, if necessary, when the table is brought out of set integrity pending state. If both TO FOREIGN KEY TABLES and TO MATERIALIZED QUERY TABLES are specified, the set integrity pending state will be immediately cascaded to all descendent foreign key tables, all descendent immediate materialized query tables of the tables in the invocation list, and to all immediate materialized query tables that are descendants of the descendent foreign key tables.

TO FOREIGN KEY TABLES
Specifies that the set integrity pending state is to be immediately cascaded to descendent foreign key tables. Other descendent tables might later be put in set integrity pending state, if necessary, when the table is brought out of set integrity pending state.

TO STAGING TABLES
Specifies that the set integrity pending state is to be immediately cascaded to descendent staging tables. Other descendent tables might later be put in set integrity pending state, if necessary, when the table is brought out of set integrity pending state. If both TO FOREIGN KEY TABLES and TO STAGING TABLES are specified, the set integrity pending state will be immediately cascaded to all descendent foreign key tables, all descendent immediate staging tables of the tables in the invocation list, and to all immediate staging tables that are descendants of the descendent foreign key tables.

CASCADE DEFERRED
Specifies that only the tables in the invocation list are to be put in set integrity pending state. The states of the descendent tables will remain unchanged. Descendent foreign key tables might later be implicitly put in set integrity pending state when their parent tables are checked for constraints violations. Descendent immediate materialized query tables and descendent immediate staging tables might be implicitly put in set integrity pending state when one of their underlying tables is checked for integrity violations. A query of a table that is in the set integrity pending state might succeed if an eligible materialized query table that is not in the set integrity pending state is accessed by the query instead of the specified table.

If *cascade-clause* is not specified, the set integrity pending state is immediately cascaded to all descendent tables.

IMMEDIATE CHECKED
Specifies that the table is to be taken out of set integrity pending state by performing required integrity processing on the table. This is done in accordance with the information set in the STATUS and CONST_CHECKED columns of the SYSCAT.TABLES catalog view. That is:

- The value in the STATUS column must be "C" (the table is in set integrity pending state), or an error is returned (SQLSTATE 51027), unless the table is a descendent foreign key table, descendent materialized query table, or descendent staging table of a table that is specified in the list, is in set integrity pending state, and whose intermediate ancestors are also in the list.
- If the table being checked is in set integrity pending state, the value in CONST_CHECKED indicates which integrity options are to be checked.

When the table is taken out of set integrity pending state, its descendent tables are, if necessary, put in set integrity pending state. A warning to indicate that descendent tables have been put in set integrity pending state is returned (SQLSTATE 01586).

If the table is a system-maintained materialized query table, the data is checked against the query and refreshed as necessary. (IMMEDIATE CHECKED cannot be used for user-maintained materialized query tables or shadow tables.) If the table is a staging table, the data is checked against its query definition and propagated as necessary.

When the integrity of a child table is checked:

- None of its parents can be in set integrity pending state, or
- Each of its parents must be checked for constraints violations in the same SET INTEGRITY statement

When an immediate materialized query table is refreshed, or deltas are propagated to a staging table:

- None of its underlying tables can be in set integrity pending state, or
- Each of its underlying tables must be checked in the same SET INTEGRITY statement

Otherwise, an error is returned (SQLSTATE 428A8).

table-checked-options

online-options
Specifies the accessibility of the table while it is being processed.

ALLOW NO ACCESS
Specifies that no other users can access the table while it is being processed, except if they are using the Uncommitted Read isolation level.

ALLOW READ ACCESS
Specifies that other users have read-only access to the table while it is being processed.

ALLOW WRITE ACCESS
Specifies that other users have read and write access to the table while it is being processed.

GENERATE IDENTITY
Specifies that if the table includes an identity column, the values are generated by the SET INTEGRITY statement. By default, when the GENERATE IDENTITY option is specified, only attached rows will have their identity column values generated by the SET INTEGRITY statement. The NOT INCREMENTAL option must be specified in conjunction with the GENERATE IDENTITY option to have the SET INTEGRITY statement generate identity column values for all rows in the table, including attached rows, loaded rows, and existing rows. If the GENERATE IDENTITY option is not specified, the current identity column values for all rows in the table are left unchanged. When the table is a system-period temporal table, GENERATE IDENTITY with the NOT INCREMENTAL option is allowed only if you first issue an ALTER TABLE statement with the DROP VERSIONING clause (SQLSTATE 428FH).

query-optimization-options
Specifies the query optimization options for the maintenance of REFRESH DEFERRED materialized query tables.

ALLOW QUERY OPTIMIZATION USING REFRESH DEFERRED TABLES WITH REFRESH AGE ANY
Specifies that when the CURRENT REFRESH AGE special register is set to "ANY", the maintenance of *table-name* will allow REFRESH DEFERRED materialized query tables to be used to optimize the query that maintains *table-name*. If *table-name* is not a REFRESH DEFERRED materialized query table, an error is returned (SQLSTATE 428FH). REFRESH IMMEDIATE materialized query tables are always considered during query optimization.

check-options

incremental-options

INCREMENTAL
Specifies the application of integrity processing on the appended portion (if any) of the table. If such a request cannot be satisfied (that is, the system detects that the whole table needs to be checked for data integrity), an error is returned (SQLSTATE 55019).

NOT INCREMENTAL
Specifies the application of integrity processing on the whole table. If the table is a materialized query table, the materialized query table definition is recomputed. If the table has at least one constraint defined on it, this option causes full processing of descendent foreign key tables and descendent immediate materialized query tables. If the table is a staging table, it is set to an inconsistent state.

If the *incremental-options* clause is not specified, the system determines whether incremental processing is possible; if not, the whole table is checked.

FORCE GENERATED
If the table includes generated by expression columns, the values are computed on the basis of the expression and stored in the column. If this option is not specified, the current values are compared to the computed value of the expression, as though an equality check constraint were in effect. If the table is processed for integrity incrementally, generated columns are computed only for the appended portion. When the table is a system-period temporal table, the FORCE GENERATED option is allowed only if you first issue an ALTER TABLE statement with the DROP VERSIONING clause (SQLSTATE 428FH).

PRUNE
This option can be specified for staging tables only. Specifies that the content of the staging table is to be pruned, and that the staging table is to be set to an inconsistent state. If any table in the *table-name* list is not a staging table, an error is returned (SQLSTATE 428FH). If the INCREMENTAL check option is also specified, an error is returned (SQLSTATE 428FH).

FULL ACCESS
Specifies that the table is to become fully accessible after the SET INTEGRITY statement executes.

When an underlying table (that has dependent immediate materialized query tables or dependent immediate staging tables) in the invocation list is incrementally processed, the underlying table is put in no data movement state, as required, after the SET INTEGRITY statement executes. When all incrementally refreshable dependent immediate materialized query tables and staging tables are taken out of set integrity pending state, the underlying table is automatically brought out of the no data movement state into the full access state. If the FULL ACCESS option is specified with the IMMEDIATE CHECKED option, the underlying table is put directly in full access state (bypassing the no data movement state). In Db2 Version 9.7. Fix Pack 1 and later, specifying the FULL ACCESS option only removes the dependency between the dependent tables and underlying table. The underlying table continues to be unavailable until the data partition detach process is completed by the asynchronous partition detach task.

Dependent immediate materialized query tables that have not been refreshed might undergo a full recomputation in the subsequent REFRESH TABLE statement, and dependent immediate staging tables that have not had the appended portions of the table propagated to them might be flagged as inconsistent.

When an underlying table in the invocation list requires full processing, or does not have dependent immediate materialized query tables, or dependent immediate staging tables, the underlying table is put directly into full access state after the SET INTEGRITY statement executes, regardless of whether the FULL ACCESS option was specified.

exception-clause

FOR EXCEPTION
Specifies that any row that is in violation of a constraint being checked is to be moved to an exception table. Even if errors are detected, the table is taken out of set integrity pending state. A warning to indicate that one or more rows have been moved to the exception tables is returned (SQLSTATE 01603).

If the FOR EXCEPTION option is not specified and any constraints are violated, only the first detected violation is returned (SQLSTATE 23514). If there is a violation in any table, all of the tables are left in set integrity pending state.

It is recommended to always use the FOR EXCEPTION option when checking for constraints violations to prevent a rollback of the SET INTEGRITY statement if a violation is found.

When the table specified after the IN keyword is a system-period temporal table, the FOR EXCEPTION option is allowed only if you first issue an ALTER TABLE statement with the DROP VERSIONING clause (SQLSTATE 428FH).

IN *table-name*
Specifies the table from which rows that violate constraints are to be moved. There must be one exception table specified for each table being checked. This clause cannot be specified for a materialized query table or a staging table (SQLSTATE 428A7).

USE *table-name*
Specifies the exception table into which error rows are to be moved.

FULL ACCESS
If the FULL ACCESS option is specified as the only operation of the statement, the table is placed into the full access state without being rechecked for integrity violations. However, dependent immediate materialized query tables that have not been refreshed might require a full recomputation in subsequent REFRESH TABLE statements, and dependent immediate staging tables that have not had the delta portions of the table propagated to them might be changed to incomplete state. This option can only be specified for a table that is in the no data movement state or the no access state, but not in the set integrity pending state (SQLSTATE 428FH).

PRUNE
This option can be specified for staging tables only. Specifies that the content of the staging table is to be pruned, and that the staging table is to be set to an inconsistent state. If any table in the *table-name* list is not a staging table, an error is returned (SQLSTATE 428FH).

table-unchecked-options

integrity-options
Used to define the types of required integrity processing that are to be bypassed when the table is taken out of the set integrity pending state.

ALL
The table will be immediately taken out of set integrity pending state without any of its required integrity processing being performed.

FOREIGN KEY
Required foreign key constraints checking will not be performed when the table is brought out of set integrity pending state.

CHECK
Required check constraints checking will not be performed when the table is brought out of set integrity pending state.

MATERIALIZED QUERY
Required refreshing of a materialized query table will not be performed when the table is brought out of set integrity pending state.

GENERATED COLUMN
Required generated column constraints checking will not be performed when the table is brought out of set integrity pending state.

STAGING
Required propagation of data to a staging table will not be performed when the table is brought out of set integrity pending state.

If no other types of integrity processing are required on the table after a specific type of integrity processing has been marked as bypassed, the table is immediately taken out of set integrity pending state.

FULL ACCESS
Specifies that the tables are to become fully accessible after the SET INTEGRITY statement executes.

When an underlying table in the invocation list is incrementally processed, and it has dependent immediate materialized query tables or dependent immediate staging tables, the underlying table is placed, as required, in the no data movement state after the SET INTEGRITY statement executes. When all incrementally refreshable dependent immediate materialized query tables and staging tables have been taken out of set integrity pending state, the underlying table is automatically brought out of the no data movement state into the full access state. If the

FULL ACCESS option is specified with the IMMEDIATE UNCHECKED option, the underlying table is placed directly in full access state (it bypasses the no data movement state). Dependent immediate materialized query tables that have not been refreshed might undergo a full recomputation in the subsequent REFRESH TABLE statement, and dependent immediate staging tables that have not had the appended portions of the table propagated to them might be flagged as inconsistent.

In Db2 V9.7. Fix Pack 1 and later, specifying the FULL ACCESS option only removes the dependency between the dependent tables and underlying table. The underlying table continues to be unavailable until the data partition detach process is completed by the asynchronous partition detach task.

When an underlying table in the invocation list requires full processing, or does not have dependent immediate materialized query tables, or dependent immediate staging tables, the underlying table is placed directly in full access state after the SET INTEGRITY statement executes, regardless of whether the FULL ACCESS option has been specified.

If the FULL ACCESS option has been specified with the IMMEDIATE UNCHECKED option, and the statement does not bring the table out of set integrity pending state, an error is returned (SQLSTATE 428FH).

IMMEDIATE UNCHECKED
Specifies one of the following:

- The table is to be brought out of set integrity pending state immediately without any required integrity processing.
- The table is to have one or more types of required integrity processing bypassed when the table is brought out of set integrity pending state by a subsequent SET INTEGRITY statement using the IMMEDIATE CHECKED option.

Consider the data integrity implications of this option before using it. See the "Notes" section.

Notes

- The following restrictions apply to the SMP parallelization of set integrity checking:
  - SMP parallelization of set integrity checking is not enabled by default. The **DB2_EXTENDED_OPTIMIZATION** registry variable with the PRLSI option must be set in order to enable it.
  - The CURRENT DEGREE special register should be used to ensure an appropriate degree of parallelism.
  - SMP parallelization is not supported in MPP environments.
  - SMP parallelization is not supported for INCREMENTAL checking or set integrity after attach.
  - SMP parallelization is not supported when any of the following are involved:
    - Materialized query tables or staging tables
    - Identity or generated columns
  - Parallelization will be limited when there are multiple tables referenced or self-references involved. The constraint and referential checking will be parallelized for one table at the most. The checks for the table with the most expensive constraint checking will be the one which may be parallelized. However, if a table has multiple self-references, some or all of its constraint checking may not be parallelized.
  - Exception handling will not be parallelized.
- Effects on tables in one of the restricted set integrity-related states:
  - Use of INSERT, UPDATE, or DELETE is disallowed on a table that is in read access state or in no access state. Furthermore, any statement that requires this type of modification to a table that is in such a state will be rejected. For example, deletion of a row in a parent table that cascades to a dependent table that is in the no access state is not allowed.

- Use of SELECT is disallowed on a table that is in the no access state. Furthermore, any statement that requires read access to a table that is in the no access state will be rejected.
- New constraints added to a table are normally enforced immediately. However, if the table is in set integrity pending state, the checking of any new constraints is deferred until the table is taken out of set integrity pending state. If the table is in set integrity pending state, addition of a new constraint places the table into set integrity pending no access state, because validity of data is at risk.
- The CREATE INDEX statement cannot reference any table that is in read access state or in no access state. Similarly, an ALTER TABLE statement to add a primary key or a unique constraint cannot reference any table that is in read access state or in no access state.
- The import utility is not allowed to operate on a table that is in read access state or in no access state.
- The export utility is not allowed to operate on a table that is in no access state, but is allowed to operate on a table that is in read access state. If a table is in read access state, the export utility will only export the data that is in the non-appended portion.
- Operations (like REORG, REDISTRIBUTE, update distribution key, update multidimensional clustering key, update range clustering key, update table partitioning key, and so on) that might involve data movement within a table are not allowed on a table that is in any of the following states: read access, no access, or no data movement.
- The load, backup, restore, update statistics, runstats, reorgchk, list history, and rollforward utilities are allowed on a table that is in any of the following states: full access, read access, no access, or no data movement.
- The ALTER TABLE, COMMENT, DROP TABLE, CREATE ALIAS, CREATE TRIGGER, CREATE VIEW, GRANT, REVOKE, and SET INTEGRITY statements can reference a table that is in any of the following states: full access, read access, no access, or no data movement. However, they might cause the table to be put into no access state.
- Packages, views, and any other objects that depend on a table that is in no access state will return an error when the table is accessed at run time. Packages that depend on a table that is in read access state will return an error when an insert, update, or delete operation is attempted on the table at run time.
- The ALL or GENERATED COLUMN option cannot be specified with the IMMEDIATE UNCHECKED option if the table's database partitioning key, table-partitioning key, multidimensional clustering key, or range-clustering key references a generated column whose expression was altered through an ALTER TABLE statement.

The removal of violating rows by the SET INTEGRITY statement is not a delete event. Therefore, triggers are never activated by a SET INTEGRITY statement. Similarly, updating generated columns using the FORCE GENERATED option does not activate triggers.

- Warning about the use of the IMMEDIATE UNCHECKED clause:
  - This clause is intended to be used by utility programs, and its use by application programs is not recommended. If there is data in the table that does not meet the integrity specifications that were defined for the table, and the IMMEDIATE UNCHECKED option is used, incorrect query results might be returned.

    The fact that the table was taken out of the set integrity pending state without performing the required integrity processing will be recorded in the catalog (the respective byte in the CONST_CHECKED column in the SYSCAT.TABLES view will be set to "U"). This indicates that the user has assumed responsibility for data integrity with respect to the specific constraints. This value remains unchanged until either:

    - The table is put back into set integrity pending state (by referencing the table in a SET INTEGRITY statement with the OFF option), at which time "U" values in the CONST_CHECKED column are changed to "W" values, indicating that the user had previously assumed responsibility for data integrity, and the system needs to verify the data.
    - All unchecked constraints for the table are dropped.

The "W" state differs from the "N" state in that it records the fact that integrity was previously checked by the user, but not yet by the system. If the user issues the SET INTEGRITY ... IMMEDIATE CHECKED statement with the NOT INCREMENTAL option, the system rechecks the whole table for data integrity (or performs a full refresh on a materialized query table), and then changes the "W" state to the "Y" state. If IMMEDIATE UNCHECKED is specified, or if NOT INCREMENTAL is not specified, the "W" state is changed back to the "U" state to record the fact that some data has still not been verified by the system. In the latter case (when the NOT INCREMENTAL is not specified), a warning is returned (SQLSTATE 01636).

If an underlying table's integrity has been checked using the IMMEDIATE UNCHECKED clause, the "U" values in the CONST_CHECKED column of the underlying table will be propagated to the corresponding CONST_CHECKED column of:

- Dependent immediate materialized query tables
- Dependent deferred materialized query tables
- Dependent staging tables

For a dependent immediate materialized query table, this propagation is done whenever the underlying table is brought out of set integrity pending state, and whenever the materialized query table is refreshed. For a dependent deferred materialized query table, this propagation is done whenever the materialized query table is refreshed. For dependent staging tables, this propagation is done whenever the underlying table is brought out of set integrity pending state. These propagated "U" values in the CONST_CHECKED columns of dependent materialized query tables and staging tables record the fact that these materialized query tables and staging tables depend on some underlying table whose required integrity processing has been bypassed using the IMMEDIATE UNCHECKED option.

For a materialized query table, the "U" value in the CONST_CHECKED column that was propagated by the underlying table will remain until the materialized query table is fully refreshed and none of its underlying tables have a "U" value in their corresponding CONST_CHECKED column. After such a refresh, the "U" value in the CONST_CHECKED column for the materialized query table will be changed to "Y".

For a staging table, the "U" value in the CONST_CHECKED column that was propagated by the underlying table will remain until the corresponding deferred materialized query table of the staging table is refreshed. After such a refresh, the "U" value in the CONST_CHECKED column for the staging table will be changed to "Y".

– If a child table and its parent table are checked in the same SET INTEGRITY statement with the IMMEDIATE CHECKED option, and the parent table requires full checking of its constraints, the child table will have its foreign key constraints checked, independently of whether or not the child table has a "U" value in the CONST_CHECKED column for foreign key constraints.

- If the table is data partitioned and there are nonpartitioned indexes (except the XML column path index) to maintain, IMMEDIATE UNCHECKED behavior when a single target table is specified is the same as IMMEDIATE CHECKED behavior with the ALLOW WRITE ACCESS option: all integrity processing is performed and any resulting errors are returned. If the statement references more than one target table, an error is returned (SQLSTATE 428FH).
- After appending data using LOAD INSERT or ALTER TABLE ATTACH, the SET INTEGRITY statement with the IMMEDIATE CHECKED option checks the table for constraints violations. The system determines whether incremental processing on the table is possible. If so, only the appended portion is checked for integrity violations. If not, the system checks the whole table for integrity violations.
- Consider the statement:

```
SET INTEGRITY FOR T IMMEDIATE CHECKED
```

In the following scenarios, neither the INCREMENTAL check option for T nor an incremental refresh of T---if T is a materialized query table (MQT) or a staging table---is supported:

– New constraints have been added to T while it is in set integrity pending state

– When a LOAD REPLACE operation against T, it parents, or its underlying tables has taken place

  - When the NOT LOGGED INITIALLY WITH EMPTY TABLE option has been activated after the last integrity check on T, its parents, or its underlying tables
  - The cascading effect of full processing, when any parent of T (or underlying table, if T is a materialized query table or a staging table) has been checked for integrity non-incrementally
  - If the table space containing the table or its parent (or underlying table of a materialized query table or a staging table) has been rolled forward to a point in time, and the table and its parent (or underlying table if the table is a materialized query table or a staging table) reside in different table spaces
  - T is an MQT, and a LOAD REPLACE or LOAD INSERT operation directly into T has taken place after the last refresh
- Incremental processing will be used whenever the situation allows it, because it is more efficient. The INCREMENTAL option is not needed in most cases. It is needed, however, to ensure that integrity checks are indeed processed incrementally. If the system detects that full processing is needed to ensure data integrity, an error is returned (SQLSTATE 55019).
- If the conditions for full processing described in the previous bullet are not satisfied, the system will attempt to check only the appended portion for integrity, or perform an incremental refresh (if it is a materialized query table) when the user does not specify the NOT INCREMENTAL option for the statement `SET INTEGRITY FOR T IMMEDIATE CHECKED`.
- If an error occurs during integrity processing, all the effects of the processing (including deleting from the original and inserting into the exception tables) will be rolled back.
- If a SET INTEGRITY statement issued with the FORCE GENERATED option fails because of a lack of log space, increase available active log space and reissue the SET INTEGRITY statement. Alternatively, use the SET INTEGRITY statement with the GENERATED COLUMN and IMMEDIATE UNCHECKED options to bypass generated column checking for the table. Then, issue a SET INTEGRITY statement with the IMMEDIATE CHECKED option and without the FORCE GENERATED option to check the table for other integrity violations (if applicable) and to bring it out of set integrity pending state. After the table is out of the set integrity pending state, the generated columns can be updated to their default (generated) values by assigning them to the keyword DEFAULT in an UPDATE statement. This is accomplished by using either multiple searched update statements based on ranges (each followed by a commit), or a cursor-based approach using intermittent commits. A "with hold" cursor should be used if locks are to be retained after intermittent commits using the cursor-based approach.
- A table that was put into set integrity pending state using the CASCADE DEFERRED option of the SET INTEGRITY statement or the LOAD command, or through the ALTER TABLE statement with the ATTACH clause, and that is checked for integrity violations using the IMMEDIATE CHECKED option of the SET INTEGRITY statement, will have its descendent foreign key tables, descendent immediate materialized query tables, and descendent immediate staging tables put in set integrity pending state, as required:
  - If the entire table is checked for integrity violations, its descendent foreign key tables, descendent immediate materialized query tables, and descendent immediate staging tables will be put in set integrity pending state.
  - If the table is checked for integrity violations incrementally, its descendent immediate materialized query tables and staging tables will be put in set integrity pending state, and its descendent foreign key tables will remain in their original states.
  - If the table requires no checking at all, its descendent immediate materialized query tables, descendent staging tables, and descendent foreign key tables will remain in their original states.
- A table that was put in set integrity pending state using the CASCADE DEFERRED option (of the SET INTEGRITY statement or the LOAD command), and that is brought out of set integrity pending state using the IMMEDIATE UNCHECKED option of the SET INTEGRITY statement, will have its descendent foreign key tables, descendent immediate materialized query tables, and descendent immediate staging tables put in set integrity pending state, as required:
  - If the table has been loaded using the REPLACE mode, its descendent foreign key tables, descendent immediate materialized query tables, and descendent immediate staging tables will be put in set integrity pending state.

  - If the table has been loaded using the INSERT mode, its descendent immediate materialized query tables and staging tables will be put in set integrity pending state, and its descendent foreign key tables will remain in their original states.
  - If the table has not been loaded, its descendent immediate materialized query tables, descendent staging tables, and its descendent foreign key tables will remain in their original states.
- SET INTEGRITY is usually a long running statement. In light of this, to reduce the risk of a rollback of the entire statement because of a lock timeout, you can issue the SET CURRENT LOCK TIMEOUT statement with the WAIT option before executing the SET INTEGRITY statement, and then reset the special register to its previous value after the transaction commits. Note, however, that the CURRENT LOCK TIMEOUT special register only impacts a specific set of lock types.
- If you use the ALLOW QUERY OPTIMIZATION USING REFRESH DEFERRED TABLES WITH REFRESH AGE ANY option, ensure that the maintenance order is correct for REFRESH DEFERRED materialized query tables. For example, consider two materialized query tables, MQT1 and MQT2, whose materialized queries share the same underlying tables. The materialized query for MQT2 can be calculated using MQT1, instead of the underlying tables. If separate statements are used to maintain these two materialized query tables, and MQT2 is maintained first, the system might choose to use the contents of MQT1, which has not yet been maintained, to maintain MQT2. In this case, MQT1 would contain current data, but MQT2 could still contain stale data, even though both were maintained at almost the same time. The correct maintenance order, if two SET INTEGRITY statements are used instead of one, is to maintain MQT1 first.
- When using the SET INTEGRITY statement to perform integrity processing on a base table that has been loaded or attached, it is recommended that you process its dependent REFRESH IMMEDIATE materialized query tables and its PROPAGATE IMMEDIATE staging tables in the same SET INTEGRITY statement to avoid putting these dependent tables in set integrity pending no access state at the end of SET INTEGRITY processing. Note that for base tables that have a large number of dependent REFRESH IMMEDIATE materialized query tables and PROPAGATE IMMEDIATE staging tables, memory constraints might make it impossible to process all of the dependents in the same statement as the base table.
- If the FORCE GENERATED or the GENERATE IDENTITY option is specified, and the column that is generated is part of a unique index, the SET INTEGRITY statement returns an error (SQLSTATE 23505) and rolls back if it detects duplicate keys in the unique index. This error is returned even if there is an exception table for the table being processed.

  This scenario can occur under the following circumstances:

  - The SET INTEGRITY statement runs after a LOAD command against the table, and the GENERATEDOVERRIDE or the IDENTITYOVERRIDE file type modifier is specified during the load operation. To prevent this scenario, it is recommended that you use the GENERATEDIGNORE or the GENERATEDMISSING file type modifier instead of GENERATEDOVERRIDE, and that you use the IDENTITYIGNORE or the IDENTITYMISSING modifier instead of IDENTITYOVERRIDE. Using the recommended modifiers will prevent the need for any generated by expression column or identity column processing during SET INTEGRITY statement execution.
  - The SET INTEGRITY statement is run after an ALTER TABLE statement that alters the expression of a generated by expression column.

  To bring a table out of the set integrity pending state after encountering such a scenario:

  - Do not use the FORCE GENERATED or the GENERATE IDENTITY option to regenerate the column values. Instead, use the IMMEDIATE CHECKED option in conjunction with the FOR EXCEPTION option to move any rows that violate the generated column expression to an exception table. Then, re-insert the rows into the table from the exception table, which will generate the correct expression and perform unique key checking. This prevents having to reprocess the entire table, because only those rows that violated the generated column expression will need to be processed again.
  - If the table being processed has attached partitions, detach those partitions before performing the actions that are described in the previous bullet. Then, re-attach the partitions and execute a SET INTEGRITY statement to process integrity on the attached partitions separately.
- If a protected table is specified for the SET INTEGRITY statement along with an exception table, all of the following table criteria must be met; otherwise, an error is returned (SQLSTATE 428A5):

  - The tables must be protected by the same security policy.
  - If a column in the protected table has data type DB2SECURITYLABEL, the corresponding column in the exception table must also have data type DB2SECURITYLABEL.
  - If a column in the protected table is protected by a security label, the corresponding column in the exception table must also be protected by the same security label.
- Rows that violate the integrity being checked in a system-period temporal table cannot be moved to an exception table. If the violating rows must be moved to an exception table, the table must be altered to drop versioning before issuing the SET INTEGRITY statement with the FOR EXCEPTION clause.
- ***Syntax alternatives***: The following are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - SET CONSTRAINTS can be specified in place of SET INTEGRITY
  - SUMMARY can be specified in place of MATERIALIZED QUERY

Examples

- *Example 1:* The following is an example of a query that provides information about the set integrity pending state and the set integrity-related access restriction states of tables. SUBSTR is used to extract individual bytes of the CONST_CHECKED column of SYSCAT.TABLES. The first byte represents foreign key constraints; the second byte represents check constraints; the fifth byte represents materialized query table integrity; the sixth byte represents generated column constraints; the seventh byte represents staging table integrity; and the eighth byte represents data partitioning constraints. STATUS gives the set integrity pending state, and ACCESS_MODE gives the set integrity-related access restriction state.

```
SELECT TABNAME, STATUS, ACCESS_MODE,
  SUBSTR(CONST_CHECKED,1,1) AS FK_CHECKED,
  SUBSTR(CONST_CHECKED,2,1) AS CC_CHECKED,
  SUBSTR(CONST_CHECKED,5,1) AS MQT_CHECKED,
  SUBSTR(CONST_CHECKED,6,1) AS GC_CHECKED,
  SUBSTR(CONST_CHECKED,7,1) AS STG_CHECKED,
  SUBSTR(CONST_CHECKED,8,1) AS DP_CHECKED
FROM SYSCAT.TABLES
```

- *Example 2:* Put the PARENT table in set integrity pending no access state, and immediately cascade the set integrity pending state to its descendants.

```
SET INTEGRITY FOR PARENT OFF
  NO ACCESS CASCADE IMMEDIATE
```

- *Example 3:* Put the PARENT table in set integrity pending read access state without immediately cascading the set integrity pending state to its descendants.

```
SET INTEGRITY FOR PARENT OFF
  READ ACCESS CASCADE DEFERRED
```

- *Example 4:* Check integrity for a table named FACT_TABLE. If there are no integrity violations detected, the table is brought out of set integrity pending state. If any integrity violations are detected, the entire statement is rolled back, and the table remains in set integrity pending state.

```
SET INTEGRITY FOR FACT_TABLE IMMEDIATE CHECKED
```

- *Example 5:* Check integrity for the SALES and PRODUCTS tables, and move the rows that violate integrity into exception tables named SALES_EXCEPTIONS and PRODUCTS_EXCEPTIONS. Both the SALES and PRODUCTS tables are brought out of set integrity pending state, whether or not there are any integrity violations.

```
SET INTEGRITY FOR SALES, PRODUCTS IMMEDIATE CHECKED
  FOR EXCEPTION IN SALES USE SALES_EXCEPTIONS,
  IN PRODUCTS USE PRODUCTS_EXCEPTIONS
```

- *Example 6:* Enable FOREIGN KEY constraint checking in the MANAGER table, and CHECK constraint checking in the EMPLOYEE table, to be bypassed with the IMMEDIATE UNCHECKED option.

```
SET INTEGRITY FOR MANAGER FOREIGN KEY,
  EMPLOYEE CHECK IMMEDIATE UNCHECKED
```

- *Example 7:* Add a check constraint and a foreign key to the EMP_ACT table, using two ALTER TABLE statements. The SET INTEGRITY statement with the OFF option is used to put the table in set integrity pending state, so that the constraints are not checked immediately upon execution of the two ALTER TABLE statements. The single SET INTEGRITY statement with the IMMEDIATE CHECKED option is used to check both of the added constraints during a single pass through the table.

```
SET INTEGRITY FOR EMP_ACT OFF;
ALTER TABLE EMP_ACT ADD CHECK
  (EMSTDATE <= EMENDATE);
ALTER TABLE EMP_ACT ADD FOREIGN KEY
  (EMPNO) REFERENCES EMPLOYEE;
SET INTEGRITY FOR EMP_ACT IMMEDIATE CHECKED
  FOR EXCEPTION IN EMP_ACT USE EMP_ACT_EXCEPTIONS
```

- *Example 8:* Update generated columns with the correct values.

```
SET INTEGRITY FOR SALES IMMEDIATE CHECKED
  FORCE GENERATED
```

- *Example 9:* Append (using LOAD INSERT) from different sources into an underlying table (SALES) of a REFRESH IMMEDIATE materialized query table (SALES_SUMMARY). Check SALES incrementally for data integrity, and refresh SALES_SUMMARY incrementally. In this scenario, integrity checking for SALES and refreshing of SALES_SUMMARY are incremental, because the system chooses incremental processing. The ALLOW READ ACCESS option is used on the SALES table to allow concurrent reads of existing data while integrity checking of the loaded portion of the table is taking place.

```
LOAD FROM 2000_DATA.DEL OF DEL
  INSERT INTO SALES ALLOW READ ACCESS;
LOAD FROM 2001_DATA.DEL OF DEL
  INSERT INTO SALES ALLOW READ ACCESS;
SET INTEGRITY FOR SALES ALLOW READ ACCESS IMMEDIATE CHECKED
  FOR EXCEPTION IN SALES USE SALES_EXCEPTIONS;
REFRESH TABLE SALES_SUMMARY;
```

- *Example 10:* Attach a new partition to a data partitioned table named SALES. Incrementally check for constraints violations in the attached data of the SALES table and incrementally refresh the dependent SALES_SUMMARY table. The ALLOW WRITE ACCESS option is used on both tables to allow concurrent updates while integrity checking is taking place.

```
ALTER TABLE SALES
  ATTACH PARTITION STARTING (100) ENDING (200)
  FROM SOURCE;
SET INTEGRITY FOR SALES ALLOW WRITE ACCESS, SALES_SUMMARY ALLOW WRITE ACCESS
  IMMEDIATE CHECKED FOR EXCEPTION IN SALES
  USE SALES_EXCEPTIONS;
```

- *Example 11:* Detach a partition from a data partitioned table named SALES. Incrementally refresh the dependent SALES_SUMMARY table.

```
ALTER TABLE SALES
  DETACH PARTITION 2000_PART INTO ARCHIVE_TABLE;
SET INTEGRITY FOR SALES_SUMMARY
  IMMEDIATE CHECKED;
```

- *Example 12:* Bring a new user-maintained materialized query table out of set integrity pending state.

```
CREATE TABLE YEARLY_SALES
  AS (SELECT YEAR, SUM(SALES)AS SALES
  FROM FACT_TABLE GROUP BY YEAR)
  DATA INITIALLY DEFERRED REFRESH DEFERRED MAINTAINED BY USER

SET INTEGRITY FOR YEARLY_SALES
  ALL IMMEDIATE UNCHECKED
```

- *Example 13:* Attach a new partition to a data partitioned table named SALES. Assume that this table has no nonpartitioned user indexes. Assume also that data integrity checking, including range validation and other constraints checking, has already been done (through application logic that is independent of the data server). Optimize the data roll-in process by using the SET INTEGRITY ... ALL IMMEDIATE UNCHECKED statement to skip range and constraints violation checking.

```
ALTER TABLE SALES
  ATTACH PARTITION STARTING (300) ENDING (400)
  FROM SOURCE_TABLE;
SET INTEGRITY FOR SALES ALL IMMEDIATE UNCHECKED;
```

  The SALES table is brought out of SET INTEGRITY pending state, and the new data is available for applications to use immediately.
- *Example 14*: Setting the **DB2_EXTENDED_OPTIMIZATION** registry variable with the option is done by running:

```
db2set DB2_EXTENDED_OPTIMIZATION=
```

SET PASSTHRU

The SET PASSTHRU statement opens and closes a session for submitting a data source's native SQL directly to that data source.

The statement is not under transaction control.

Invocation

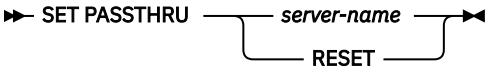
This statement can be issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must provide authorization to:

- Pass through to the data source
- Satisfy security measures at the data source

Syntax



Description

server-name
Names the data source for which a pass-through session is to be opened. *server-name* must identify a data source that is described in the catalog.

RESET
Closes a pass-through session.

Notes

- The following restrictions apply to Microsoft SQL Server, Sybase, and Oracle data sources:
  - User-defined transactions cannot be used for Microsoft SQL Server and Sybase data sources in pass-through mode, because Microsoft SQL Server and Sybase restrict which SQL statements can be specified within a user-defined transaction. Because SQL statements that are processed in pass-

through mode are not parsed by the database manager, it is not possible to detect whether the user specified an SQL statement that is permitted within a user-defined transaction.

- The COMPUTE clause is not supported on Microsoft SQL Server and Sybase data sources.
- DDL statements are not subject to transaction semantics on Microsoft SQL Server, Oracle and Sybase data sources. The operation, when complete, is automatically committed by Microsoft SQL Server, Oracle or Sybase. If a rollback occurs, the DDL is not rolled back.

Examples

- *Example 1:* Start a pass-through session to data source BACKEND.

```
strcpy (PASS_THRU,"SET PASSTHRU BACKEND");
EXEC SQL EXECUTE IMMEDIATE :PASS_THRU;
```

- *Example 2:* Start a pass-through session with a PREPARE statement.

```
strcpy (PASS_THRU,"SET PASSTHRU BACKEND");
EXEC SQL PREPARE STMT FROM :PASS_THRU;
EXEC SQL EXECUTE STMT;
```

- *Example 3:* End a pass-through session.

```
strcpy (PASS_THRU_RESET,"SET PASSTHRU RESET");
EXEC SQL EXECUTE IMMEDIATE :PASS_THRU_RESET;
```

- *Example 4:* Use the PREPARE and EXECUTE statements to end a pass-through session.

```
strcpy (PASS_THRU_RESET,"SET PASSTHRU RESET");
EXEC SQL PREPARE STMT FROM :PASS_THRU_RESET;
EXEC SQL EXECUTE STMT;
```

- *Example 5:* Open a session to pass through to a data source, create a clustered index for a table at this data source, and close the pass-through session.

```
strcpy (PASS_THRU,"SET PASSTHRU BACKEND");
EXEC SQL EXECUTE IMMEDIATE :PASS_THRU;
EXEC SQL PREPARE STMT                 pass-through mode
  FROM "CREATE UNIQUE
        CLUSTERED INDEX TABLE_INDEX
        ON USER2.TABLE              table is not an
        WITH IGNORE DUP KEY";        alias
EXEC SQL EXECUTE STMT;
strcpy (PASS_THRU_RESET,"SET PASSTHRU RESET");
EXEC SQL EXECUTE IMMEDIATE :PASS_THRU_RESET;
```

SET PATH

The SET PATH statement changes the value of the CURRENT PATH special register.

This statement is not under transaction control.

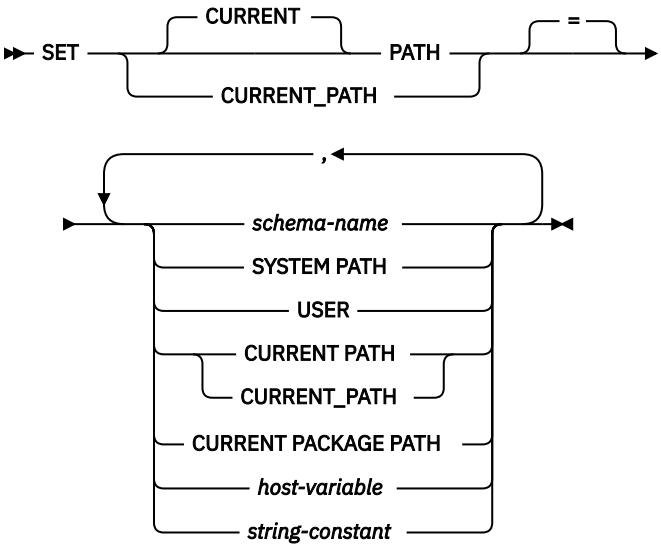
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

schema-name
This one-part name identifies a schema that exists at the application server. No validation that the schema exists is made at the time that the path is set. If a *schema-name* is, for example, misspelled, the error will not be caught, and it could affect the way subsequent SQL operates.

SYSTEM PATH
This value is the same as specifying the schema names *"SYSIBM","SYSFUN","SYSPROC","SYSIBMADM"*.

USER
The value of the USER special register.

CURRENT PATH
The value of the CURRENT PATH special register before this statement executes.

CURRENT PACKAGE PATH
The value of the CURRENT PACKAGE PATH special register.

host-variable
A variable of type CHAR or VARCHAR. The length of the contents of the *host-variable* must not exceed 128 bytes (SQLSTATE 42815). It cannot be set to null. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

The characters of the *host-variable* must be left-aligned. When specifying the *schema-name* with a *host-variable*, all characters must be specified in the exact case intended as there is no conversion to uppercase characters.

string-constant
A character string constant with a maximum length of 128 bytes.

Rules

- A schema name cannot appear more than once in the SQL path (SQLSTATE 42732).
- The schema name SYSPUBLIC cannot be specified in the SQL path (SQLSTATE 42815).
- The number of schemas that can be specified is limited by the total length of the CURRENT PATH special register. The special register string is built by taking each schema name specified and removing trailing blanks, delimiting with double quotation marks, doubling quotation marks within the schema name as necessary, and then separating each schema name by a comma. The length of the resulting string cannot exceed 2048 bytes (SQLSTATE 42907).

Notes

- The initial value of the CURRENT PATH special register is "SYSIBM","SYSFUN","SYSPROC","SYSIBMADM","X" where X is the value of the USER special register.
- The schema SYSIBM does not need to be specified. If it is not included in the SQL path, it is implicitly assumed as the first schema (in this case, it is not included in the CURRENT PATH special register).
- The CURRENT PATH special register specifies the SQL path used to resolve function names, procedure names, data type names, global variable names, and module object names in dynamic SQL statements. The FUNCPATH bind option specifies the SQL path to be used for resolving function names, procedure names, data type names, global variable names, and module object names in static SQL statements.
- ***Syntax alternatives***: The following syntax alternatives are supported for compatibility with previous versions of Db2 and with other database products. These alternatives are non-standard and should not be used.
  - CURRENT FUNCTION PATH can be specified in place of CURRENT PATH

Examples

- *Example 1:* The following statement sets the CURRENT PATH special register.

```
SET PATH = FERMAT, "McDrw #8", SYSIBM
```

- *Example 2:* The following example retrieves the current value of the CURRENT PATH special register into the host variable called CURPATH.

```
EXEC SQL VALUES (CURRENT PATH) INTO :CURPATH;
```

  The value would be "FERMAT","McDrw #8","SYSIBM" if set by the previous example.

SET ROLE

The SET ROLE statement verifies that the authorization ID of the session is a member of a specific role. An authorization ID acquires membership in a role when the role is granted to the authorization ID, or to a group or role in which the authorization ID is a member.

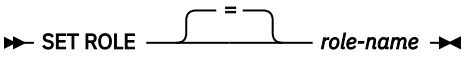
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

None required.

Syntax



Description

role-name

Specifies a role in whose membership the authorization ID of the session is to be verified. The *role-name* must identify an existing role at the current server (SQLSTATE 42704). If the authorization ID of the session is not a member of *role-name*, an error is returned (SQLSTATE 42501).

Notes

- All roles that have been granted to an authorization ID are used for authorization checking. The SET ROLE statement does not affect which roles are used for this authorization checking. Use the GRANT ROLE and REVOKE ROLE statements to change the roles in which an authorization ID has membership.

Examples

- *Example 1:* User WALID has been granted the role EDITOR, but not the role AUTHOR. Verify that WALID is a member of the EDITOR role.

```
SET ROLE EDITOR
```

- *Example 2:* Verify that WALID is not a member of the AUTHOR role. The following statement returns an error (SQLSTATE 42501).

```
SET ROLE AUTHOR
```

SET SCHEMA

The SET SCHEMA statement changes the value of the CURRENT SCHEMA special register.

This statement is not under transaction control. If the package is bound with the DYNAMICRULES BIND option, this statement does not affect the qualifier used for unqualified database object references.

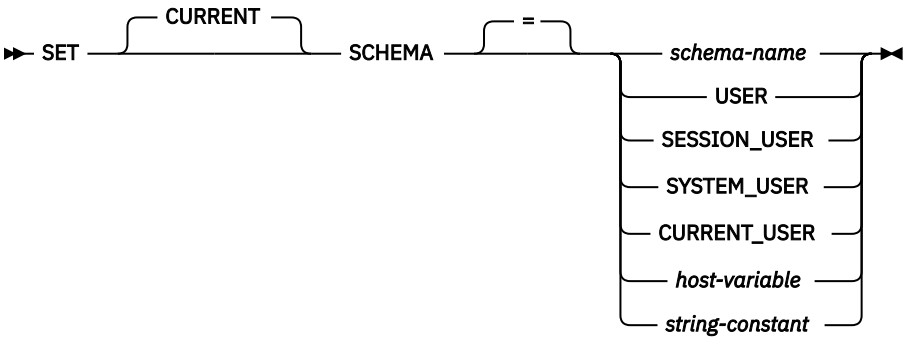
Invocation

The statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

schema-name
This one-part name identifies a schema that exists at the application server. The length must not exceed 128 bytes (SQLSTATE 42815). No validation that the schema exists is made at the time that the schema is set. If a *schema-name* is misspelled, the error will not be caught, and that could affect the way that subsequent SQL statements execute.

USER
The value in the USER special register.

SESSION_USER
The value in the SESSION_USER special register.

SYSTEM_USER
The value in the SYSTEM_USER special register.

CURRENT_USER
The value in the CURRENT_USER special register.

host-variable
A variable of type CHAR or VARCHAR. The length of the contents of the *host-variable* must not exceed 128 bytes (SQLSTATE 42815). It cannot be set to null. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 42815).

The characters of the *host-variable* must be left-aligned. When specifying the *schema-name* with a *host-variable*, all characters must be specified in the exact case intended as there is no conversion to uppercase characters.

string-constant
A character string constant with a maximum length of 128 bytes.

Rules

- If the value specified does not conform to the rules for a *schema-name*, an error (SQLSTATE 3F000) is raised.
- The value of the CURRENT SCHEMA special register is used as the schema name in all dynamic SQL statements, with the exception of the CREATE SCHEMA statement, where an unqualified reference to a database object exists.
- The QUALIFIER bind option specifies the schema name for use as the qualifier for unqualified database object names in static SQL statements.

Notes

- The initial value of the CURRENT SCHEMA special register is equivalent to USER.
- Setting the CURRENT SCHEMA special register does not affect the CURRENT PATH special register. Hence, the CURRENT SCHEMA will not be included in the SQL path and functions, procedures and user-defined type resolution may not find these objects. To include the current schema value in the SQL path, whenever the SET SCHEMA statement is issued, also issue the SET PATH statement including the schema name from the SET SCHEMA statement.
- CURRENT SQLID is accepted as a synonym for CURRENT SCHEMA and the effect of a SET CURRENT SQLID statement will be identical to that of a SET CURRENT SCHEMA statement. No other effects, such as statement authorization changes, will occur.

Examples

- *Example 1:* The following statement sets the CURRENT SCHEMA special register.

```
SET SCHEMA RICK
```

- *Example 2:* The following example retrieves the current value of the CURRENT SCHEMA special register into the host variable called CURSCHEMA.
-

```
EXEC SQL VALUES (CURRENT SCHEMA) INTO :CURSCHEMA;
```

The value would be RICK, set by the previous example.

SET SERVER OPTION

The SET SERVER OPTION statement specifies a server option setting that is to remain in effect while a user or application is connected to the federated database. When the connection ends, this server option's previous setting is reinstated.

This statement is not under transaction control.

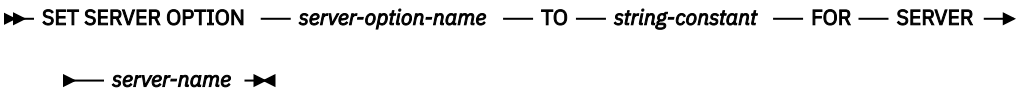
Invocation

This statement can be issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax



Description

server-option-name
Names the server option that is to be set.

TO *string-constant*
Specifies the setting for *server-option-name* as a character string constant.

SERVER *server-name*
Names the data source to which *server-option-name* applies. It must be a server described in the catalog.

Notes

- Server option names can be entered in uppercase or lowercase.
- One or more SET SERVER OPTION statements can be submitted when a user or application connects to the federated database. The statement (or statements) must be specified at the start of the first unit of work that is processed after the connection is established.
- SYSCAT.SERVEROPTIONS will not be updated based on a SET SERVER OPTION statement, because this change only affects the current connection.
- For static SQL, using the SET SERVER OPTION statement affects only the execution of the static SQL statement. Using the SET SERVER OPTION statement has no effect on the plans that are generated by the optimizer.

Examples

- *Example 1:* An Oracle data source called ORASERV is defined to a federated database called DJDB. ORASERV is configured to disallow plan hints. However, the DBA would like plan hints to be enabled for a test run of a new application. When the run is over, plan hints will be disallowed again.

```
CONNECT TO DJDB;
strcpy(stmt,"set server option plan_hints to 'Y' for server oraserv");
EXEC SQL EXECUTE IMMEDIATE :stmt;
strcpy(stmt,"select c1 from ora_t1 where c1 > 100"); /*Generate plan hints*/
EXEC SQL PREPARE s1 FROM :stmt;
EXEC SQL DECLARE c1 CURSOR FOR s1;
```

```
EXEC SQL OPEN c1;
EXEC SQL FETCH c1 INTO :hv;
```

- *Example 2:* You have set the server option PASSWORD to 'Y' (validating passwords at the data source) for all Oracle 8 data sources. However, for a particular session in which an application is connected to the federated database in order to access a specific Oracle 8 data source-one defined to the federated database DJDB as ORA8A-passwords will not need to be validated.

```
CONNECT TO DJDB;
strcpy(stmt,"set server option password to 'N' for server ora8a");
EXEC SQL PREPARE STMT_NAME FROM :stmt;
EXEC SQL EXECUTE STMT_NAME FROM :stmt;
strcpy(stmt,"select max(c1) from ora8a_t1");
EXEC SQL PREPARE STMT_NAME FROM :stmt;
EXEC SQL DECLARE c1 CURSOR FOR STMT_NAME;
EXEC SQL OPEN c1; /*Does not validate password at ora8a*/
EXEC SQL FETCH c1 INTO :hv;
```

SET SESSION AUTHORIZATION

The SET SESSION AUTHORIZATION statement changes the value of the SESSION_USER special register.

The statement is not under transaction control. The SET SESSION AUTHORIZATION statement is intended to provide support for a single user assuming different authorization IDs on the same connection, and should not be used for scenarios in which different users reuse the same connection, commonly referred to as connection pooling.

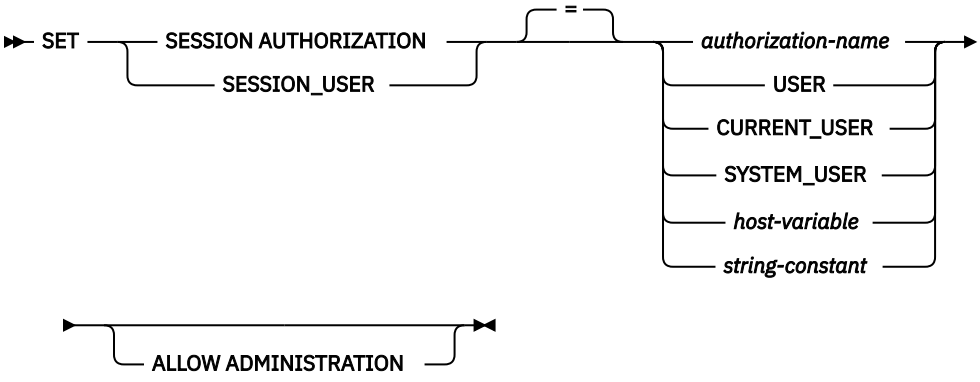
Invocation

The statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include SETSESSIONUSER on the authorization ID value to which the special register is being set.

Syntax



Description

authorization-name
Specifies the authorization ID that is to be used as the new value for the SESSION_USER special register.

USER
The value in the USER special register.

CURRENT_USER
The value in the CURRENT USER special register.

SYSTEM_USER
The value in the SYSTEM_USER special register.

host-variable
A variable of type CHAR or VARCHAR. The length of the contents of *host-variable* must not exceed 128 bytes (SQLSTATE 28000). It cannot be set to null. If *host-variable* has an associated indicator variable, the value of that indicator variable must not indicate a null value (SQLSTATE 28000).

The characters of *host-variable* must be left-aligned. When specifying *authorization-name* with a host variable, all characters must be specified in uppercase, because there is no conversion to uppercase characters.

string-constant
A character string constant with a maximum length of 128 bytes.

ALLOW ADMINISTRATION
Specifies that SQL schema statements can be specified before this statement in the same unit of work.

Rules

- The value specified for the SESSION_USER special register must conform to the rules for an authorization ID of type USER (SQLSTATE 42602).
- The OWNER bind option specifies the authorization ID that is to be used for static SQL statements.
- This statement can only be issued as the first statement (other than a SET special register statement) in a new unit of work without any open WITH HOLD cursors (SQLSTATE 25001). This restriction includes any PREPARE request for a statement other than a SET special register statement.
- The value of the SESSION_USER special register is used as the authorization ID for all dynamic SQL statements in a package bound with the DYNAMICRULES(RUN) bind option. (This includes INVOKERUN and DEFINERUN when the package is not used by a routine). If a package is using owner, invoker, or definer authorization based on the DYNAMICRULES option, this statement has no effect on dynamic SQL statements issued from within that package.

Notes

- The SET SESSION AUTHORIZATION statement lets you change the session authorization ID. The session authorization ID represents the current user of the connection and is the authorization ID that the database manager considers for all authorization checking relative to dynamic SQL within a DYNAMICRULES run package. The SESSION_USER special register can be used to see the current value of this session authorization ID.
- The initial value of the SESSION_USER special register for a new connection is the same as the value of the SYSTEM_USER special register.
- The group information for the session authorization ID specified in this statement is acquired at the time of statement execution.
- Setting the SESSION_USER special register does not effect either the CURRENT SCHEMA or the CURRENT PATH special register.
- If any error occurs during the setting of the SESSION_USER special register, the register reverts to its previous value.
- This statement should not be used to allow multiple, different users to reuse the same connection, because each user will inherit the ability to change the value of the SESSION_USER special register that the original connection owner had. This statement is dependent upon the value of SYSTEM_USER for privileges checking, and the initial connection authorization ID is not changed by the SET SESSION AUTHORIZATION statement. Moreover, the following behaviors impacting connection reuse are not addressed by this statement:

  - The CONNECT privilege is not checked for the new authorization ID
  - The content of any updatable special register is not reset; in particular, the content of the ENCRYPTION PASSWORD special register is not modified and is available to the new authorization ID for encryption or decryption
  - The content of any declared global temporary table is not affected, and is accessible to the new authorization ID
  - Any existing links to remote servers are not reset
- If the ALLOW ADMINISTRATION clause is specified, the following types of statements or operations can precede the SET SESSION AUTHORIZATION statement:
  - Data definition language (DDL), including the definition of savepoints and the declaration of global temporary tables, but not including SET INTEGRITY
  - GRANT and REVOKE statements
  - LOCK TABLE statement
  - COMMIT and ROLLBACK statements
  - SET of special registers
  - SET of global variables

Examples

- *Example 1:* The following statement sets the SESSION_USER special register.

```
SET SESSION_USER = RAJIV
```

- *Example 2:* Set the session authorization ID (the SESSION_USER special register) to be the value of the system authorization ID, which is the ID that established the connection on which the statement has been issued.

```
SET SESSION AUTHORIZATION SYSTEM_USER
```

SET USAGE LIST STATE

The SET USAGE LIST STATE statement manages the state of a usage list and the associated data and memory.

This statement is not under transaction control.

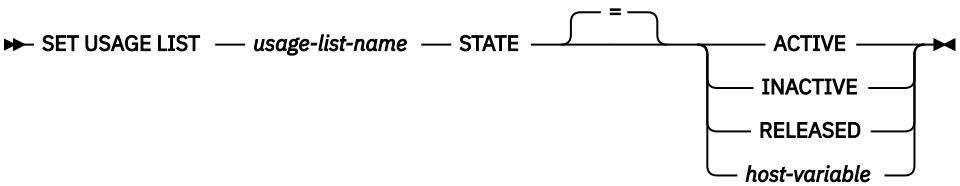
Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include DBADM or SQLADM authority.

Syntax



Description

usage-list-name
Identifies the usage list. The *usage-list-name*, including the implicit or explicit qualifier, must identify a usage list that is described in the catalog (SQLSTATE 42704).

ACTIVE
Indicates that the usage list is activated for monitoring. Memory for the usage list is allocated when the table or index is first referenced by a section. If the usage list is for a partitioned table or index then the memory is allocated when the data partition is first referenced by a section. In a partitioned database environment or Db2 pureScale environment, memory is allocated at each member. If the usage list is already in the ACTIVE state then a warning is returned (SQLSTATE 01598).

On activation, the data in the usage list is removed and collection starts from the beginning of the list.

INACTIVE
Indicates that the usage list is deactivated for monitoring. If the state of a usage list is already set to INACTIVE then this keyword is ignored. If the state of the usage list for a partitioned table or index is set to INACTIVE, then the state of the usage list for each data partition is set to INACTIVE. If the state is already INACTIVE then this keyword is ignored. Similarly, in a partitioned database environment or Db2 pureScale environment, the state of the usage list for each member is set to INACTIVE. If the state is already INACTIVE then this keyword is ignored.

Data collected in the list is not removed when the state of the usage list is set to INACTIVE.

RELEASED
Indicates that the memory associated with a usage list is released. If the state of the usage list for a partitioned table or index is set to RELEASED, then the memory associated with each data partition is released. In a partitioned database environment or Db2 pureScale environment, the memory associated with each member is released.

Notes

- ***Determining current state***: The current state of a usage list is determined by using the MON_GET_USAGE_LIST_STATUS built-in function.
- ***Considerations for Db2 pureScale or partitioned database environments***: If a usage list for a partitioned table or index is activated, memory is allocated for each data partition. Similarly, in a partitioned database environment or Db2 pureScale environment, memory is allocated at each active member.
- ***Memory allocation for unavailable members***: If a member is unavailable at the time of activation, then the memory associated with the usage list for this member is allocated when the member is next activated (if the state of the usage list is still active). This also applies when a member is added to the cluster.
- ***Memory allocation for data partitions that are being added or attached***: For data partitions that are being added or attached, the memory associated with the usage list for this newly added or attached data partition is allocated when the next section that references the partitioned table or index is executed.
- ***Setting INACTIVE independently***: If the usage list was created with the property, WHEN FULL DEACTIVATE, then the state of the usage list for each data partition or member is set to INACTIVE independently.
- ***Implicit reactivation of an active usage list***: If the state of an INACTIVE ON START DATABASE usage list is set to ACTIVE in a partitioned database environment or Db2 pureScale environment, then its behavior is similar to ACTIVE ON START DATABASE until the usage list is explicitly deactivated or the instance is recycled. That is, if state of the usage list is active when a database member is deactivated or offline, and that database member is subsequently reactivated, the usage list for this member is implicitly reactivated.
- ***Definition of released state***: A usage list is considered to be in the released state if it is defined and has not been activated (explicitly or automatically) or has been released using the SET USAGE LIST STATE

statement. Usage lists in the state released are not returned by the MON_GET_USAGE_LIST_STATUS table function.

- ***Activation pending, active, and failed states***: If a usage list is activated (explicitly or automatically) then the state of the usage list is set to activation pending and the memory is allocated when the table or index is first referenced by the section. At this point the state of the usage list is set to active. If the memory for the usage list cannot be allocated, then the state of the usage list is set to failed and it must be explicitly activated using the SET USAGE LIST STATE statement.
- ***Inactive usage lists remain inactive upon database member reactivation***: If the state of an ACTIVE ON START DATABASE usage list is set to INACTIVE in a partitioned database environment or Db2 pureScale environment, then its behavior is similar to INACTIVE ON START DATABASE until the usage list is explicitly activated or the instance is recycled. That is, if the state of a usage list is inactive when a database member is deactivated or offline, and that database member is subsequently reactivated, the state of the usage list for this member will remain inactive.
- ***Activating, deactivating, or releasing a usage list for a partitioned table or index***: If a usage list for a partitioned table or index is activated, deactivated, or released then the state change applies to each data partition. Similarly, in a partitioned database environment or Db2 pureScale environment, the state change applies to each member.
- ***Usage list size considerations***: When activated, the memory associated with the usage list is allocated from the monitor heap. At the maximum list size setting, the usage list is approximately 2MB. For partitioned tables or indexes, memory is allocated for each data partition. For example, if a partitioned table has three data partitions defined, the total memory allocated is approximately 6MB. Therefore, activating multiple usage lists imposes more memory requirements on the monitor heap. It is therefore suggested that a reasonable list size is selected or that you set the **mon_heap_sz** configuration parameter to AUTOMATIC so that the database manager manages the monitor heap size.
- ***Data collection when a usage list is set to INACTIVE***: Data collected in the list is not removed when the state of the usage list is set to INACTIVE.
- ***Data access and memory***: The data in the list is still accessible (using MON_GET_TABLE_USAGE_LIST and MON_GET_INDEX_USAGE_LIST table functions) provided that the memory for the list is allocated.
- ***Releasing memory***: The memory associated with the usage list is released when one of the following events occurs:
  - The usage list is dropped.
  - The table or index on which the usage list is defined is dropped. The memory that is associated with the usage is released for all data partitions. In a partitioned database environment or Db2 pureScale environment, the memory that is associated with the usage list is released for all active members.
  - When a data partition is detached from a partitioned table or index. Only the memory associated with the data partition is released.
  - When a database member is deactivated. Only the memory associated with the member is released.
  - When the entire instance or database is deactivated. Usage list data does not persist when the database is deactivated and restarted.
  - When memory associated with the usage list is explicitly released using the SET USAGE LIST STATE statement.

SET variable

The SET variable statement assigns values to variables.

This statement is not under transaction control.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

To reference a transition variable, the privileges held by the authorization ID of the trigger creator must include at least one of the following authorities:

- UPDATE privilege on any columns referenced on the left side of the assignment or UPDATEIN privilege on the schema containing the tables having the columns referenced on the left side of the assignment
- SELECT privilege on any columns referenced on the right side or SELECTIN privilege on the schema containing the tables having the columns referenced on the right side
- CONTROL privilege on the table (subject table of the trigger)
- DATAACCESS authority on the schema containing the table
- DATAACCESS authority

If a global variable is referenced in the right side of the assignment statement, the privileges held by the authorization ID of the statement must include one of the following authorities:

- READ privilege on the global variable that is not defined in a module
- EXECUTE privilege on the module of the global variable that is defined in a module
- EXECUTEIN privilege on the schema containing the module of the global variable that is defined in a module
- DATAACCESS authority on the schema containing the module of the global variable that is defined in a module

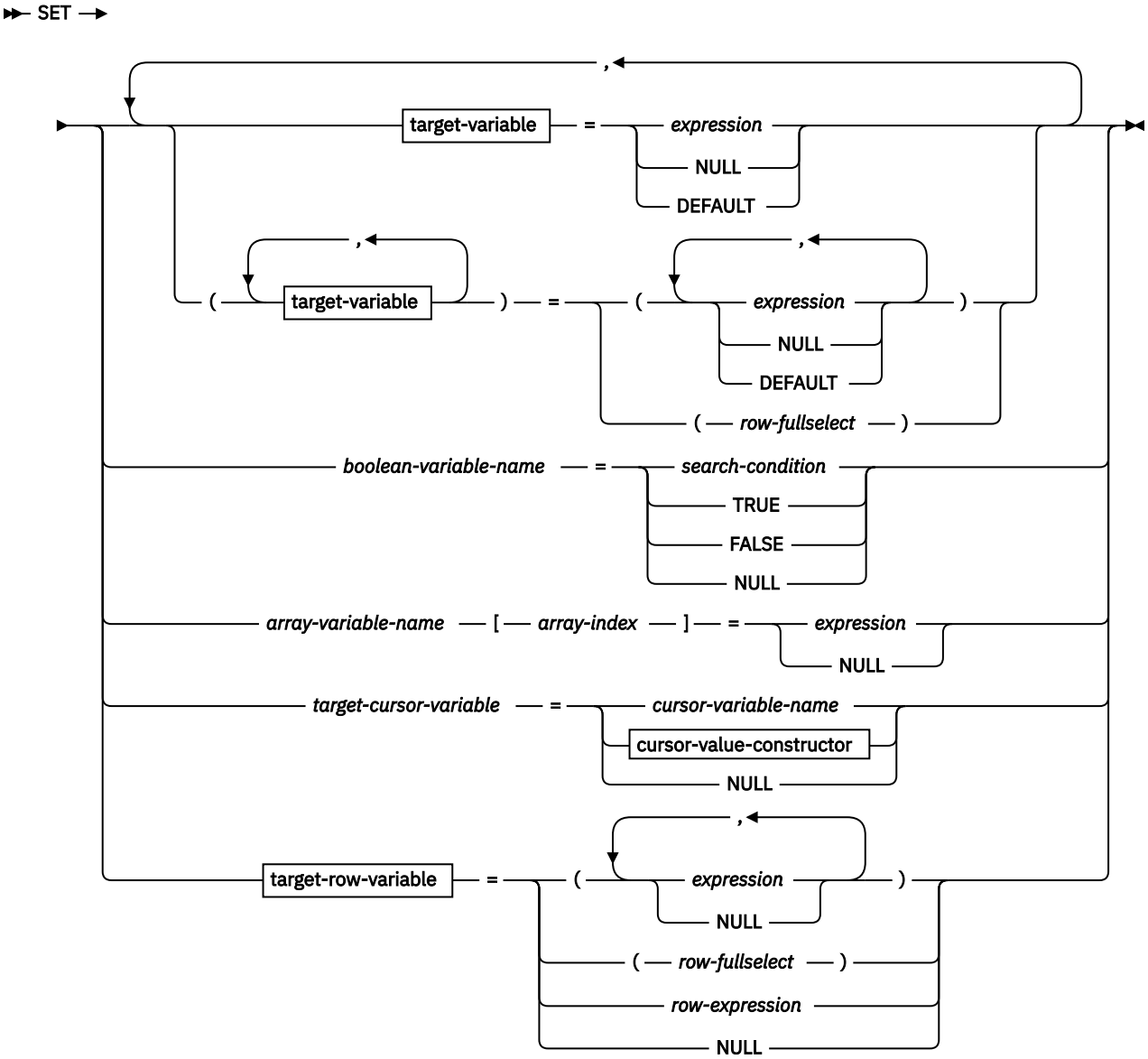
If a global variable is assigned a value in the left side of the assignment statement, the privileges held by the authorization ID of the statement must include one of the following authorities:

- WRITE privilege on the global variable that is not defined in a module
- EXECUTE privilege on the module of the global variable that is defined in a module
- EXECUTEIN privilege on the schema containing the module of the global variable that is defined in a module
- DATAACCESS authority on the schema containing the module of the global variable that is defined in a module

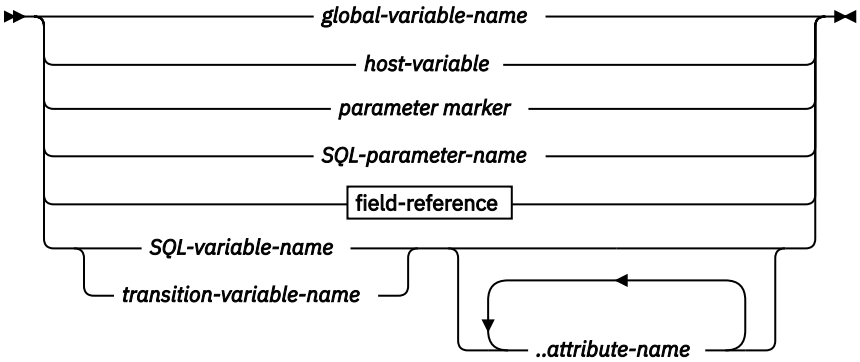
To execute this statement with a *row-fullselect* as the right side of the assignment, the privileges held by the authorization ID of the statement must include the privileges necessary to execute the *row-fullselect*. See the Authorization section in "SQL queries".

To execute this statement with a *cursor-value-constructor* that uses a *select-statement*, the privileges held by the authorization ID of the statement must include the privileges necessary to execute the *select-statement*. See the Authorization section in "SQL queries".

Syntax

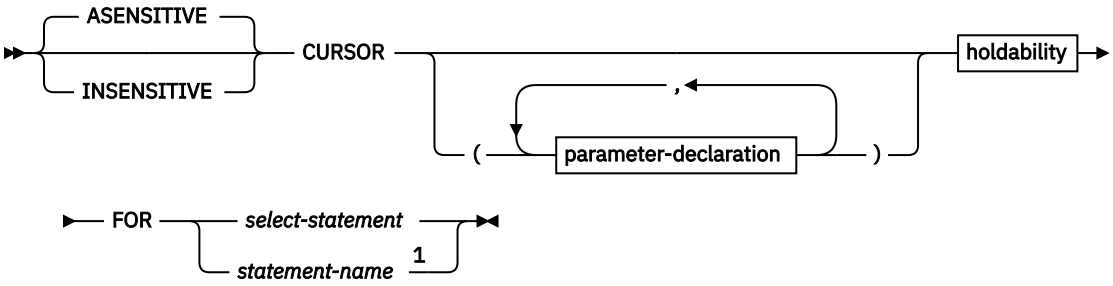


target-variable

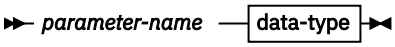


field-reference

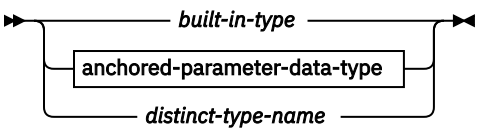
cursor-value-constructor



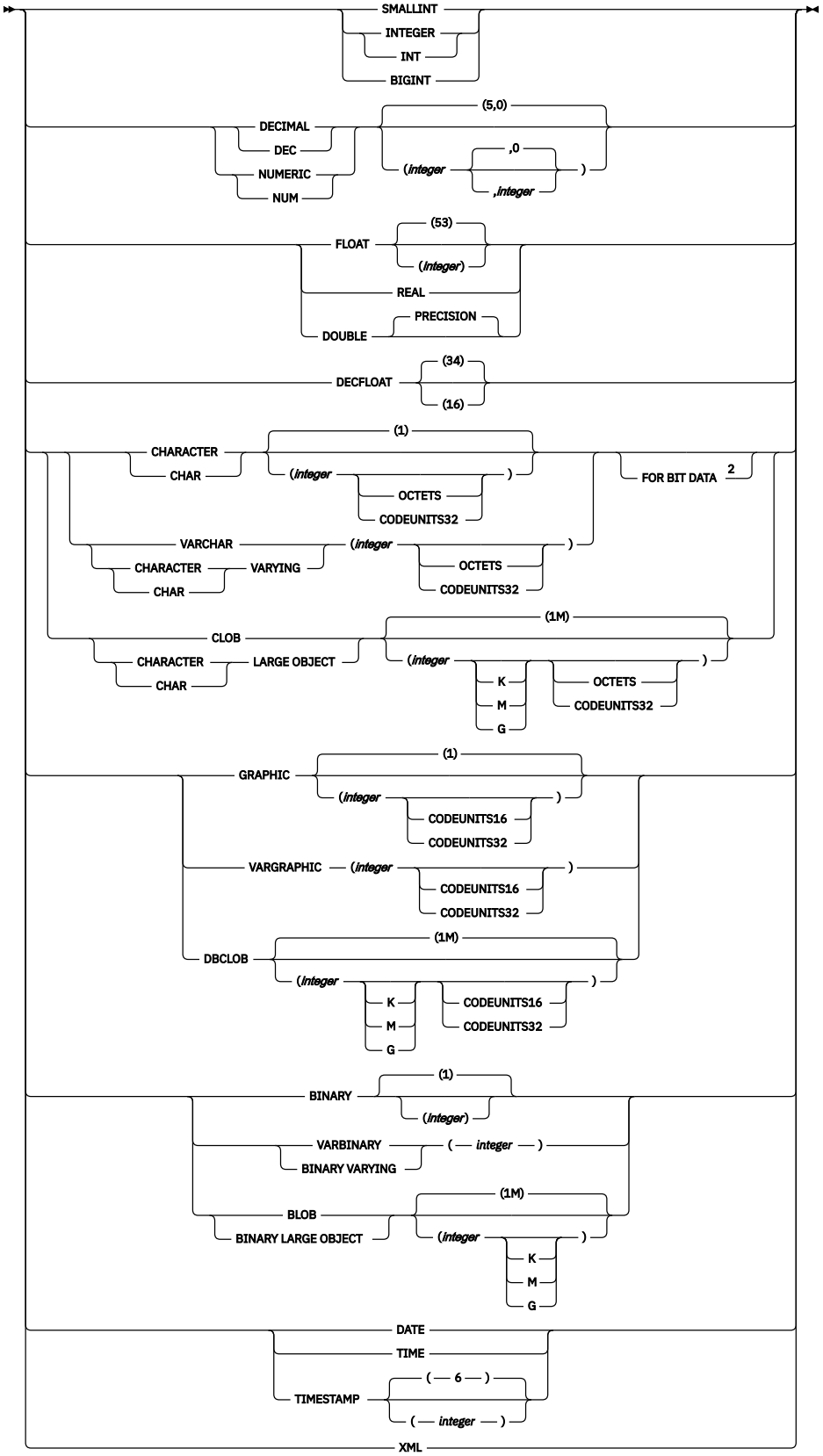
parameter-declaration



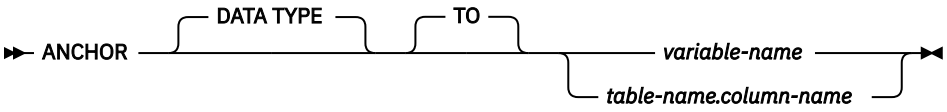
data-type



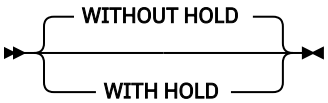
built-in-type



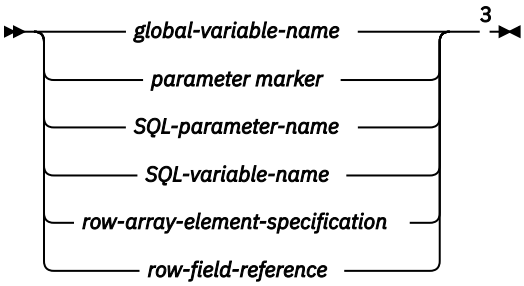
anchored-parameter-data-type



holdability



target-row-variable



Notes:

[1] *statement-name* cannot be specified if *parameter-declaration* is specified.

[2] The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

[3] The data type must be a row type.

Description

target-variable

Identifies the target variable of the assignment. A *target-variable* representing the same variable must not be specified more than once (SQLSTATE 42701).

global-variable-name

Identifies the global variable that is the assignment target. The *global-variable-name* must identify a global variable that exists at the current server (SQLSTATE 42704).

host-variable

Identifies the host variable that is the assignment target.

parameter-marker

Identifies the parameter marker that is the assignment target.

SQL-parameter-name

Identifies the parameter that is the assignment target. The parameter must be specified in *parameter-declaration* in the CREATE PROCEDURE statement.

field-reference

Identifies the field within a row type value that is the assignment target.

row-variable-name

The name of a variable with a data type that is a row type.

field-name

The name of a field within the row type.

SQL-variable-name

Identifies the SQL variable that is the assignment target. SQL variables must be declared before they are used.

transition-variable-name
Identifies the column to be updated in the transition row. A *transition-variable-name* must identify a column in the subject table of a trigger, optionally qualified by a correlation name that identifies the new value (SQLSTATE 42703).

..attribute-name
Specifies the attribute of a structured type that is set (referred to as an *attribute assignment*). The *SQL-variable-name* or *transition-variable-name* specified must be defined with a user-defined structured type (SQLSTATE 428DP). The *..attribute-name* must be an attribute of the structured type (SQLSTATE 42703). An assignment that does not involve the *..attribute-name* clause is referred to as a *conventional assignment*.

expression
Indicates the new value of the target of the assignment. The expression is any expression of the type described in "Expressions". The expression cannot include an aggregate function except when it occurs within a scalar fullselect (SQLSTATE 42903). In the context of a CREATE TRIGGER statement, an *expression* can contain references to OLD and NEW transition variables. The transition variables must be qualified by the *correlation-name* (SQLSTATE 42702).

NULL
Specifies the null value. If the target of the assignment is a row variable, each field is assigned the null value. NULL cannot be the value in an attribute assignment unless it was specifically cast to the data type of the attribute (SQLSTATE 429B9).

DEFAULT
Specifies that the default value should be used.

In SQL procedures, the DEFAULT clause can be specified only for static SQL statements. The exception is that the DEFAULT clause can be specified when *target-variable* is a global variable in a dynamic SQL statement.

If *target-variable* is a column, the value inserted depends on how the column was defined in the table.

- If the column was defined using the WITH DEFAULT clause, the value is set to the default defined for the column (see *default-clause* in "ALTER TABLE").
- If the column was defined using the IDENTITY clause, the value is generated by the database manager.
- If the column was defined without specifying the WITH DEFAULT clause, the IDENTITY clause, or the NOT NULL clause, the value is NULL.
- If the column was defined using the NOT NULL clause and:
  - The IDENTITY clause is not used or
  - The WITH DEFAULT clause was not used or
  - DEFAULT NULL was used

  the DEFAULT keyword cannot be specified for that column (SQLSTATE 23502).

If *target-variable* is an SQL variable, the value inserted is the default, as specified or implied in the variable declaration.

If *target-variable* is a global variable, the value inserted is the default, as specified in the variable creation.

If *target-variable* is an SQL variable or an SQL parameter in an SQL procedure, a host variable, or a parameter marker, the DEFAULT keyword cannot be specified (SQLSTATE 42608).

row-fullselect
A fullselect that returns a single row with the number of columns corresponding to the number of target variables or fields in the row variable specified for assignment. The values are assigned to each corresponding target variable or field. If the result of the row fullselect is no rows, null values are assigned to the target variables in the list or, in an assignment to a row variable, a single null is assigned. In the context of a CREATE TRIGGER statement, a *row-fullselect* can contain references to OLD and NEW transition variables, which must be qualified by their *correlation-name* to specify which

transition variable is to be used (SQLSTATE 42702). An error is returned if there is more than one row in the result (SQLSTATE 21000).

boolean-variable-name
Identifies an SQL variable or parameter or a global variable. The variable or parameter must be of Boolean type (SQLSTATE 428H0). The SET statement must be issued within a compound SQL (compiled) statement (SQLSTATE 428H2).

search-condition
A search condition whose result is true, false, or unknown. A result of unknown is returned as the Boolean value NULL.

TRUE
Specifies the Boolean value TRUE.

FALSE
Specifies the Boolean value FALSE.

NULL
Specifies the Boolean value NULL.

array-variable-name
Identifies an SQL variable, SQL parameter, or global variable of an array type (SQLSTATE 428H0).

[*array-index*]
An expression that specifies which element in the array will be the target of the assignment. For an ordinary array, the array-index must be assignable to INTEGER (SQLSTATE 22018 or 428H1). Its value must be between 1 and the maximum cardinality defined for the array and cannot be the null value (SQLSTATE 2202E).

For an associative array, the array index expression must be assignable to the index data type of the associative array (SQLSTATE 22018 or 428H1) and cannot be the null value (SQLSTATE 2202E).

target-cursor-variable
Identifies a cursor variable. The data type of *target-cursor-variable* must be a cursor type (SQLSTATE 42821).

cursor-variable-name
Identifies a cursor variable of the same cursor type as *target-cursor-variable*.

cursor-value-constructor
A *cursor-value-constructor* specifies the *select-statement* that is associated with the target variable. The assignment of a *cursor-value-constructor* to a cursor variable defines the underlying cursor of that cursor variable.

ASENSITIVE or INSENSITIVE
Specifies whether the cursor is asensitive or insensitive to changes. See "DECLARE CURSOR" for more information. The default is ASENSITIVE.

ASENSITIVE
Specifies that the cursor should be as sensitive as possible to inserts, updates, or deletes made to the rows underlying the result table, depending on how the *select-statement* is optimized. ASENSITIVE is the default.

INSENSITIVE
Specifies that the cursor does not have sensitivity to inserts, updates, or deletes that are made to the rows underlying the result table. If INSENSITIVE is specified, the cursor is read-only and the result table is materialized when the cursor is opened. As a result, the size of the result table, the order of the rows, and the values for each row do not change after the cursor is opened. The SELECT statement cannot contain a FOR UPDATE clause, and the cursor cannot be used for positioned updates or deletes.

(parameter-declaration, ...)
Specifies the input parameters of the cursor, including the name and the data type of each parameter. Named input parameters can be specified only if *select-statement* is also specified in *cursor-value-constructor* (SQLSTATE 428HU).

parameter-name
Names the cursor parameter for use as an SQL variable within *select-statement*. The name cannot be the same as any other parameter name for the cursor. Names should also be chosen to avoid any column names that could be used in *select-statement*, since column names are resolved before parameter names.

data-type
Specifies the data type of the cursor parameter used within *select-statement*. Structured types, and reference types cannot be specified (SQLSTATE 429BB).

built-in-type
Specifies a built-in data type. For a more complete description of each built-in data type, see "CREATE TABLE".

anchored-parameter-data-type
Identifies another object used to determine the data type of the cursor parameter. The data type of the anchor object is bound by the same limitations that apply when specifying the data type directly.

ANCHOR DATA TYPE TO
Indicates an anchored data type is used to specify the data type.

variable-name
Identifies a local SQL variable, an SQL parameter, or a global variable. The data type of the referenced variable is used as the data type for the cursor parameter.

table-name.column-name
Identifies a column name of an existing table or view. The data type of the column is used as the data type for the cursor parameter.

distinct-type-name
Specifies the name of a distinct type. If *distinct-type-name* is specified without a schema name, the distinct type is resolved by searching the schemas in the SQL path.

holdability
Specifies whether the cursor is prevented from being closed as a consequence of a commit operation. See "DECLARE CURSOR" for more information. The default is WITHOUT HOLD.

WITHOUT HOLD
Does not prevent the cursor from being closed as a consequence of a commit operation.

WITH HOLD
Maintains resources across multiple units of work. Prevents the cursor from being closed as a consequence of a commit operation.

select-statement
Specifies the SELECT statement of the cursor. See "select-statement" for more information. If *parameter-declaration* is included in *cursor-value-constructor*, then *select-statement* must not include any local SQL variables or routine SQL parameters (SQLSTATE 42704).

statement-name
Specifies the prepared *select-statement* of the cursor. See "PREPARE" for an explanation of prepared statements. The target cursor variable must not have a data type that is a strongly-typed user-defined cursor type (SQLSTATE 428HU). Named input parameters must not be specified in *cursor-value-constructor* if *statement-name* is specified (SQLSTATE 428HU).

target-row-variable
Identifies the target row variable of the assignment. The data type must be of a row type.

row-expression
Specifies the new row value for the target of the assignment. It can be any row expression of the type described in "Row expression". The number of fields in the row must match the target of the assignment and each field in the row must be assignable to the corresponding field in the target of the assignment. If the source and the target values are a user-defined row type, the type names must be the same (SQLSTATE 42821).

Rules

- The number of values to be assigned from expressions, NULLs, DEFAULTs, or the *row-fullselect* must match the number of *target-variables* specified for assignment (SQLSTATE 42802).
- A SET variable statement cannot assign an SQL variable and a transition variable in one statement (SQLSTATE 42997).
- Global variables cannot be assigned inside triggers that are not defined using a compound SQL (compiled) statement, functions that are not defined using a compound SQL (compiled) statement, methods, or compound SQL (inlined) statements (SQLSTATE 428GX).
- If the value being assigned is an array resulting from an array constructor or from ARRAY_AGG, the base types of the array and of the target variable must be identical (SQLSTATE 42821).
- ***Use of anchored data types:*** An anchored data type cannot refer to the following objects (SQLSTATE 428HS): a nickname, typed table, typed view, statistical view that is associated with an expression-based index, declared temporary table, row definition that is associated with a weakly typed cursor, object with a code page or collation that is different from the database code page or database collation.
- ***Assignments involving cursor variables:*** Assignments that reference a cursor variable that set it to the value of a cursor value constructor can only be used in compound SQL (compiled) statements. Any OPEN statement using a cursor variable must occur within the same scope as the assignment (SQLSTATE 51044).

Notes

- Values are assigned to target variables according to specific assignment rules.
- ***Assignment statement in SQL procedures:*** Assignment statements in SQL procedures must conform to the SQL assignment rules. String assignments use retrieval assignment rules.
- ***Assignments of array elements:*** If the assignment is of the form `SET A[idx] = rhs`, where A is an array variable name, `idx` is an expression used as the array-index, and `rhs` is an expression of the same type as the array element, then:

  1. If array A is the null value, set A to the empty array.
  2. Let C be the cardinality of array A.
  3. If A is an ordinary array:
     - If `idx` is less than or equal to C, the value in the position identified by `idx` is replaced by the value of `rhs`.
     - If `idx` is greater than C, then:
       - The value in position *i*, for *i* greater than C and less than `idx`, is set to the null value.
       - The value in position `idx` is set to the value of `rhs`.
       - The cardinality of A is set to `idx`.
  4. If A is an associative array:
     - If `idx` matches an existing array index value, the element value with array index `idx` is replaced by the value of `rhs`.
     - If `idx` does not match any existing array index value, then:
       - The cardinality of A is incremented by `1`
       - The new element value is set to `rhs` with associated array index value `idx`.
  5. If `idx` is less than or equal to C, the value in the position identified by `idx` is replaced by the value of `rhs`.
  6. If `idx` is greater than C, then:
     a. The value in position *i*, for *i* greater than C and less than `idx`, is set to the null value.
     b. The value in position `idx` is set to the value of `rhs`.
     c. The cardinality of A is set to `idx`.

- If a variable has been declared with an identifier that matches the name of a special register (such as PATH), the variable must be delimited to prevent unintentional assignment to the special register (for example, `SET "PATH" = 1;` for a variable called PATH that has been declared as an integer).
- If more than one assignment is included, each *expression* and *row-fullselect* is evaluated before the assignments are performed. Thus, references to target variables in an expression or row fullselect are always the value of the target variable before any assignment in the single SET statement.
- When an identity column defined as a distinct type is updated, the entire computation is done in the source type, and the result is cast to the distinct type before the value is actually assigned to the column. (There is no casting of the previous value to the source type before the computation.)
- To have the database manager generate a value on a SET statement for an identity column, use the DEFAULT keyword:

```
SET NEW.EMPNO = DEFAULT
```

  In this example, NEW.EMPNO is defined as an identity column, and the value used to update this column is generated by the database manager.
- For more information about consuming values of a generated sequence for an identity column, and for information about exceeding the maximum value for an identity column, see "INSERT".

Examples

- *Example 1:* Set the salary column of the row for which the trigger action is currently executing to 50000.

```
SET NEW_VAR.SALARY = 50000;
```

  Or:

```
SET (NEW_VAR.SALARY) = (50000);
```

- *Example 2:* Set the salary and the commission column of the row for which the trigger action is currently executing to 50000 and 8000, respectively.

```
SET NEW_VAR.SALARY = 50000, NEW_VAR.COMM = 8000;
```

  Or:

```
SET (NEW_VAR.SALARY, NEW_VAR.COMM) = (50000, 8000);
```

- *Example 3:* Set the salary and the commission column of the row for which the trigger action is currently executing to the average salary and commission of employees in the department that is associated with the updated row.

```
SET (NEW_VAR.SALARY, NEW_VAR.COMM)
   = (SELECT AVG(SALARY), AVG(COMM)
        FROM EMPLOYEE E
        WHERE E.WORKDEPT = NEW_VAR.WORKDEPT);
```

- *Example 4:* Set the salary and the commission column of the row for which the trigger action is currently executing to 10000 and the original value of salary (that is, before the SET statement was executed), respectively.

```
SET NEW_VAR.SALARY = 10000, NEW_VAR.COMM = NEW_VAR.SALARY;
```

  Or:

```
SET (NEW_VAR.SALARY, NEW_VAR.COMM) = (10000, NEW_VAR.SALARY);
```

- *Example 5:* Increase the SQL variable P_SALARY by 10 percent.

```
SET P_SALARY = P_SALARY + (P_SALARY * .10)
```

- *Example 6:* Set the SQL variable P_SALARY to the null value.

```
SET P_SALARY = NULL
```

- *Example 7:* Assign numbers 2.71828183 and 3.1415926 to the first and tenth elements of the array variable SPECIALNUMBERS. After the first assignment, the cardinality of P_PHONENUMBERS is 1. After the second assignment, the cardinality is 10, and elements 2 to 9 have been implicitly assigned the null value.

```
SET SPECIALNUMBERS[1] = 2.71828183;

SET SPECIALNUMBERS[10] = 3.14159265;
```

- *Example 8:* Given a table named SECURITY.USERS, which has a row for every user that could connect to the database, assign the current time and the authorization level to the global variables USERINFO.GV_CONNECT_TIME and USERINFO.GV_AUTH_LEVEL, respectively.

```
SET USERINFO.GV_CONNECT_TIME = CURRENT TIMESTAMP,
  USERINFO.GV_AUTH_LEVEL = (
    SELECT AUTHLEVEL FROM SECURITY.USERS
      WHERE USERID = CURRENT USER)
```

- *Example 9:* Assign values to associative array variable, CAPITALS, which has been declared as the array type CAPITALSARRAY.

```
SET CAPITALS['British Columbia'] = 'Victoria';
SET CAPITALS['Alberta'] = 'Edmonton';
SET CAPITALS['Manitoba'] = 'Winnipeg';
SET CAPITALS['Canada'] = 'Ottawa';
```

  When populating the CAPITALS array, the array indexes are province, territory, and country names specified by strings and the associated array elements are capital cities, also specified by strings.

- *Example 10:* Assign easy to remember names as indexes for personal phone numbers stored in the array variable PHONELIST of array type PERSONAL_PHONENUMBERS.

```
SET PHONELIST['Home'] = '4163053745';
SET PHONELIST['Work'] = '4163053746';
SET PHONELIST['Mom']  = '4164789683';
```

SIGNAL

The SIGNAL statement is used to signal an error or warning condition. It causes an error or warning to be returned with the specified SQLSTATE, along with optional message text.

Invocation

This statement can be embedded in an:

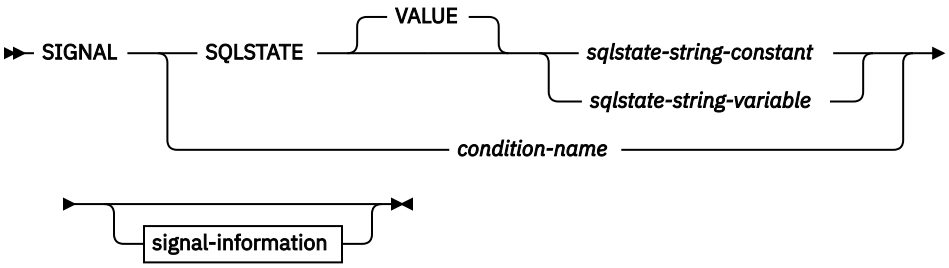
- SQL procedure definition
- Compound SQL (compiled) statement
- Compound SQL (inlined) statement

The compound statements can be embedded in an SQL procedure definition, SQL function definition, or SQL trigger definition. It is not an executable statement and cannot be dynamically prepared.

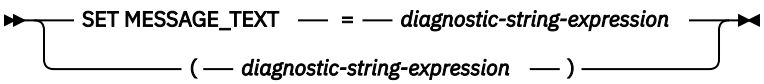
Authorization

If a module condition is referenced, the privileges held by the authorization ID of the statement must include EXECUTE privilege on the module or EXECUTEIN privilege or DATAACCESS authority on the schema containing the module.

Syntax



signal-information



Description

SQLSTATE VALUE

Specifies the SQLSTATE that will be returned. Any valid SQLSTATE value can be used. The specified value must follow the rules for SQLSTATEs:

- Each character must be from the set of digits ("0" through "9") or upper case letters ("A" through "Z") without diacritical marks
- The SQLSTATE class (first two characters) cannot be "00", since this represents successful completion.

In the context of a compound SQL (inlined) statement, a MERGE statement, or as the only statement in a trigger body, the following rules must also be applied:

- The SQLSTATE class (first two characters) cannot be "01" or "02", since these are not error classes.
- If the SQLSTATE class starts with the numbers "0" through "6" or the letters "A" through "H", then the subclass (the last three characters) must start with a letter in the range of "I" through "Z".
- If the SQLSTATE class starts with the numbers "7", "8", "9", or the letters "I" through "Z", then the subclass can be any of "0" through "9" or "A" through "Z".

If the SQLSTATE does not conform to these rules, an error is returned.

sqlstate-string-constant

The *sqlstate-string-constant* must be a character string constant with exactly 5 characters.

sqlstate-string-variable

The specified SQL variable or SQL parameter must be of data type CHAR(5) and must not be the null value.

condition-name

Specifies the name of a condition that will be returned. The condition-name must be declared within the compound-statement or identify a condition that exists at the current server (SQLSTATE 42373).

SET MESSAGE_TEXT =

Specifies a string that describes the error or warning. The string is returned in the SQLERRMC field of the SQLCA. If the actual string is longer than 70 bytes, it is truncated without warning.

diagnostic-string-expression

A literal string, or a local variable or parameter that describes the error condition. If the string is longer than 70 bytes, it is truncated.

(*diagnostic-string-expression*)

An expression of type CHAR or VARCHAR that returns a character string of up to 70 bytes to describe the error condition. If the string is longer than 70 bytes, it is truncated. This option is only provided within the scope of a CREATE TRIGGER statement for compatibility with previous versions of Db2. Regular use of this option is not recommended.

Notes

- If a SIGNAL statement is issued using a *condition-name* that has no associated SQLSTATE value and the condition is not handled, SQLSTATE 45000 is returned and the SQLCODE is set to -438. Note that such a condition will not be handled by a condition handler for SQLSTATE 45000 that is within the scope of the routine issuing the SIGNAL statement.
- If a SIGNAL statement is issued using an SQLSTATE value or a *condition-name* with an associated SQLSTATE value, the SQLCODE returned is based on the SQLSTATE value as follows:
  - If the specified SQLSTATE class is either "01" or "02", a warning or not found condition is returned and the SQLCODE is set to +438.
  - Otherwise, an exception condition is returned and the SQLCODE is set to -438.
- A SIGNAL statement has the indicated fields of the SQLCA set as follows:
  - `sqlerrd` fields are set to zero
  - `sqlwarn` fields are set to blank
  - `sqlerrmc` is set to the first 70 bytes of MESSAGE_TEXT
  - `sqlerrml` is set to the length of `sqlerrmc`, or to zero if no SET MESSAGE_TEXT clause is specified
  - `sqlerrp` is set to ROUTINE
- SQLSTATE values are composed of a two-character class code value, followed by a three-character subclass code value. Class code values represent classes of successful and unsuccessful execution conditions.

  Any valid SQLSTATE value can be used in the SIGNAL statement. However, it is recommended that programmers define new SQLSTATEs based on ranges reserved for applications. This prevents the unintentional use of an SQLSTATE value that might be defined by the database manager in a future release.
  - SQLSTATE classes that begin with the characters "7" through "9", or "I" through "Z" may be defined. Within these classes, any subclass may be defined.
  - SQLSTATE classes that begin with the characters "0" through "6", or "A" through "H" are reserved for the database manager. Within these classes, subclasses that begin with the characters "0" through "H" are reserved for the database manager. Subclasses that begin with the characters "I" through "Z" may be defined.

Example

An SQL procedure for an order system that signals an application error when a customer number is not known to the application. The ORDERS table includes a foreign key to the CUSTOMER table, requiring that the CUSTNO exist before an order can be inserted.

```
CREATE PROCEDURE SUBMIT_ORDER
  (IN ONUM INTEGER, IN CNUM INTEGER,
   IN PNUM INTEGER, IN QNUM INTEGER)
   SPECIFIC SUBMIT_ORDER
   MODIFIES SQL DATA
  LANGUAGE SQL
  BEGIN
    DECLARE EXIT HANDLER FOR SQLSTATE VALUE '23503'
      SIGNAL SQLSTATE '75002'
        SET MESSAGE_TEXT = 'Customer number is not known';
    INSERT INTO ORDERS (ORDERNO, CUSTNO, PARTNO, QUANTITY)
      VALUES (ONUM, CNUM, PNUM, QNUM);
  END
```

TRANSFER OWNERSHIP

The TRANSFER OWNERSHIP statement transfers ownership of a database object.

Invocation

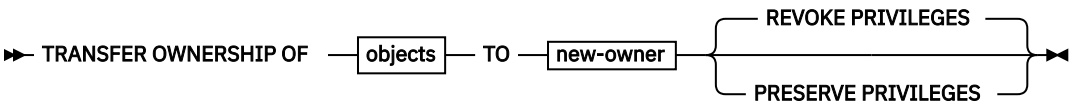
This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

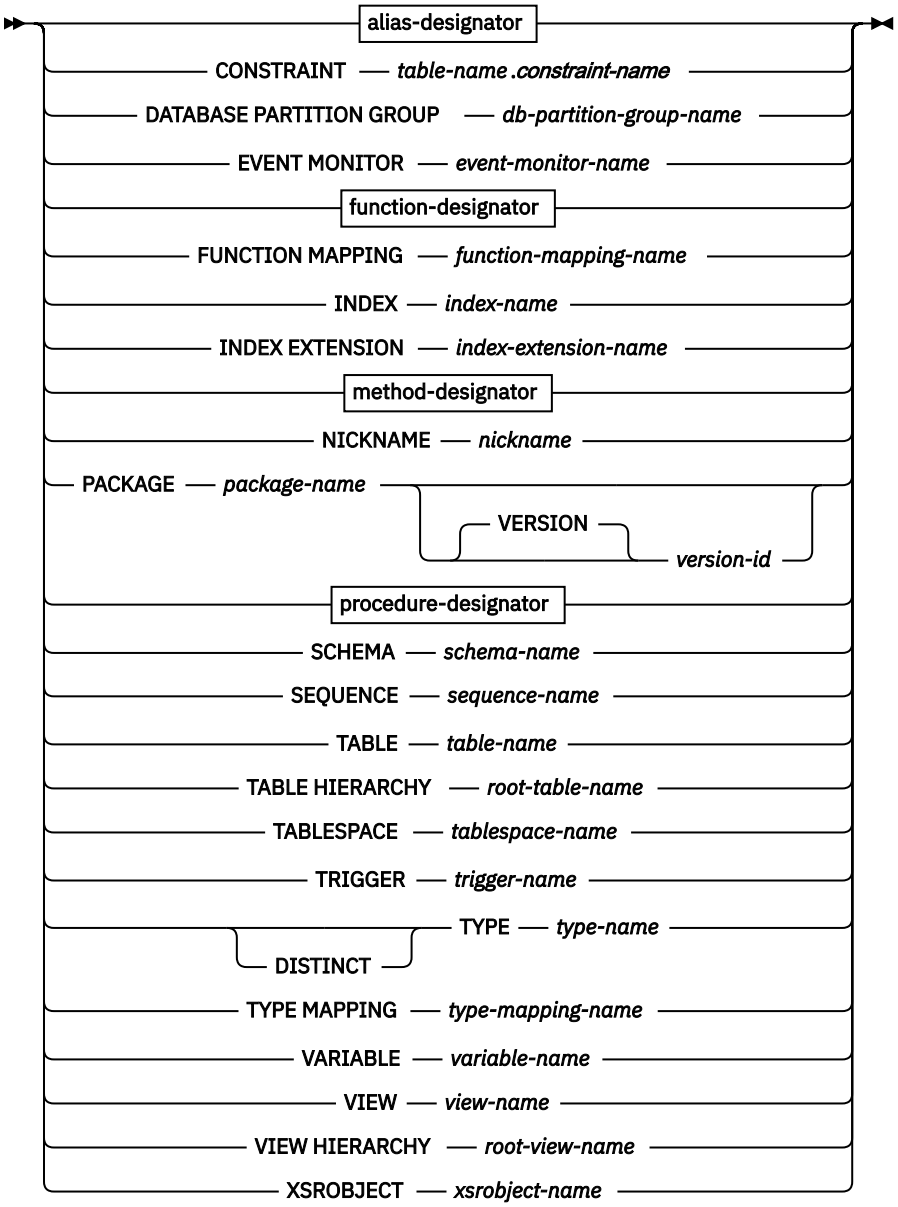
The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- Ownership of the object
- SECADM authority

Syntax

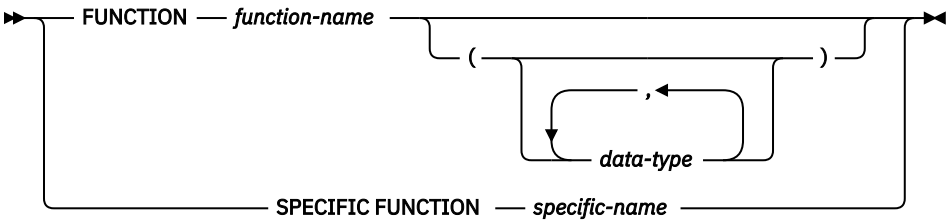


objects

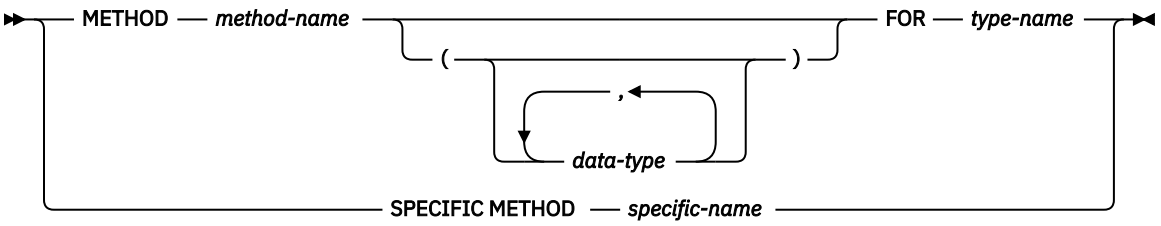


alias-designator

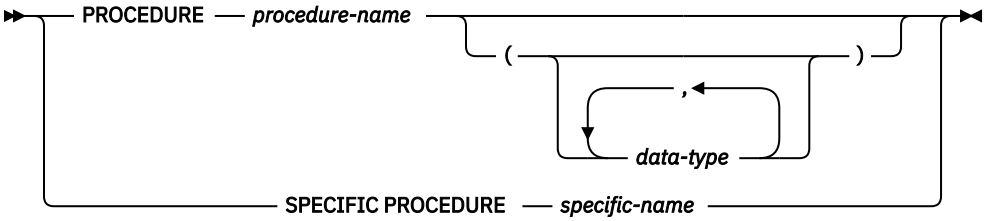
function-designator



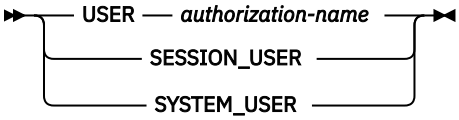
method-designator



procedure-designator



new-owner



Description

alias-designator

ALIAS *alias-name*
Identifies the alias that is to have its ownership transferred. The *alias-name* must identify an alias that is described in the catalog (SQLSTATE 42704). If PUBLIC is specified, the *alias-name* must identify a public alias that exists at the current server (SQLSTATE 42704).

FOR TABLE, or FOR SEQUENCE
Specifies the object type for the alias.

FOR TABLE
The alias is for a table, view, or nickname. When ownership of the alias is transferred, the value in the OWNER column for the alias in the SYSCAT.TABLES catalog view is replaced with the authorization ID of the new owner.

FOR SEQUENCE
The alias is for a sequence. When ownership of the alias is transferred, the value in the OWNER column for the alias in the SYSCAT.SEQUENCES catalog view is replaced with the authorization ID of the new owner.

CONSTRAINT *table-name.constraint-name*
Identifies the constraint that is to have its ownership transferred. The *table-name.constraint-name* combination must identify a constraint and the table that it constrains. The *constraint-name* must identify a constraint that is described in the catalog (SQLSTATE 42704).

When ownership of the constraint is transferred, the value in the OWNER column for the constraint in the SYSCAT.TABCONST catalog view is replaced with the authorization ID of the new owner.

- If the constraint is a FOREIGN KEY constraint, the OWNER column in the SYSCAT.REFERENCES catalog view is replaced with the authorization ID of the new owner.
- If the constraint is a PRIMARY KEY or UNIQUE constraint, the OWNER column in the SYSCAT.INDEXES catalog view for the index that was created implicitly for this constraint is replaced with the authorization ID of the new owner. If the index existed, and it is reused in this case, the owner of the index is not changed.

DATABASE PARTITION GROUP *db-partition-group-name*
Identifies the database partition group that is to have its ownership transferred. The *db-partition-group-name* must identify a database partition group that is described in the catalog (SQLSTATE 42704).

When ownership of the database partition group is transferred, the value in the OWNER column for the database partition group in the SYSCAT.DBPARTITIONGROUPS catalog view is replaced with the authorization ID of the new owner.

EVENT MONITOR *event-monitor-name*
Identifies the event monitor that is to have its ownership transferred. The *event-monitor-name* must identify an event monitor that is described in the catalog (SQLSTATE 42704).

When ownership of the event monitor is transferred, the value in the OWNER column for the event monitor in the SYSCAT.EVENTMONITORS catalog view is replaced with the authorization ID of the new owner.

If the identified event monitor is active, an error is returned (SQLSTATE 429BT).

If there are event files in the target path of a WRITE TO FILE event monitor whose ownership is being transferred, the event files are not deleted.

When ownership of WRITE TO TABLE event monitors is transferred, table information in the SYSCAT.EVENTTABLES catalog view is retained.

function-designator
Identifies the function that is to have its ownership transferred. For more information, see "Function, method, and procedure designators" on page 745. The specified function instance must be a user-defined function or function template that is described in the catalog. Ownership of functions that are implicitly generated by CREATE TYPE statements cannot be transferred (SQLSTATE 429BT).

When ownership of the function is transferred, the value in the OWNER column for the function in the SYSCAT.ROUTINES catalog view is replaced with the authorization ID of the new owner. Transferring ownership of an SQL function that has an associated package also implicitly transfers ownership of the package to the new owner.

SPECIFIC FUNCTION *specific-name*
Identifies the particular user-defined function that is to have its ownership transferred, using the specific name either specified or defaulted to at function creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile or bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific function instance in the named or implied schema; otherwise, an error is returned (SQLSTATE 42704).

When ownership of the specific function is transferred, the value in the OWNER column for the specific function in the SYSCAT.ROUTINES catalog view is replaced with the authorization ID of the new owner.

FUNCTION MAPPING *function-mapping-name*
Identifies the function mapping that is to have its ownership transferred. The *function-mapping-name* must identify a function mapping that is described in the catalog (SQLSTATE 42704).

When ownership of the function mapping is transferred, the value in the OWNER column for the function mapping in the SYSCAT.FUNCMAPPINGS catalog view is replaced with the authorization ID of the new owner.

INDEX *index-name*
Identifies the index or index specification that is to have its ownership transferred. The *index-name* must identify an index or index specification that is described in the catalog (SQLSTATE 42704).

When ownership of the index is transferred, the value in the OWNER column for the index in the SYSCAT.INDEXES catalog view is replaced with the authorization ID of the new owner.

Ownership of an index cannot be transferred if the table on which the index is defined is a global temporary table (SQLSTATE 429BT).

INDEX EXTENSION ***index-extension-name***
Identifies the index extension that is to have its ownership transferred. The *index-extension-name* must identify an index extension that is described in the catalog (SQLSTATE 42704).

When ownership of the index extension is transferred, the value in the OWNER column for the index extension in the SYSCAT.INDEXEXTENSIONS catalog view is replaced with the authorization ID of the new owner.

method-designator
Identifies the method that is to have its ownership transferred. For more information, see "Function, method, and procedure designators" on page 745. The method body specified must be a method that is described in the catalog (SQLSTATE 42704). The ownership of methods that are implicitly generated by the CREATE TYPE statement cannot be transferred (SQLSTATE 429BT).

When ownership of the method is transferred, the value in the OWNER column for the method in the SYSCAT.ROUTINES catalog view is replaced with the authorization ID of the new owner.

NICKNAME ***nickname***
Identifies the nickname that is to have its ownership transferred. The *nickname* must be a nickname that is described in the catalog (SQLSTATE 42704).

When ownership of the nickname is transferred, the value in the OWNER column for the nickname in the SYSCAT.TABLES catalog view is replaced with the authorization ID of the new owner.

PACKAGE ***package-name***
Identifies the package that is to have its ownership transferred. The package name must identify a package that is described in the catalog (SQLSTATE 42704).

VERSION ***version-id***
Identifies which package version is to have its ownership transferred. If a value is not specified, the version defaults to the empty string, and the ownership of this package is transferred. If multiple packages with the same package name but different versions exist, only the ownership of the package whose *version-id* is specified in the TRANSFER OWNERSHIP statement is transferred. Delimit the version identifier with double quotation marks when it:

- Is generated by the VERSION(AUTO) precompiler option
- Begins with a digit
- Contains lowercase or mixed-case letters

If the statement is invoked from an operating system command prompt, precede each double quotation mark delimiter with a back slash character to ensure that the operating system does not strip the delimiters.

When ownership of the package is transferred, the value in the BOUNDBY column for the package in the SYSCAT.PACKAGES catalog view is replaced with the authorization ID of the new owner.

The ownership of packages that are associated with SQL procedures, compiled SQL functions or compiled triggers cannot be transferred (SQLSTATE 429BT).

procedure-designator
Identifies the procedure that is to have its ownership transferred. For more information, see "Function, method, and procedure designators" on page 745. The procedure instance specified must be a procedure that is described in the catalog.

When ownership of the procedure is transferred, the value in the OWNER column for the procedure in the SYSCAT.ROUTINES catalog view is replaced with the authorization ID of the new owner.

Transferring ownership of an SQL procedure that has an associated package also implicitly transfers ownership of the package to the new owner.

SPECIFIC PROCEDURE ***specific-name***
Identifies the particular procedure that is to have its ownership transferred, using the specific name either specified or defaulted to at procedure creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In

static SQL statements, the QUALIFIER precompile or bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific procedure instance in the named or implied schema; otherwise, an error is returned (SQLSTATE 42704).

When ownership of the specific procedure is transferred, the value in the OWNER column for the specific procedure in the SYSCAT.ROUTINES catalog view is replaced with the authorization ID of the new owner.

SCHEMA *schema-name*
Identifies the schema that is to have its ownership transferred. The *schema-name* must identify a schema that is described in the catalog (SQLSTATE 42704).

When ownership of the schema is transferred, the value in the OWNER column and the DEFINER column for the schema in the SYSCAT.SCHEMATA catalog view is replaced with the authorization ID of the new owner.

Ownership of built-in schemas (where the definer is SYSIBM) cannot be transferred (SQLSTATE 42832).

SEQUENCE *sequence-name*
Identifies the sequence that is to have its ownership transferred. The *sequence-name* must identify a sequence that is described in the catalog (SQLSTATE 42704).

When ownership of the sequence is transferred, the value in the OWNER column for the schema in the SYSCAT.SEQUENCES catalog view is replaced with the authorization ID of the new owner.

TABLE *table-name*
Identifies the table that is to have its ownership transferred. The *table-name* must identify a table that exists in the database (SQLSTATE 42704) and must not identify a declared temporary table (SQLSTATE 42995).

When ownership of the table is transferred:

- The value in the OWNER column for the table in the SYSCAT.TABLES catalog view is replaced with the authorization ID of the new owner.
- The value in the OWNER column for all dependent objects on the table in the SYSCAT.TABDEP catalog view is replaced with the authorization ID of the new owner.

Ownership of subtables in a table hierarchy cannot be transferred (SQLSTATE 429BT).

In a federated system, ownership of a remote table that was created using transparent DDL can be transferred. Transferring the ownership of a remote table will not transfer ownership of the nickname that is associated with the table. Ownership of such a nickname can be transferred explicitly using the TRANSFER OWNERSHIP statement.

TABLE HIERARCHY *root-table-name*
Identifies the typed table that is the root table in a typed table hierarchy that is to have its ownership transferred. The *root-table-name* must identify a typed table that is the root table in the typed table hierarchy (SQLSTATE 428DR), and must refer to a typed table that exists in the database (SQLSTATE 42704).

When ownership of the table hierarchy is transferred:

- The value in the OWNER column for the root table and all of its subtables in the SYSCAT.TABLES catalog view is replaced with the authorization ID of the new owner.
- The value in the OWNER column for all dependent objects on the table and all of its subtables in the SYSCAT.TABDEP catalog view is replaced with the authorization ID of the new owner.

TABLESPACE *tablespace-name*
Identifies the table space that is to have its ownership transferred. The *tablespace-name* must identify a table space that is described in the catalog (SQLSTATE 42704).

When ownership of the table space is transferred, the value in the OWNER column for the table space in the SYSCAT.TABLESPACES catalog view is replaced with the authorization ID of the new owner.

TRIGGER *trigger-name*
Identifies the trigger that is to have its ownership transferred. The *trigger-name* must identify a trigger that is described in the catalog (SQLSTATE 42704).

When ownership of the trigger is transferred, the value in the OWNER column for the trigger in the SYSCAT.TRIGGERS catalog view is replaced with the authorization ID of the new owner. Transferring ownership of a compiled trigger also implicitly transfers ownership of the associated package to the new owner.

TYPE *type-name*
Identifies the user-defined type that is to have its ownership transferred. The *type-name* must identify a type that is described in the catalog (SQLSTATE 42704). If DISTINCT is specified, *type-name* must identify a distinct type that is described in the catalog (SQLSTATE 42704).

In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile or bind option implicitly specifies the qualifier for unqualified object names.

When ownership of the type is transferred, the value in the OWNER column for the type in the SYSCAT.DATATYPES catalog view is replaced with the authorization ID of the new owner.

TYPE MAPPING *type-mapping-name*
Identifies the user-defined data type mapping that is to have its ownership transferred. The *type-mapping-name* must identify a data type mapping that is described in the catalog (SQLSTATE 42704).

When ownership of the type mapping is transferred, the value in the OWNER column for the type mapping in the SYSCAT.TYPEMAPPINGS catalog view is replaced with the authorization ID of the new owner.

VARIABLE *variable-name*
Indicates that the object whose ownership is to be transferred is a created global variable. The *variable-name* must identify a global variable that exists at the current server (SQLSTATE 42704).

When the global variable is transferred, the value in the OWNER column for the global variable in the SYSCAT.VARIABLES catalog view is replaced with the authorization ID of the new owner.

VIEW *view-name*
Identifies the view that is to have its ownership transferred. The *view-name* must identify a view that exists in the database (SQLSTATE 42704).

When ownership of the view is transferred:

- The value in the OWNER column for the view in the SYSCAT.VIEWS catalog view is replaced with the authorization ID of the new owner.
- The value in the OWNER column for all dependent objects on the view in the SYSCAT.TABDEP catalog view is replaced with the authorization ID of the new owner.

The ownership of a subview in a view hierarchy cannot be transferred (SQLSTATE 429BT).

VIEW HIERARCHY *root-view-name*
Identifies the typed view that is the root view in a typed view hierarchy that is to have its ownership transferred. The *root-view-name* must identify a typed view that is the root view in the typed view hierarchy (SQLSTATE 428DR), and must refer to a typed view that exists in the database (SQLSTATE 42704).

When ownership of the view hierarchy is transferred:

- The value in the OWNER column for the root view and all of its subviews in the SYSCAT.VIEWS catalog view is replaced with the authorization ID of the new owner.
- The value in the OWNER column for all dependent objects on the view and all of its subviews in the SYSCAT.TABDEP catalog view is replaced with the authorization ID of the new owner.

XSROBJECT *xsrobject-name*
Identifies the XSR object that is to have its ownership transferred. The *xsrobject-name* must identify an XSR object that is described in the catalog (SQLSTATE 42704).

When ownership of the XSR object is transferred, the value in the OWNER column for the XSR object in the SYSCAT.XSROBJECTS catalog view is replaced with the authorization ID of the new owner.

USER *authorization-name*
Specifies the authorization ID to which ownership of the object is being transferred.

SESSION_USER
Specifies that the value of the SESSION_USER special register is to be used as the authorization ID to which ownership of the object is being transferred.

SYSTEM_USER
Specifies that the value of the SYSTEM_USER special register is to be used as the authorization ID to which ownership of the object is being transferred.

REVOKE PRIVILEGES
Indicates that the old owner of an object will lose all privileges on the object being transferred. This is the default option.

PRESERVE PRIVILEGES
Specifies that the current owner of an object that is to have its ownership transferred will continue to hold any existing privileges on the object after the transfer. For example, any privileges that were granted to the creator of a view when that view was created continue to be held by the original owner even after ownership has been transferred to another user.

Rules

- Ownership of most built-in objects (where the owner is SYSIBM) cannot be transferred (SQLSTATE 42832). However, you can transfer ownership of implicitly created schema objects that have SYSIBM in the OWNER column and do not have SYSIBM in the DEFINER column.
- Ownership of schemas whose name starts with 'SYS' cannot be transferred (SQLSTATE 42832).
- Ownership of the following objects cannot be explicitly transferred (SQLSTATE 429BT):
  - Subtables in a table hierarchy (they are transferred with the root hierarchy table)
  - Subviews in a view hierarchy (they are transferred with the root hierarchy view)
  - Indexes that are defined on global temporary tables
  - Methods or functions that are implicitly generated when a user-defined type is created
  - Module aliases and modules
  - Packages that depend on SQL procedures (they are transferred with the SQL procedure)
  - Event monitors that are active (they can be transferred when they are not active)
- An authorization ID that has SECADM authority cannot transfer the ownership of an object to itself, if it is not already the owner of the object (SQLSTATE 42502).
- Transfer of ownership on a variable or sequence with REVOKE PRIVILEGES will fail if any objects depend on the privileges being revoked (SQLSTATE 42893).
- Transfer of ownership on a function with REVOKE PRIVILEGES on a routine will fail if both of the following conditions are true (SQLSTATE 42893)
  - The specified routine is used in a view, trigger, constraint, index extension, SQL function, SQL method, transform group, or is referenced as the SOURCE of a sourced function.
  - The loss of the EXECUTE privilege would cause the owner of the view, trigger, constraint, index extension, SQL function, SQL method, transform group, or sourced function to no longer be able to execute the specified routine.
- Transfer of ownership on a module, table, view, nickname, or routine with REVOKE PRIVILEGES will cause any dependent view, MQT, function, trigger, and package to marked inoperative or invalid unless the old owner has the revoked privileges through a database authority or schema authority or through a group, role, or PUBLIC.

Notes

- All privileges that the current owner has that were granted as part of the creation of the object are transferred to the new owner. If the current owner has had a privilege on the object revoked, and that privilege was subsequently granted back, the privilege is not transferred. For implicitly created schema objects that have not already been transferred, the new owner is granted CREATEIN, DROPIN, and ALTERIN on the schema and can also grant only these privileges to other users.
- When the ownership of a database object is transferred, the new owner must have the set of privileges on the base objects, as indicated by the object's dependencies, that are required to maintain the object's existence unchanged. The new owner does not need the privileges required to create the object if those privileges are not required to maintain the object's existence.

  For example:

  – Consider a view with SELECT and INSERT dependencies on an underlying table. The privileges held by the new owner of the view must include at least SELECT (with or without the GRANT OPTION) and INSERT (with or without the GRANT OPTION) for the ownership transfer to be successful. If the dependencies were SELECT WITH GRANT OPTION and INSERT WITH GRANT OPTION, the privileges held by the new owner of the view must include at least SELECT WITH GRANT OPTION and INSERT WITH GRANT OPTION.
  – Consider a view with a dependency on a routine. The privileges held by the new owner of the view must include at least EXECUTE on the dependent routine.
  – Consider a trigger with a dependency on a table. The privileges held by the new owner of the trigger must include the same set of privileges on the table that are indicated by the trigger's dependencies. ALTER privilege on the table on which the trigger is defined is not required.

  The following table lists the catalog views that describe the objects on which other database objects depend.

Table 154. Catalog Views that Describe Objects on which Other Objects Depend

Database Object	**Catalog View**
CONSTRAINT	SYSCAT.CONSTDEP
FUNCTION	SYSCAT.ROUTINEDEP; SYSCAT.ROUTINES (for a sourced function)
INDEX	SYSCAT.INDEXDEP
INDEX EXTENSION	SYSCAT.INDEXEXTENSIONDEP
METHOD	SYSCAT.ROUTINEDEP
PACKAGE	SYSCAT.PACKAGEDEP
PROCEDURE	SYSCAT.ROUTINEDEP
TABLE	SYSCAT.TABDEP
TRIGGER	SYSCAT.TRIGDEP
VIEW	SYSCAT.TABDEP
XSROBJECT	SYSCAT.XSROBJECTDEP

  If ownership of a database object that depends on another object is to be transferred successfully, the new owner of the database object must hold certain privileges on the dependent object of that dependency:

  – If the dependent object is a sequence, the new owner must have the USAGE privilege on that sequence.
  – If the dependent object is a function, method, or procedure, the new owner must have the EXECUTE privilege on that function, method, or procedure.

  - If the dependent object is a package, the new owner must have the EXECUTE privilege on that package.
  - If the dependent object is an XSR object, the new owner must have the USAGE privilege on that XSR object.

  For any other dependent object of a dependency, use the TABAUTH column in the appropriate catalog view to determine what privileges the new owner must hold.
- If an attempt is made to transfer ownership of an object to its owner, a warning is returned (SQLSTATE 01676).
- Ownership of the following database objects cannot be transferred, because these objects have no owner: audit policies, buffer pools, roles, security labels, security label components, security policies, servers, transformation functions, trusted contexts, user mappings, and wrappers. Note that there is no OWNER column in the SYSCAT.AUDITPOLICIES, SYSCAT.BUFFERPOOLS, SYSCAT.CONTEXTS, SYSCAT.ROLES, SYSCAT.SECURITYLABELS, SYSCAT.SECURITYLABELCOMPONENTS, SYSCAT.SECURITYPOLICIES, SYSCAT.SERVERS, SYSCAT.TRANSFORMS, SYSCAT.USEROPTIONS, and SYSCAT.WRAPPERS catalog views.
- The schema name of an object whose ownership was transferred does not automatically change.
- ***Syntax alternatives***: For consistency with other SQL statements:
  - NODEGROUP can be specified in place of DATABASE PARTITION GROUP
  - SYNONYM can be specified in place of ALIAS

Examples

- *Example 1:* Transfer ownership of table T1 to PAUL.

```
TRANSFER OWNERSHIP OF TABLE WALID.T1
  TO USER PAUL PRESERVE PRIVILEGES
```

  The value in the OWNER column for the table WALID.T1 in the SYSCAT.TABLES catalog view is replaced with 'PAUL'. Paul is implicitly granted the following privileges on table WALID.T1 (assuming that the previous owner of the table did not lose any privileges on it): CONTROL and ALTER, DELETE, INDEX, INSERT, SELECT, UPDATE, REFERENCE (WITH GRANT OPTION).
- *Example 2:* Assume that JOHN creates tables T1 and T2, and that MIKE holds SELECT privilege on tables JOHN.T1 and JOHN.T2. MIKE creates view V1 that depends on tables JOHN.T1 and JOHN.T2. Transfer ownership of view V1 to HENRY, who has DBADM authority.

```
TRANSFER OWNERSHIP OF VIEW V1
  TO USER HENRY PRESERVE PRIVILEGES
```

  The value in the OWNER column for the view V1 in the SYSCAT.VIEWS catalog view is replaced with 'HENRY'. A new row is added to SYSCAT.TABAUTH with the following values: GRANTOR = 'SYSIBM', GRANTEE = 'HENRY', and TABNAME = 'V1'.
- *Example 3:* Assume that HENRY, who holds DBADM authority, creates a trigger TR1 that depends on table T1. Transfer ownership of trigger TR1 to WALID, who does not hold DBADM authority.

```
TRANSFER OWNERSHIP OF TRIGGER TR1
  TO USER WALID PRESERVE PRIVILEGES
```

  Ownership of the trigger is transferred successfully, even though Walid does not hold DBADM authority.
- *Example 4:* Assume that JOHN creates tables T1 and T2, and that MIKE holds SELECT privilege on table JOHN.T1 and CONTROL privilege on table JOHN.T2. PAUL holds SELECT privilege on tables JOHN.T1 and JOHN.T2. MIKE creates view V1 that depends on tables JOHN.T1 and JOHN.T2. The view has an

entry for the SELECT privilege in SYSCAT.TABAUTH and two SELECT dependencies in SYSCAT.TABDEP for tables JOHN.T1 and JOHN.T2. Transfer ownership of view V1 to PAUL, who is a regular user.

```
TRANSFER OWNERSHIP OF VIEW V1
  TO USER PAUL PRESERVE PRIVILEGES
```

Ownership of the view is transferred successfully, even though Paul does not hold CONTROL privilege on table JOHN.T2. Paul only needs SELECT privilege on tables JOHN.T1 and JOHN.T2 to maintain the view's existence. (The view only has SELECT privilege because Paul did not hold CONTROL privilege on both tables when the view was created and, as a result, he was not granted CONTROL on the view.) The value in the OWNER column for the view V1 in the SYSCAT.VIEWS catalog view is replaced with 'PAUL'. The value in the OWNER column for the view V1 in the SYSCAT.TABDEP catalog view is replaced with 'PAUL'. A new row is added to SYSCAT.TABAUTH with the following values: GRANTOR = 'SYSIBM', GRANTEE = 'PAUL', and TABNAME = 'V1'.

- *Example 5:* Assume that JOHN creates table T1, and that PUBLIC holds SELECT privilege on JOHN.T1. PAUL holds SELECT privilege on JOHN.T1 explicitly, and creates view V1 that depends on table JOHN.T1. Transfer ownership of view V1 to MIKE, who is not a DBADM, but who holds the required privileges to acquire view ownership through the special group PUBLIC.

  ```
  TRANSFER OWNERSHIP OF VIEW V1
    TO USER MIKE PRESERVE PRIVILEGES
  ```

  Ownership of the view is transferred successfully, because Mike holds SELECT privilege on table JOHN.T1 through PUBLIC. The value in the OWNER column for the view V1 in the SYSCAT.VIEWS catalog view is replaced with 'MIKE'. The value in the OWNER column for the view V1 in the SYSCAT.TABDEP catalog view is replaced with 'MIKE'. A new row is added to SYSCAT.TABAUTH with the following values: GRANTOR = 'SYSIBM', GRANTEE = 'MIKE', and TABNAME = 'V1'.
- *Example 6:* Similar to example 5, assume that JOHN creates table T1, and that role R1 holds SELECT privilege on JOHN.T1. PAUL holds SELECT privilege on JOHN.T1 explicitly, and creates view V1 that depends on table JOHN.T1. Transfer ownership of view V1 to MIKE, who is not a DBADM, but who holds the required privileges through membership in role R1 to acquire view ownership.

  ```
  TRANSFER OWNERSHIP OF VIEW V1
    TO USER MIKE PRESERVE PRIVILEGES
  ```

  Ownership of the view is transferred successfully, because Mike holds SELECT privilege on table JOHN.T1 through membership in role R1. The value in the OWNER column for the view V1 in the SYSCAT.VIEWS catalog view is replaced with 'MIKE'. The value in the OWNER column for the view V1 in the SYSCAT.TABDEP catalog view is replaced with 'MIKE'. A new row is added to SYSCAT.TABAUTH with the following values: GRANTOR = 'SYSIBM', GRANTEE = 'MIKE', and TABNAME = 'V1'.

TRUNCATE

The TRUNCATE statement deletes all of the rows from a table.

Invocation

This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared only if DYNAMICRULES run behavior is in effect for the package (SQLSTATE 42509).

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities for the table, and all subtables of a table hierarchy:

- DELETE privilege on the table to be truncated
- DELETEIN privilege on the schema containing the table
- CONTROL privilege on the table to be truncated

- DATAACCESS authority on the schema containing the table
- DATAACCESS authority

To ignore any DELETE triggers that are defined on the table, the privileges held by the authorization ID of the statement must include at least one of the following authorities for the table, and all subtables of a table hierarchy:

- ALTER privilege on the table
- ALTERIN privilege on the schema containing the table and all subtables of a table hierarchy
- CONTROL privilege on the table
- SCHEMAADM authority on the schema containing the table and all subtables of a table hierarchy
- DBADM authority

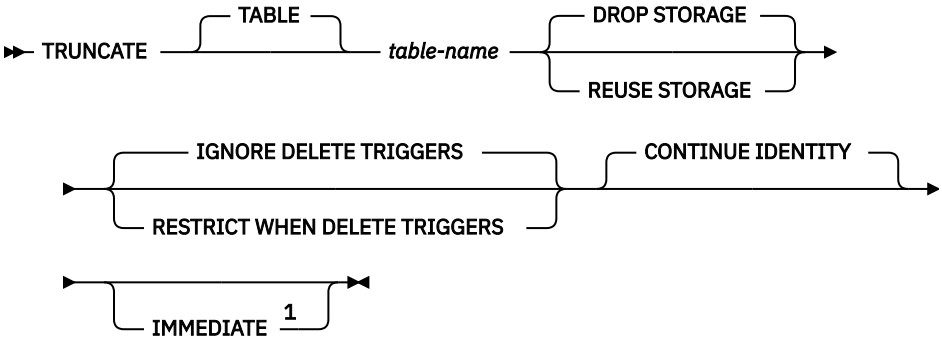
To truncate a table that is protected by a security policy, the privileges held by the authorization ID of the statement must include at least one of the following authorities:

- CONTROL privilege on the table
- DBADM authority

To truncate a table that has row access control activated, the authorization ID of the statement must include at least one of the following authorities:

- CONTROL privilege on the table
- DBADM authority

Syntax



Notes:

[1] IMMEDIATE is optional only for column-organized tables.

Description

table-name
Identifies the table to be truncated. The name must identify a table that exists at the current server (SQLSTATE 42704), but it cannot be a catalog table (SQLSTATE 42832), a nickname (SQLSTATE 42809), a view, a subtable, a staging table, a system-maintained materialized query table, a system-period temporal table (SQLSTATE 428HZ), or a range-clustered table (SQLSTATE 42807).

If *table-name* is the root table of a table hierarchy, all tables in the table hierarchy will be truncated.

DROP STORAGE or REUSE STORAGE
Specifies whether to drop or reuse the existing storage that is allocated for the table. The default is DROP STORAGE.

DROP STORAGE
All storage allocated for the table is released and made available. If this option is specified (implicitly or explicitly), an online backup would be blocked.

REUSE STORAGE
All storage allocated for the table will continue to be allocated for the table, but the storage will be considered empty. This option is only applicable to tables in DMS table spaces and is ignored otherwise.

IGNORE DELETE TRIGGERS or RESTRICT WHEN DELETE TRIGGERS
Specifies what to do when delete triggers are defined on the table. The default is IGNORE DELETE TRIGGERS.

IGNORE DELETE TRIGGERS
Any delete triggers that are defined for the table are not activated by the truncation operation.

RESTRICT WHEN DELETE TRIGGERS
An error is returned if delete triggers are defined on the table (SQLSTATE 428GJ).

CONTINUE IDENTITY
If an identity column exists for the table, the next identity column value generated continues with the next value that would have been generated if the TRUNCATE statement had not been executed.

IMMEDIATE
Specifies that the truncate operation is processed immediately and cannot be undone. The statement must be the first statement in a transaction (SQLSTATE 25001).

The truncated table is immediately available for use in the same unit of work. Although a ROLLBACK statement is allowed to execute after a TRUNCATE statement, the truncate operation is not undone, and the table remains in a truncated state. For example, if another data change operation is done on the table after the TRUNCATE IMMEDIATE statement and then the ROLLBACK statement is executed, the truncate operation will not be undone, but all other data change operations are undone.

The IMMEDIATE clause can be excluded only for column organized tables.

If IMMEDIATE is not specified for column organized tables, the TRUNCATE statement does not need to be the first statement in the work unit and you can use a ROLLBACK statement to undo the truncate operation.

Attention: The IMMEDIATE clause is optional only in Db2 Version 11.5 Mod Pack 2 and later versions.

Rules

- ***Referential Integrity:*** The table, and all tables in a table hierarchy, must not be a parent table in an enforced referential constraint (SQLSTATE 428GJ). A self-referencing RI constraint is permitted.
- ***Partitioned tables:*** The table must not be in set integrity pending state due to being altered to attach a data partition (SQLSTATE 55019). The table needs to be checked for integrity before executing the TRUNCATE statement. The table must not have any logically detached partitions (SQLSTATE 55057). The asynchronous partition detach task must complete before executing the TRUNCATE statement.
- ***Exclusive Access:*** No other session can have a cursor open on the table, or a lock held on the table (SQLSTATE 25001).
- ***WITH HOLD cursors:*** The current session cannot have a WITH HOLD cursor open on the table (SQLSTATE 25001).

Notes

- ***Table statistics:*** The statistics for the table are not changed by the TRUNCATE statement.
- ***Number of rows deleted:*** SQLERRD(3) in the SQLCA is set to -1 for the truncate operation. The number of rows that were deleted from the table is not returned.
- On column-organized tables, the following rules apply to the IMMEDIATE clause when it is optional:
  - If you do not specify the IMMEDIATE option, the TRUNCATE statement is processed and can be undone.
  - The TRUNCATE statement can be anywhere within the transaction scope.

- The TRUNCATE statement can be undone before the transaction completes.
- The truncated table can be used immediately in the same unit of work.
- A ROLLBACK statement can be executed after a TRUNCATE statement without the IMMEDIATE option is processed. The TRUNCATE operation is then undone.
- For example, if another data change operation is done on the table after the TRUNCATE statement without the IMMEDIATE option is processed, and if then the ROLLBACK statement is executed, the TRUNCATE operation is also undone.
- The storage is automatically asynchronously reclaimed after the transaction completes.
- The truncate operation will perform in a similar way to a mass delete operation. The number of rows that are deleted from the table is returned at SQLERRD(4).
- The SQLWARN(5) value is 'W' because the underlying DELETE statement does not include a WHERE clause.

Examples

- *Example 1:* Empty an unused inventory table regardless of any existing triggers and return its allocated space.

```
TRUNCATE TABLE INVENTORY
  IGNORE DELETE TRIGGERS
  DROP STORAGE
  IMMEDIATE
```

- *Example 2:* Empty an unused inventory table regardless of any existing delete triggers but preserve its allocated space for later reuse.

```
TRUNCATE TABLE INVENTORY
   REUSE STORAGE
   IGNORE DELETE TRIGGERS
   IMMEDIATE
```

- *Example 3:* If IMMEDIATE is not specified for column organized tables, the TRUNCATE statement does not have to be the first statement in the work unit, and you can use a ROLLBACK statement to undo the truncate operation.

```
SELECT COUNT(*) FROM TAB10;
TRUNCATE TABLE TAB10;
SELECT COUNT(*) FROM TAB10;
ROLLBACK;
```

  Note that the truncate statement without IMMEDIATE is not the first statement in the work unit, and that the truncate statement without IMMEDIATE can be rolled back.

UPDATE

The UPDATE statement updates the values of specified columns in rows of a table, view or nickname, or the underlying tables, nicknames, or views of the specified fullselect.

Updating a row of a view updates a row of its base table, if no INSTEAD OF trigger is defined for the update operation on this view. If such a trigger is defined, the trigger will be executed instead. Updating a row using a nickname updates a row in the data source object to which the nickname refers.

The forms of this statement are:

- The *Searched* UPDATE form is used to update one or more rows (optionally determined by a search condition).
- The *Positioned* UPDATE form is used to update exactly one row (as determined by the current position of a cursor).

Invocation

An UPDATE statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:

- UPDATE privilege on the target table, view, or nickname
- UPDATE privilege on each of the columns that are to be updated, including the columns of the BUSINESS_TIME period if a period-clause is specified
- UPDATEIN privilege on schema of the target table, view or nickname
- CONTROL privilege on the target table, view, or nickname
- DATAACCESS on the schema containing the target table, view, or nickname
- DATAACCESS authority

If a *row-fullselect* is included in the assignment, the privileges held by the authorization ID of the statement must include at least one of the following authorities for each referenced table, view, or nickname:

- SELECT privilege
- SELECTIN privilege on the schema containing the referenced table, view, or nickname
- CONTROL privilege
- DATAACCESS on the schema containing the referenced table, view, or nickname
- DATAACCESS authority

For each table, view, or nickname referenced by a subquery, the privileges held by the authorization ID of the statement must also include at least one of the following authorities:

- SELECT privilege
- SELECTIN privilege on the schema containing the table, view or nickname
- CONTROL privilege
- DATAACCESS on the schema containing the table, view or nickname
- DATAACCESS authority

If the package used to process the statement is precompiled with SQL92 rules (option LANGLEVEL with a value of SQL92E or MIA), and the searched form of an UPDATE statement includes a reference to a column of the table, view, or nickname in the right side of the *assignment-clause*, or anywhere in the *search-condition*, the privileges held by the authorization ID of the statement must also include at least one of the following authorities:

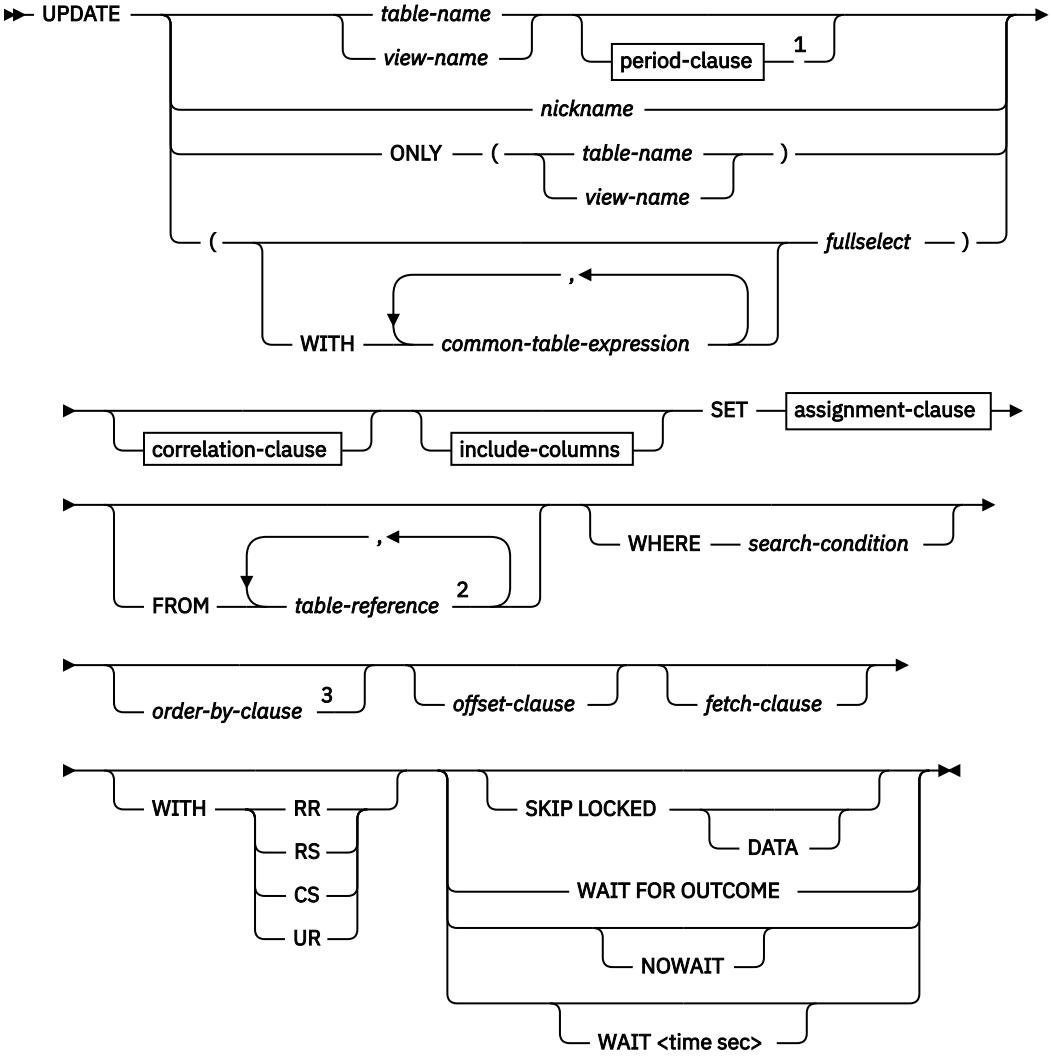
- SELECT privilege
- CONTROL privilege
- SELECTIN privilege on the schema containing the table, view, or nickname
- DATAACCESS on the schema containing the table, view or nickname
- DATAACCESS authority

If the specified table or view is preceded by the ONLY keyword, the privileges held by the authorization ID of the statement must also include the SELECT privilege for every subtable or subview of the specified table or view, or SELECTIN privilege on the schema containing the subtables or subviews of the specified table or view.

GROUP privileges are not checked for static UPDATE statements.

If the target of the update operation is a nickname, privileges on the object at the data source are not considered until the statement is executed at the data source. At this time, the authorization ID that is used to connect to the data source must have the privileges that are required for the operation on the object at the data source. The authorization ID of the statement can be mapped to a different authorization ID at the data source.

Syntax (searched-update)



Notes:

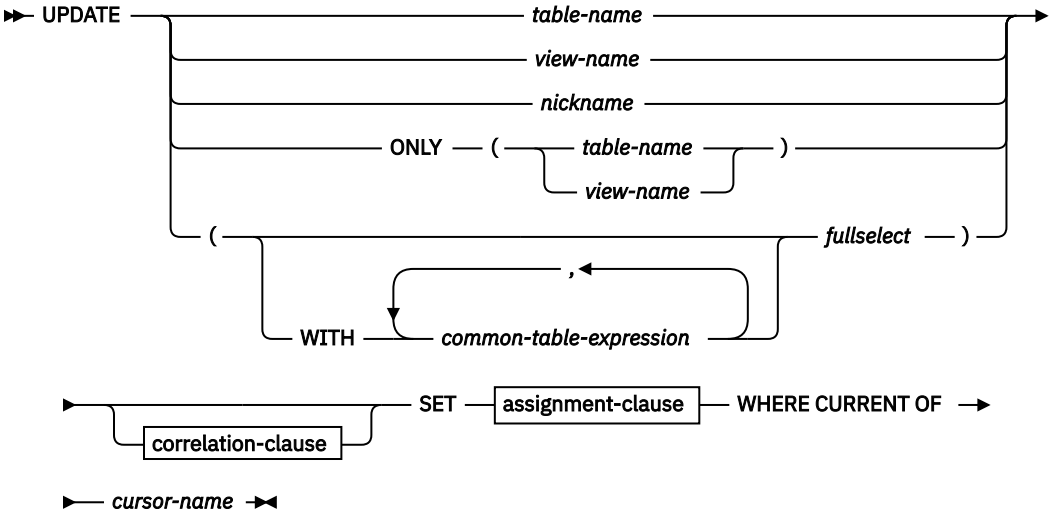
[1] If the *period-clause* is specified, neither the *offset-clause* nor the *fetch-clause* can be specified (SQLSTATE 42601).

[2] The specified table-reference cannot be an *analyze_table-expression* (that is, the result of a data mining model) or a *data-change-table-reference* (that is, the result of a nested UPDATE, DELETE, or INSERT statement) (SQLSTATE 42601).

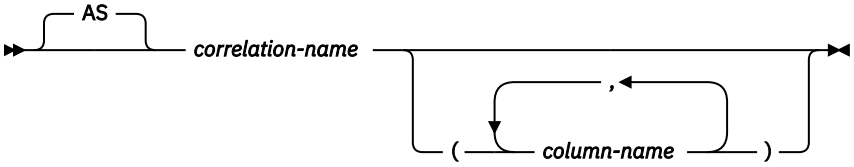
[3] If the *order-by-clause* is specified, either the *offset-clause* or *fetch-clause* must also be specified (SQLSTATE 42601).

period-clause

Syntax (positioned-update)

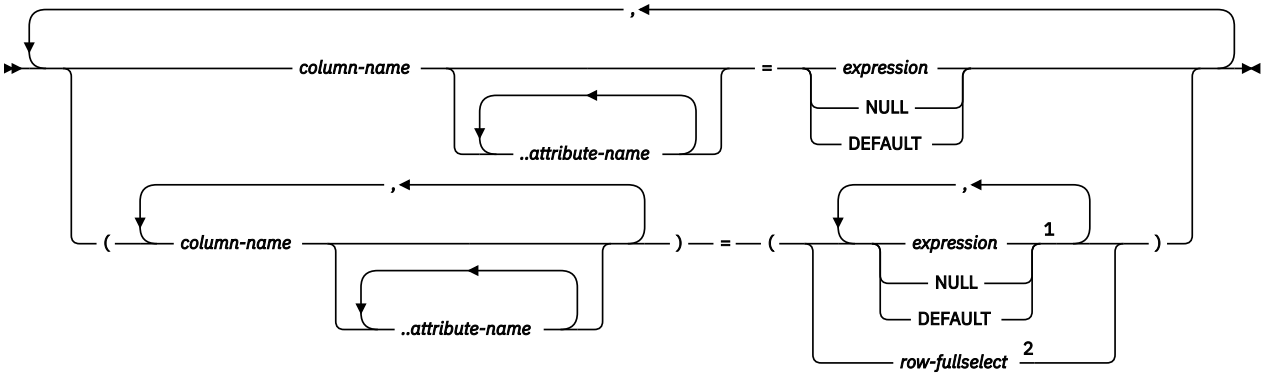


correlation-clause



include-columns

assignment-clause



Notes:

[1] The number of expressions, NULLs and DEFAULTs must match the number of column names.
[2] The number of columns in the select list must match the number of column names.

Description

table-name*, *view-name*, *nickname*, or *(fullselect)

Identifies the object of the update operation. The name must identify one of the following objects:

- A table, view, or nickname described in the catalog at the current server
- A table or view at a remote server specified using a remote-object-name

The object must not be a catalog table, a view of a catalog table (unless it is one of the updatable SYSSTAT views), a system-maintained materialized query table, or a read-only view that has no INSTEAD OF trigger defined for its update operations.

If *table-name* is a typed table, rows of the table or any of its proper subtables may get updated by the statement. Only the columns of the specified table may be set or referenced in the WHERE clause. For a positioned UPDATE, the associated cursor must also have specified the same table, view or nickname in the FROM clause without using ONLY.

If the object of the update operation is a fullselect, the fullselect must be updatable, as defined in the "Updatable views" Notes item in the description of the CREATE VIEW statement.

If the object of the update operation is a nickname, the extended indicator variable values of DEFAULT and UNASSIGNED must not be used (SQLSTATE 22539).

For additional restrictions related to temporal tables and use of a view or *fullselect* as the target of the update operation, see "Considerations for a system-period temporal table" and "Considerations for an application-period temporal table" in the Notes section of this topic.

ONLY (*table-name*)
Applicable to typed tables, the ONLY keyword specifies that the statement should apply only to data of the specified table and rows of proper subtables cannot be updated by the statement. For a positioned UPDATE, the associated cursor must also have specified the table in the FROM clause using ONLY. If *table-name* is not a typed table, the ONLY keyword has no effect on the statement.

ONLY (*view-name*)
Applicable to typed views, the ONLY keyword specifies that the statement should apply only to data of the specified view and rows of proper subviews cannot be updated by the statement. For a positioned UPDATE, the associated cursor must also have specified the view in the FROM clause using ONLY. If *view-name* is not a typed view, the ONLY keyword has no effect on the statement.

period-clause
Specifies that a period clause applies to the target of the update operation. If the target of the update operation is a view, the following conditions apply to the view:

- The FROM clause of the outer fullselect of the view definition must include a reference, directly or indirectly, to an application-period temporal table (SQLSTATE 42724M).
- An INSTEAD OF UPDATE trigger must not be defined for the view (SQLSTATE 428HY).

FOR PORTION OF BUSINESS_TIME
Specifies that the update only applies to row values for the portion of the period in the row that is specified by the period clause. The BUSINESS_TIME period must exist in the table (SQLSTATE 4274M).

FROM *value1* TO *value2*
Specifies that the update applies to rows for the period specified from *value1* up to *value2*. No rows are updated if *value1* is greater than or equal to *value2*, or if *value1* or *value2* is the null value (SQLSTATE 02000).

For the period specified with FROM *value1* TO *value2*, the BUSINESS_TIME period in a row in the target of the update is in any of the following states:

- **Overlaps the beginning** of the specified period if the value of the begin column is less than *value1* and the value of the end column is greater than *value1*.
- **Overlaps the end** of the specified period if the value of the end column is greater than or equal to *value2* and the value of the begin column is less than *value2*.
- Is **fully contained** within the specified period if the value for the begin column for BUSINESS_TIME is greater than or equal to *value1* and the value for the corresponding end column is less than or equal to *value2*.
- Is **partially contained** in the specified period if the row overlaps the beginning of the specified period or the end of the specified period, but not both.

- **Fully overlaps** the specified period if the period in the row overlaps the beginning and end of the specified period.
- Is **not contained** in the period if both columns of BUSINESS_TIME are less than or equal to *value1* or greater than or equal to *value2*.

If the BUSINESS_TIME period in a row is not contained in the specified period, the row is not updated. Otherwise, the update is applied based on how the values in the columns of the BUSINESS_TIME period overlap the specified period as follows:

- If the BUSINESS_TIME period in a row is fully contained within the specified period, the row is updated and the values of the begin column and end column of BUSINESS_TIME are unchanged.
- If the BUSINESS_TIME period in a row is partially contained in the specified period and overlaps the beginning of the specified period:
  - The row is updated. In the updated row, the value of the begin column is set to *value1* and the value of the end column is the original value of the end column.
  - A row is inserted using the original values from the row, except that the end column is set to *value1*.
- If the BUSINESS_TIME period in a row is partially contained in the specified period and overlaps the end of the specified period:
  - The row is updated. In the updated row, the value of the begin column is the original value of the begin column and the end column is set to *value2*.
  - A row is inserted using the original values from the row, except that the begin column is set to *value2*.
- If the BUSINESS_TIME period in a row fully overlaps the specified period:
  - The row is updated. In the updated row the value of the begin column is set to *value1* and the value of the end column is set to *value2*.
  - A row is inserted using the original values from the row, except that the end column is set to *value1*.
  - An additional row is inserted using the original values from the row, except that the begin column is set to *value2*.

value1* and *value2

Each expression must return a value that has a date data type, timestamp data type, or a valid data type for a string representation of a date or timestamp (SQLSTATE 428HY). The result of each expression must be comparable to the data type of the columns of the specified period (SQLSTATE 42884). See the comparison rules described in "Assignments and comparisons".

Each expression can contain any of the following supported operands (SQLSTATE 428HY):

- Constant
- Special register
- Variable. For details, refer to "References to variables" in the "Identifiers" topic, in *SQL Reference Volume 1* .
- Scalar function whose arguments are supported operands (though user-defined functions and non-deterministic functions cannot be used)
- CAST specification where the cast operand is a supported operand
- Expression using arithmetic operators and operands

correlation-clause

Can be used within *search-condition* or *assignment-clause* to designate a table, view, nickname, or fullselect. For a description of *correlation-clause*, see "table-reference" in the description of "Subselect".

include-columns
Specifies a set of columns that are included, along with the columns of *table-name* or *view-name*, in the intermediate result table of the UPDATE statement when it is nested in the FROM clause of a fullselect. The *include-columns* are appended at the end of the list of columns that are specified for *table-name* or *view-name*.

INCLUDE
Specifies a list of columns to be included in the intermediate result table of the UPDATE statement.

column-name
Specifies a column of the intermediate result table of the UPDATE statement. The name cannot be the same as the name of another include column or a column in *table-name* or *view-name* (SQLSTATE 42711).

data-type
Specifies the data type of the include column. The data type must be one that is supported by the CREATE TABLE statement.

SET
Introduces the assignment of values to column names.

assignment-clause

column-name
Identifies a column to be updated. If extended indicator variables are not enabled, the *column-name* must identify an updatable column of the specified table, view, or nickname, or identify an INCLUDE column. The object ID column of a typed table is not updatable (SQLSTATE 428DZ). A column must not be specified more than once, unless it is followed by *..attribute-name* (SQLSTATE 42701).

If it specifies an INCLUDE column, the column name cannot be qualified.

For a Positioned UPDATE:

- If the *update-clause* was specified in the *select-statement* of the cursor, each column name in the *assignment-clause* must also appear in the *update-clause*.
- If the *update-clause* was not specified in the *select-statement* of the cursor and LANGLEVEL MIA or SQL92E was specified when the application was precompiled, the name of any updatable column may be specified.
- If the *update-clause* was not specified in the *select-statement* of the cursor and LANGLEVEL SAA1 was specified either explicitly or by default when the application was precompiled, no columns may be updated.

..attribute-name
Specifies the attribute of a structured type that is set (referred to as an *attribute assignment*. The *column-name* specified must be defined with a user-defined structured type (SQLSTATE 428DP). The attribute-name must be an attribute of the structured type of *column-name* (SQLSTATE 42703). An assignment that does not involve the *..attribute-name* clause is referred to as a *conventional assignment*.

expression
Indicates the new value of the column. The expression is any expression of the type described in "Expressions". The expression cannot include an aggregate function except when it occurs within a scalar fullselect (SQLSTATE 42903).

An *expression* may contain references to columns of the target table of the UPDATE statement. For each row that is updated, the value of such a column in an expression is the value of the column in the row before the row is updated.

An expression cannot contain references to an INCLUDE column. If *expression* is a single host variable, the host variable can include an indicator variable that is enabled for extended indicator variables. If extended indicator variables are enabled, the extended indicator variable values of

default (-5) or unassigned (-7) must not be used (SQLSTATE 22539) if either of the following statements is true:

- The expression is more complex than a single host variable with explicit casts
- The target column has data type of structured type

NULL
Specifies the null value and can only be specified for nullable columns (SQLSTATE 23502). NULL cannot be the value in an attribute assignment (SQLSTATE 429B9) unless it is specifically cast to the data type of the attribute.

DEFAULT
Specifies that the default value should be used based on how the corresponding column is defined in the table. The value that is inserted depends on how the column was defined.

- If the column was defined as a generated column based on an expression, the column value will be generated by the system, based on the expression.
- If the column was defined using the IDENTITY clause, the value is generated by the database manager.
- If the column was defined using the WITH DEFAULT clause, the value is set to the default defined for the column (see *default-clause* in "ALTER TABLE").
- If the column was defined using the NOT NULL clause and the GENERATED clause was not used, or the WITH DEFAULT clause was not used, or DEFAULT NULL was used, the DEFAULT keyword cannot be specified for that column (SQLSTATE 23502).
- If the column was defined using the ROW CHANGE TIMESTAMP clause, the value is generated by the database manager.

The only value that a generated column defined with the GENERATED ALWAYS clause can be set to is DEFAULT (SQLSTATE 428C9).

The DEFAULT keyword cannot be used as the value in an attribute assignment (SQLSTATE 429B9).

The DEFAULT keyword cannot be used as the value in an assignment for update on a nickname where the data source does not support DEFAULT syntax.

row-fullselect
Specifies a fullselect that returns a single row. The result column values are assigned to each corresponding *column-name*. If the fullselect returns no rows, the null value is assigned to each column; an error occurs if any column to be updated is not nullable. An error also occurs if there is more than one row in the result.

A *row-fullselect* may contain references to columns of the target table of the UPDATE statement. For each row that is updated, the value of such a column in an expression is the value of the column in the row before the row is updated. An error is returned if there is more than one row in the result (SQLSTATE 21000).

FROM

Specifies a list of source tables that supply values for assignment to target table columns. The source tables are implicitly inner joined with the target table with the WHERE clause specifying the join condition. The rows in the target table that satisfy the WHERE condition are updated with the values from the source table rows.

When an UPDATE statement specifies a FROM clause:

- The target of the update operation cannot be a nickname.
- The source for the update operation cannot be an *analyze_table-expression* (that is, the result of a data mining model) or a *data-change-table-reference* (that is, the result of a nested UPDATE, DELETE, or INSERT statement).

For a description of *table-reference*, see “table-reference” on page 644.

WHERE
Introduces a condition that indicates what rows are updated. You can omit the clause, give a search condition, or name a cursor. If the clause is omitted, all rows of the table, view or nickname are updated.

search-condition
Each *column-name* in the search condition, other than in a subquery, must name a column of the table, view or nickname. When the search condition includes a subquery in which the same table is the base object of both the UPDATE and the subquery, the subquery is completely evaluated before any rows are updated.

The search-condition is applied to each row of the table, view or nickname and the updated rows are those for which the result of the search-condition is true.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a row, and the results used in applying the search condition. In actuality, a subquery with no correlated references is executed only once, whereas a subquery with a correlated reference may have to be executed once for each row.

CURRENT OF *cursor-name*
Identifies the cursor to be used in the update operation. The *cursor-name* must identify a declared cursor, explained in "DECLARE CURSOR". The DECLARE CURSOR statement must precede the UPDATE statement in the program.

The specified table, view, or nickname must also be named in the FROM clause of the SELECT statement of the cursor, and the result table of the cursor must not be read-only. (For an explanation of read-only result tables, see "DECLARE CURSOR".)

When the UPDATE statement is executed, the cursor must be positioned on a row; that row is updated.

This form of UPDATE cannot be used (SQLSTATE 42828) if the cursor references:

- A view on which an INSTEAD OF UPDATE trigger is defined
- A view that includes an OLAP function in the select list of the fullselect that defines the view
- A view that is defined, either directly or indirectly, using the WITH ROW MOVEMENT clause

order-by-clause
Specifies the order of the rows for application of the *offset-clause* and *fetch-clause*. Specify an *order-by-clause* to ensure a predictable order for determining the set of rows to be updated based on the *offset-clause* and *fetch-clause*. For details on the *order-by-clause*, see “order-by-clause” on page 703.

offset-clause
Limits the effect of the update by skipping a subset of the qualifying rows. For details on the *offset-clause*, refer to “offset-clause” on page 706.

fetch-clause
Limits the effect of the update to a subset of the qualifying rows. For details on the *fetch-clause*, refer to “fetch-clause” on page 705.

WITH
Specifies the isolation level at which the UPDATE statement is executed.

RR
Repeatable Read

RS
Read Stability

CS
Cursor Stability

UR
Uncommitted Read

The default isolation level of the statement is the isolation level of the package in which the statement is bound. The WITH clause has no effect on nicknames, which always use the default isolation level of the statement.

SKIP LOCKED DATA

The SKIP LOCKED DATA clause specifies that rows are skipped when incompatible locks that would block the progress of the statement are held on the rows by other transactions. These rows can belong to any accessed table addressed in the statement, including tables accessed in a subquery. This clause applies when the isolation level is CS or RS and is ignored when an isolation level of UR or RR is in effect. It applies to row and block level locks.

Invocation

SKIP LOCKED DATA is ignored if it is specified when WITH RR or WITH UR. The default isolation level of the statement depends on the isolation of the package or plan with which the statement is bound, and whether the result table is read-only. If the default isolation level of the statement is Repeatable Read or Uncommitted Read, then SKIP LOCKED DATA is ignored.

NOWAIT / WAIT <time sec>

Attention: The following feature is available in Db2 11.5.6 and later versions.

The NOWAIT and WAIT clauses specify the number of seconds to wait for a lock before returning an error indicating that a lock cannot be obtained.

When using the WAIT clause, `<time sec>` is an integer between `-1` and `32767`.

Note: For `NOWAIT` and `WAIT 0`, locks are not waited for. If no lock is available at the time of the request, a `-911` error is returned.

When a WAIT value of `-1` is specified, lock timeout detection is turned off. In this situation a lock is waited for (if one is not available at the time of the request) until either of the following events occur:

- The lock is granted.
- A deadlock occurs.

Use of the NOWAIT and WAIT clauses overwrites the value of the LOCKTIMEOUT database configuration variable and the value of the CURRENT LOCK TIMEOUT special register for this update statement. This means that adding the NOWAIT/WAIT clause with a wait time value of **t** has the same effect as executing the update statement with a LOCKTIMEOUT value or CURRENT LOCK TIMEOUT value of **t**.

While the NOWAIT and WAIT clauses are not allowed for positioned updates and deletes, you can use them in the declaration of the cursor. When used in the cursor declaration, the specified wait time value is inherited by the statements that use this cursor.

Rules

- ***Triggers:*** UPDATE statements may cause triggers to be executed. A trigger may cause other statements to be executed, or may raise error conditions based on the update values. If an update operation on a view causes an INSTEAD OF trigger to fire, validity, referential integrity, and constraints will be checked against the updates that are performed in the trigger, and not against the view that caused the trigger to fire, or its underlying tables.
- ***Assignment***: Update values are assigned to columns according to specific assignment rules.
- ***Validity***: The updated row must conform to any constraints imposed on the table (or on the base table of the view) by any unique index on an updated column.

  If a view is used that is not defined using WITH CHECK OPTION, rows can be changed so that they no longer conform to the definition of the view. Such rows are updated in the base table of the view and no longer appear in the view.

  If a view is used that is defined using WITH CHECK OPTION, an updated row must conform to the definition of the view. For an explanation of the rules governing this situation, see "CREATE VIEW".

- ***Check constraint***: Update value must satisfy the check-conditions of the check constraints defined on the table.

  An UPDATE to a table with check constraints defined has the constraint conditions for each column updated evaluated once for each row that is updated. When processing an UPDATE statement, only the check constraints referring to the updated columns are checked.
- ***Referential integrity***: The value of the parent unique keys cannot be changed if the update rule is RESTRICT and there are one or more dependent rows. However, if the update rule is NO ACTION, parent unique keys can be updated as long as every child has a parent key by the time the update statement completes. A non-null update value of a foreign key must be equal to a value of the primary key of the parent table of the relationship.
- ***XML values:*** When an XML column value is updated, the new value must be a well-formed XML document (SQLSTATE 2200M).
- ***Security policy:*** If the identified table or the base table of the identified view is protected with a security policy, the session authorization ID must have the label-based access control (LBAC) credentials that allow:
  - Write access to all protected columns that are being updated (SQLSTATE 42512)
  - Write access for any explicit value provided for a DB2SECURITYLABEL column for security policies that were created with the RESTRICT NOT AUTHORIZED WRITE SECURITY LABEL option (SQLSTATE 23523)
  - Read and write access to all rows that are being updated (SQLSTATE 42519)

  The session authorization ID must also have been granted a security label for write access for the security policy if an implicit value is used for a DB2SECURITYLABEL column (SQLSTATE 23523), which can happen when:
  - The DB2SECURITYLABEL column is not included in the list of columns that are to be updated (and so it will be implicitly updated to the security label for write access of the session authorization ID)
  - A value for the DB2SECURITYLABEL column is explicitly provided but the session authorization ID does not have write access for that value, and the security policy is created with the OVERRIDE NOT AUTHORIZED WRITE SECURITY LABEL option
- **Extended indicator variable usage**: If enabled, indicator variable values other than 0 (zero) through -7 must not be input (SQLSTATE 22010). Also, if enabled, the default and unassigned extended indicator variable values must not appear in contexts in which they are not supported (SQLSTATE 22539).
- **Extended indicator variables**: In the *assignment-clause* of an UPDATE statement, an *expression* that is a reference to a single host variable, or a host variable being explicitly cast can result in assigning an extended indicator variable value. Assigning an extended indicator variable-based value of unassigned has the effect of leaving the target column set to its current value, as if it had not been specified in the statement. Assigning an extended indicator variable-based value of default assigns the default value of the column. For information about default values of data types, see the description of the DEFAULT clause in “CREATE TABLE ” on page 1351.

  If a target column is not updatable (for example, a column in a view that is defined as an expression), then it must be assigned the extended indicator variable-based value of unassigned (SQLSTATE 42808).

  If the target column is a column defined as GENERATED ALWAYS, then it must be assigned the DEFAULT keyword, or the extended indicator variable-based values of default or unassigned (SQLSTATE 428C9).

  The UPDATE statement must not assign all target columns to an extended indicator variable-based value of unassigned (SQLSTATE 22540).

Notes

- If an update value violates any constraints, or if any other error occurs during the execution of the UPDATE statement, no rows are updated. The order in which multiple rows are updated is undefined.

- An update to a view defined using the WITH ROW MOVEMENT clause could cause a delete operation and an insert operation against the underlying tables of the view. For details, see the description of the CREATE VIEW statement.
- When an UPDATE statement completes execution, the value of SQLERRD(3) in the SQLCA is the number of rows that qualified for the update operation. In the context of an SQL procedure statement, the value can be retrieved using the ROW_COUNT variable of the GET DIAGNOSTICS statement. The SQLERRD(5) field contains the number of rows inserted, deleted, or updated by all activated triggers.
- Unless appropriate locks already exist, one or more exclusive locks are acquired by the execution of a successful UPDATE statement. Until the locks are released, the updated row can only be accessed by the application process that performed the update (except for applications using the Uncommitted Read isolation level). For further information on locking, see the descriptions of the COMMIT, ROLLBACK, and LOCK TABLE statements.
- When updating the column distribution statistics for a typed table, the subtable that first introduced the column must be specified.
- Multiple attribute assignments on the same structured type column occur in the order specified in the SET clause and, within a parenthesized set clause, in left-to-right order.
- An attribute assignment invokes the mutator method for the attribute of the user-defined structured type. For example, the assignment `st..a1=x` has the same effect as using the mutator method in the assignment `st = st..a1(x)`.
- While a given column may be a target column in only one conventional assignment, a column may be a target column in multiple attribute assignments (but only if it is not also a target column in a conventional assignment).
- When an identity column defined as a distinct type is updated, the entire computation is done in the source type, and the result is cast to the distinct type before the value is actually assigned to the column. (There is no casting of the previous value to the source type before the computation.)
- To have a generated value on a SET statement for an identity column, use the DEFAULT keyword:

  ```
  SET NEW.EMPNO = DEFAULT
  ```

  In this example, NEW.EMPNO is defined as an identity column, and the value used to update this column is generated.
- For more information about consuming values of a generated sequence for an identity column, or about exceeding the maximum value for an identity column, see "INSERT".
- With partitioned tables, an UPDATE WHERE CURRENT OF *cursor-name* operation can move a row from one data partition to another. After this occurs, the cursor is no longer positioned on the row, and no further UPDATE WHERE CURRENT OF *cursor-name* modifications to that row are possible. The next row in the cursor can be fetched, however.
- For a column defined using the ROW CHANGE TIMESTAMP clause, the value is always changed on update of the row. If the column is not specified in the SET list explicitly, the database manager still generates a value for that row. The value is unique for each table partition within the database partition and is set to the approximate timestamp corresponding to the row update.
- **Extended indicator variables and update triggers**: If a target column has been assigned with an extended indicator variable-based value of unassigned, that column is not considered to have been updated. That column is treated as if it had not been specified in the OF *column-name* list of any update trigger defined on the target table.
- **Extended indicator variables and deferred error checks**: When extended indicator variables are enabled, validation that would otherwise be done in statement preparation, to recognize an update of a non-updatable column, is deferred until statement execution, except for column level update privilege checking of static UPDATE statements. Whether an error should be reported can be determined only during execution based on the indicator value. The checking of column level update privilege for static UPDATE statements continues to be performed during bind processing even when extended indicator variables are enabled.

- **Considerations for a system-period temporal table:** The target of the UPDATE statement must not be a fullselect that references a view in the FROM clause followed by a period specification for SYSTEM_TIME if the view is defined with the WITH CHECK OPTION and the view definition includes a WHERE clause containing one of the following syntax elements (SQLSTATE 51046):
  - A subquery that references a system-period temporal table (directly or indirectly)
  - An invocation of an SQL routine that has a package associated with it
  - An invocation of an external routine with a data access indication other than NO SQL

  If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a non-null value, an underlying target of the UPDATE statement must not be a system-period temporal table (SQLSTATE 51046), and the target of the UPDATE statement must not be a view defined with the WITH CHECK OPTION if the view definition includes a WHERE clause containing one of the following syntax elements (SQLSTATE 51046):
  - A subquery that references a system-period temporal table (directly or indirectly)
  - An invocation of an SQL routine that has a package associated with it
  - An invocation of an external routine with a data access indication other than NO SQL

  When a row of a system-period temporal table is updated, the database manager updates the values of the row-begin and transaction-start-ID columns as follows:
  - A row-begin column is assigned a value that is generated using a reading of the time-of-day clock during execution of the first data change statement in the transaction that requires a value to be assigned to the row begin or transaction start-ID column in a table, or a row in a system-period temporal table is deleted. The database manager ensures uniqueness of the generated values for a row-begin column across transactions. The timestamp value might be adjusted to ensure that rows inserted into an associated history table have the end timestamp value greater than the begin timestamp value which can happen when a conflicting transaction is updating the same row in the system-period temporal table. The database configuration parameter **systime_period_adj** must be set to Yes for this adjustment in the timestamp value to occur. If multiple rows are updated within a single SQL transaction and an adjustment is not needed, the values for the row-begin column are the same for all the rows and are unique from the values generated for the column for another transaction.
  - A transaction start-ID column is assigned a unique timestamp value per transaction or the null value The null value is assigned to the transaction start-ID column if the column is nullable and there is a row-begin column in the table for which the value did not need to be adjusted. Otherwise, the value is generated using a reading of the time-of-day clock during execution of the first data change statement in the transaction that requires a value to be assigned to the row begin or transaction start-ID column in a table, or a row in a system-period temporal table is deleted. If multiple rows are updated within a single SQL transaction, the values for the transaction start-ID column are the same for all the rows and are unique from the values generated for the column for another transaction.

  If the UPDATE statement has a search condition containing a correlated subquery that references historical rows (explicitly referencing the name of the history table name or implicitly through the use of a period specification in the FROM clause), the old version of the updated rows that are inserted as historical rows (into the history table if any) are potentially visible to update operations for the rows subsequently processed for the statement.

  The target of an UPDATE statement cannot be a fullselect that references a view in the FROM clause followed by a period specification for SYSTEM_TIME if both of the following conditions are true (SQLSTATE 51046):
  - The view is defined with the WITH CHECK OPTION.
  - The view definition includes a WHERE clause containing one of the following syntax elements:
    - A subquery that references a system-period temporal table (directly or indirectly).
    - An invocation of an SQL routine that has a package associated with it.
    - An invocation of an external routine with a data access indication other than NO SQL.

If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a non-null value, the underlying target (direct or indirect) of the UPDATE statement cannot be a system-period temporal table (SQLSTATE 51046).

If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a non-null value, the target of an UPDATE statement cannot be a view defined with the WITH CHECK OPTION if the view definition includes a WHERE clause containing one of the following syntax elements (SQLSTATE 51046):

- A subquery that references a system-period temporal table (directly or indirectly).
- An invocation of an SQL routine that has a package associated with it.
- An invocation of an external routine with a data access indication other than NO SQL.

- **Considerations for a history table:** When a row of a system-period temporal table is updated, a historical copy of the row is inserted into the corresponding history table and the end timestamp of the historical row is captured in the form of a system determined value that corresponds to the time of the data change operation. The database manager assigns the value that is generated using a reading of the time-of-day clock during execution of the first data change statement in the transaction that requires a value to be assigned to the row begin or transaction start-ID column in a table, or a row in a system-period temporal table is deleted. The database manager ensures uniqueness of the generated values for an end column in a history table across transactions. The timestamp value might be adjusted to ensure that rows inserted into the history table have the end timestamp value greater than the begin timestamp value which can happen when a conflicting transaction is updating the same row in the system-period temporal table (SQLSTATE 01695). The database configuration parameter **systime_period_adj** must be set to Yes for this adjustment in the timestamp value to occur.

  For an update operation, the adjustment only affects the value for the end column corresponding to the row-end column in the history table associated with the system-period temporal table. Take these adjustments into consideration on subsequent references to the table whether there is a search for the transaction start time in the values for the columns corresponding to the row-begin and row-end columns of the period in the associated system-period temporal table.
- **Considerations for an application-period temporal table:** The target of the UPDATE statement must not be a fullselect that references a view in the FROM clause followed by a period specification for BUSINESS_TIME if the view is defined with the WITH CHECK OPTION and the view definition includes a WHERE clause containing one of the following syntax elements (SQLSTATE 51046):

  - A subquery that references an application-period temporal table (directly or indirectly)
  - An invocation of an SQL routine that has a package associated with it
  - An invocation of an external routine with a data access indication other than NO SQL

  If the CURRENT TEMPORAL BUSINESS_TIME special register is set to a non-null value, the target of the UPDATE statement must not be a view defined with the WITH CHECK option if the view definition includes a WHERE clause containing one of the following syntax elements (SQLSTATE 51046):

  - A subquery that references an application-period temporal table (directly or indirectly)
  - An invocation of an SQL routine that has a package associated with it
  - An invocation of an external routine with a data access indication other than NO SQL

  An UPDATE statement for an application-period temporal table that contains a FOR PORTION OF BUSINESS_TIME clause indicates between which two points in time that the specified updates are effective. When FOR PORTION OF BUSINESS_TIME is specified and the period value for a row, specified by the values of the row-begin column and row-end column, is only partially contained in the period specified from *value1* up to *value2*, the row is updated and one or two rows are automatically inserted to represent the portion of the row that is not changed. New values are generated for each generated column in an application-period temporal table for each row that is automatically inserted as a result of an update operation on the table. If a generated column is defined as part of a unique or primary key, parent key in a referential constraint, or unique index, it is possible that an automatic insert will violate a constraint or index in which case an error is returned.

  When a row is inserted into an application-period temporal table that has either a primary key or unique constraint with the BUSINESS_TIME WITHOUT OVERLAPS clause defined, or a unique index with

the BUSINESS_TIME WITHOUT OVERLAPS clause defined, if the period defined by the begin and end columns of the BUSINESS_TIME period overlap the period defined by the begin and end columns of the BUSINESS_TIME period for another row with the same unique constraint or unique index in the table, an error is returned.

The target of an UPDATE statement cannot be a fullselect that references a view in the FROM clause followed by a period specification for BUSINESS_TIME if both of the following conditions are true (SQLSTATE 51046):

– The view is defined with the WITH CHECK OPTION.
– The view definition includes a WHERE clause containing one of the following syntax elements:
  - A subquery that references an application-period temporal table (directly or indirectly).
  - An invocation of an SQL routine that has a package associated with it.
  - An invocation of an external routine with a data access indication other than NO SQL.

If the CURRENT TEMPORAL BUSINESS_TIME special register is set to a non-null value, the target of an UPDATE statement cannot be a view defined with the WITH CHECK OPTION if the view definition includes a WHERE clause containing one of the following syntax elements (SQLSTATE 51046):

– A subquery that references an application-period temporal table (directly or indirectly).
– An invocation of an SQL routine that has a package associated with it.
– An invocation of an external routine with a data access indication other than NO SQL.

When an application-period temporal table is the target of an UPDATE statement, the value in effect for the CURRENT TEMPORAL BUSINESS_TIME special register is not the null value, and the BUSTIMESENSITIVE bind option is set to YES, the following additional predicates are implicit:

```
    bt_begin <= CURRENT TEMPORAL BUSINESS_TIME
AND bt_end > CURRENT TEMPORAL BUSINESS_TIME
```

where `bt_begin` and `bt_end` are the begin and end columns of the BUSINESS_TIME period of the target table of the UPDATE statement.

- **Considerations for application-period temporal tables and triggers:** When a row is updated and the FOR PORTION OF BUSINESS_TIME clause is specified, additional rows may be implicitly inserted to reflect any portion of the row that was not updated. Any existing update triggers are activated for the rows updated, and any existing insert triggers are activated for rows that are implicitly inserted.

Examples

- *Example 1:* Change the job (JOB) of employee number (EMPNO) '000290' in the EMPLOYEE table to 'LABORER'.

```
UPDATE EMPLOYEE
   SET JOB = 'LABORER'
   WHERE EMPNO = '000290'
```

- *Example 2:* Increase the project staffing (PRSTAFF) by 1.5 for all projects that department (DEPTNO) 'D21' is responsible for in the PROJECT table.

```
UPDATE PROJECT
   SET PRSTAFF = PRSTAFF + 1.5
   WHERE DEPTNO = 'D21'
```

- *Example 3:* All the employees except the manager of department (WORKDEPT) 'E21' have been temporarily reassigned. Indicate this by changing their job (JOB) to the null value and their pay (SALARY, BONUS, COMM) values to zero in the EMPLOYEE table.

```
UPDATE EMPLOYEE
  SET JOB=NULL, SALARY=0, BONUS=0, COMM=0
  WHERE WORKDEPT = 'E21' AND JOB <> 'MANAGER'
```

This statement could also be written as follows.

```
UPDATE EMPLOYEE
  SET (JOB, SALARY, BONUS, COMM) = (NULL, 0, 0, 0)
  WHERE WORKDEPT = 'E21' AND JOB <> 'MANAGER'
```

- *Example 4:* Update the salary and the commission column of the employee with employee number 000120 to the average of the salary and of the commission of the employees of the updated row's department, respectively.

```
UPDATE (SELECT EMPNO, SALARY, COMM,
  AVG(SALARY) OVER (PARTITION BY WORKDEPT),
  AVG(COMM) OVER (PARTITION BY WORKDEPT)
  FROM EMPLOYEE E) AS E(EMPNO, SALARY, COMM, AVGSAL, AVGCOMM)
SET (SALARY, COMM) = (AVGSAL, AVGCOMM)
WHERE EMPNO = '000120'
```

  The previous statement is semantically equivalent to the following statement, but requires only one access to the EMPLOYEE table, whereas the following statement specifies the EMPLOYEE table twice.

```
UPDATE EMPLOYEE EU
  SET (EU.SALARY, EU.COMM)
  =
(SELECT AVG(ES.SALARY), AVG(ES.COMM)
  FROM EMPLOYEE ES
  WHERE ES.WORKDEPT = EU.WORKDEPT)
  WHERE EU.EMPNO = '000120'
```

- *Example 5:* In a C program display the rows from the EMPLOYEE table and then, if requested to do so, change the job (JOB) of certain employees to the new job keyed in.

```
EXEC SQL  DECLARE C1 CURSOR FOR
               SELECT *
                 FROM EMPLOYEE
                 FOR UPDATE OF JOB;

EXEC SQL  OPEN C1;

EXEC SQL  FETCH C1 INTO ...     ;
if ( strcmp (change, "YES") == 0 )
  EXEC SQL  UPDATE EMPLOYEE
              SET JOB = :newjob
              WHERE CURRENT OF C1;

EXEC SQL  CLOSE C1;
```

- *Example 6:* These examples mutate attributes of column objects.

  Assume that the following types and tables exist:

```
CREATE TYPE POINT AS (X INTEGER, Y INTEGER)
  NOT FINAL WITHOUT COMPARISONS
  MODE DB2SQL
```

```
CREATE TYPE CIRCLE AS (RADIUS INTEGER, CENTER POINT)
  NOT FINAL WITHOUT COMPARISONS
  MODE DB2SQL
```

```
CREATE TABLE CIRCLES (ID INTEGER, OWNER VARCHAR(50), C CIRCLE
```

  The following example updates the CIRCLES table by changing the OWNER column and the RADIUS attribute of the CIRCLE column where the ID is 999:

```
UPDATE CIRCLES
  SET OWNER = 'Bruce'
    C..RADIUS = 5
  WHERE ID = 999
```

  The following example transposes the X and Y coordinates of the center of the circle identified by 999:

```
UPDATE CIRCLES
  SET C..CENTER..X = C..CENTER..Y,
```

```
    C..CENTER..Y = C..CENTER..X
  WHERE ID = 999
```

The following example is another way of writing both of the previous statements. This example combines the effects of both of the previous examples:

```
UPDATE CIRCLES
  SET (OWNER,C..RADIUS,C..CENTER..X,C..CENTER..Y) =
    ('Bruce',5,C..CENTER..Y,C..CENTER..X)
  WHERE ID = 999
```

- *Example 7:* Update the XMLDOC column of the DOCUMENTS table with DOCID '001' to the character string that is selected and parsed from the XMLTEXT table.

```
UPDATE DOCUMENTS SET XMLDOC =
  (SELECT XMLPARSE(DOCUMENT C1 STRIP WHITESPACE)
    FROM XMLTEXT WHERE TEXTID = '001')
WHERE DOCID = '001'
```

- *Example 8:* A new location column has been added to the project table. Update the project location with the location of the department handling the project.

```
UPDATE PROJECT P
   SET P.LOCATION = D.LOCATION
   FROM DEPARTMENT D
   WHERE P.DEPTNO = D.DEPTNO;
```

- *Example 9:* Update the estimated project staffing to the max staffing required for all activities within the project.

```
UPDATE PROJECT P
   SET P.PRSTAFF = S.ACSTAFF
   FROM (SELECT PROJNO, MAX(ACSTAFF) ACSTAFF FROM PROJACT GROUP BY PROJNO) S
   WHERE P.PROJNO = S.PROJNO AND
         P.PROJNAME = 'PAYROLL PROGRAMMING';
```

- *Example 10:* Update an employee's work department to the project department he is assigned to.

```
UPDATE EMPLOYEE E
   SET E.WORKDEPT = P.DEPTNO
   FROM PROJECT P JOIN EMPPROJACT EP ON P.PROJNO = EP.PROJNO
   WHERE E.EMPNO = EP.EMPNO AND
         E.FIRSTNME = 'PHILIP' AND E.LASTNAME = 'SMITH';
```

VALUES

The VALUES statement is a form of query.

The VALUES statement can be embedded in an application program or issued interactively.

VALUES INTO

The VALUES INTO statement produces a result table consisting of at most one row, and assigns the values in that row to host variables.

Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

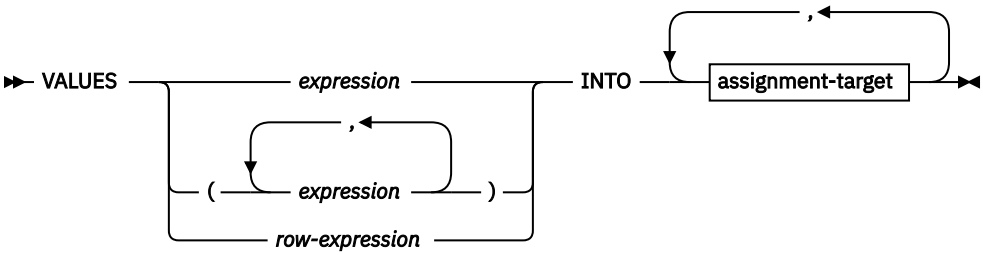
Authorization

The privileges held by the authorization ID of the statement must include any privileges that are necessary to execute each *expression* and *row-expression*.

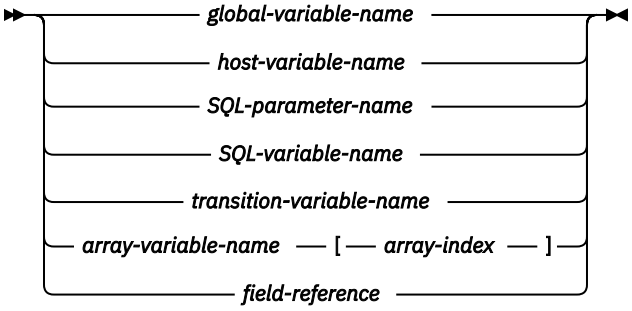
For each global variable used as an *assignment-target*, the privileges held by the authorization ID of the statement must include one of the following authorities:

- WRITE privilege on the global variable that is not defined in a module
- EXECUTE privilege on the module of the global variable that is defined in a module
- EXECUTEIN privilege on the schema containing the module of the global variable that is defined in a module
- DATAACCESS authority on the schema containing the module of the global variable that is defined in a module

Syntax



assignment-target



Description

VALUES

Introduces a single row consisting of one or more columns.

expression

An expression that defines a single value of a one column result table.

(expression,...)

One or more expressions that define the values for one or more columns of the result table.

row-expression

Specifies the new row of values. The *row-expression* is any row expression of the type described in "Row expressions". The *row-expression* must not include a column name.

INTO *assignment-target*

Identifies one or more targets for the assignment of output values.

The first value in the result row is assigned to the first target in the list, the second value to the second target, and so on. Each assignment to an *assignment-target* is made in sequence through the list. If an error occurs on any assignment, no value is assigned to any *assignment-target*.

When the data type of every *assignment-target* is not a row type, then the value 'W' is assigned to the SQLWARN3 field of the SQLCA if the number of *assignment-targets* is less than the number of result column values.

If the data type of an *assignment-target* is a row type, then there must be exactly one *assignment-target* specified (SQLSTATE 428HR), the number of columns must match the number of fields in the row type, and the data types of the columns of the fetched row must be assignable to the corresponding fields of the row type (SQLSTATE 42821).

If the data type of an *assignment-target* is an array element, then there must be exactly one *assignment-target* specified.

global-variable-name
Identifies the global variable that is the assignment target.

host-variable-name
Identifies the host variable that is the assignment target. For LOB output values, the target can be a regular host variable (if it is large enough), a LOB locator variable, or a LOB file reference variable.

SQL-parameter-name
Identifies the name parameter that is the assignment target.

SQL-variable-name
Identifies the SQL variable that is the assignment target. SQL variables must be declared before they are used.

transition-variable-name
Identifies the column to be updated in the transition row. A *transition-variable-name* must identify a column in the subject table of a trigger, optionally qualified by a correlation name that identifies the new value.

array-variable-name
Identifies an SQL variable, SQL parameter, or global variable of an array type.

[array-index]
An expression that specifies which element in the array will be the target of the assignment. For an ordinary array, the *array-index* expression must be assignable to INTEGER (SQLSTATE 428H1) and cannot be the null value. Its value must be between 1 and the maximum cardinality defined for the array (SQLSTATE 2202E). For an associative array, the *array-index* expression must be assignable to the index data type of the associative array (SQLSTATE 428H1) and cannot be the null value.

field-reference
Identifies the field within a row type value that is the assignment target. The *field-reference* must be specified as a qualified *field-name* where the qualifier identifies the row value in which the field is defined.

Rules

- Global variables cannot be assigned inside triggers that are not defined using a compound SQL (compiled) statement, functions that are not defined using a compound SQL (compiled) statement, methods, or compound SQL (inlined) statements (SQLSTATE 428GX).

Examples

- *Example 1:* This C example retrieves the value of the CURRENT PATH special register into a host variable.

```
EXEC SQL VALUES(CURRENT PATH)
         INTO :hvl;
```

- *Example 2:* This C example retrieves a portion of a LOB field into a host variable, exploiting the LOB locator for deferred retrieval.

```
EXEC SQL VALUES (substr(:locator1,35))
         INTO :details;
```

- *Example 3:* This C example retrieves the value of the SESSION_USER special register into a global variable.

```
EXEC SQL VALUES(SESSION_USER)
  INTO GV_SESS_USER;
```

WHENEVER

The WHENEVER statement specifies the action to be taken when a specified exception condition occurs.

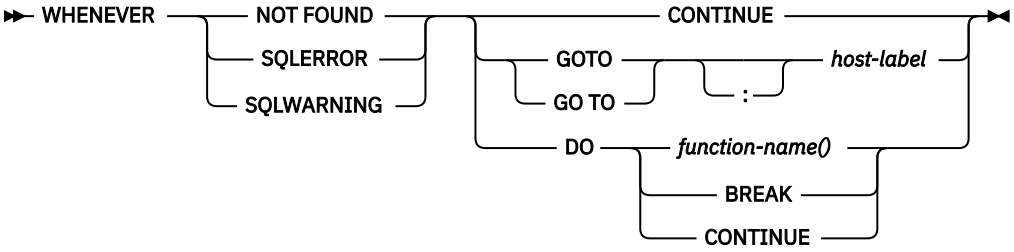
Invocation

This statement can only be embedded in an application program. It is not an executable statement. The statement is not supported in REXX.

Authorization

None required.

Syntax



Description

The NOT FOUND, SQLERROR, or SQLWARNING clause is used to identify the type of exception condition.

NOT FOUND
Identifies any condition that results in an SQLCODE of +100 or an SQLSTATE of '02000'.

SQLERROR
Identifies any condition that results in a negative SQLCODE.

SQLWARNING
Identifies any condition that results in a warning condition (SQLWARN0 is 'W'), or that results in a positive SQL return code other than +100.

The CONTINUE or GO TO clause is used to specify what is to happen when the identified type of exception condition exists.

CONTINUE
Causes the next sequential instruction of the source program to be executed.

GOTO or GO TO *host-label*
Causes control to pass to the statement identified by host-label. For host-label, substitute a single token, optionally preceded by a colon. The form of the token depends on the host language.

DO
Causes additional action in the form of a function call, `break` statement, or `continue` statement to take place.

function-name()
Specifies the C function that is to be called. The function must have a void return value and cannot accept any arguments. The function name must end with set of parentheses "(" and ")". The name of the function is limited to 255 bytes.

The function name resolution takes place during the compilation of the C and C++ embedded SQL application. The database precompiler does not resolve the function name.

BREAK
Specifies the C `break` statement. The C `break` statement exits the`do`, `for`, `switch`, or `while` statement block.

CONTINUE
Specifies the C `continue` statement. The C `continue` statement passes control to the next iteration of the do, `for`, `switch`, or `while` statement block.

Notes

There are three types of WHENEVER statements:

- WHENEVER NOT FOUND
- WHENEVER SQLERROR
- WHENEVER SQLWARNING

Every executable SQL statement in a program is within the scope of one implicit or explicit WHENEVER statement of each type. The scope of a WHENEVER statement is related to the listing sequence of the statements in the program, not their execution sequence.

An SQL statement is within the scope of the last WHENEVER statement of each type that is specified before that SQL statement in the source program. If a WHENEVER statement of some type is not specified before an SQL statement, that SQL statement is within the scope of an implicit WHENEVER statement of that type in which CONTINUE is specified.

If the WHENEVER statement is not used, the default action is to continue processing if an error, warning, or exception condition occurs during execution.

The WHENEVER statement must be used before the SQL statements that you want to affect. Otherwise, the precompiler does not know that additional error-handling code is required for the executable SQL statements. You can have any combination of the three basic forms active at any time. The order in which you declare the three forms is not significant.

To avoid an infinite looping situation, ensure that you undo the WHENEVER handling before any SQL statements are executed inside the handler. You can undo the WHENEVER handling by using the WHENEVER SQLERROR CONTINUE statement.

The WHENEVER statement support for use of the DO *function-name()*, DO BREAK, or DO CONTINUE syntax is available in Version 9.7 Fix Pack 6 and later.

Example

In the following C example, if an error is produced, go to HANDLERR. If a warning code is produced, continue with the normal flow of the program. If no data is returned, go to ENDDATA.

```
   EXEC SQL WHENEVER SQLERROR GOTO HANDLERR;
   EXEC SQL WHENEVER SQLWARNING CONTINUE;
   EXEC SQL WHENEVER NOT FOUND GO TO ENDDATA;
```

The C example for use of the DO *function-name()*, DO BREAK, or DO CONTINUE syntax are:

```
/* DO function_name */
EXEC SQL WHENEVER SQLERROR DO perform_error_action();
EXEC SQL WHENEVER SQLWARNING DO perform_warning_action();
EXEC SQL WHENEVER NOT FOUND DO perform_notfound_action();

/* DO BREAK */
EXEC SQL WHENEVER SQLERROR DO BREAK;
EXEC SQL WHENEVER SQLWARNING DO BREAK;
EXEC SQL WHENEVER NOT FOUND DO BREAK;

/* DO CONTINUE */
EXEC SQL WHENEVER SQLERROR DO CONTINUE;
EXEC SQL WHENEVER SQLWARNING DO CONTINUE;
EXEC SQL WHENEVER NOT FOUND DO CONTINUE;
```

WHILE

The WHILE statement repeats the execution of a statement or group of statements while a specified condition is true.

Invocation

This statement can be embedded in an:

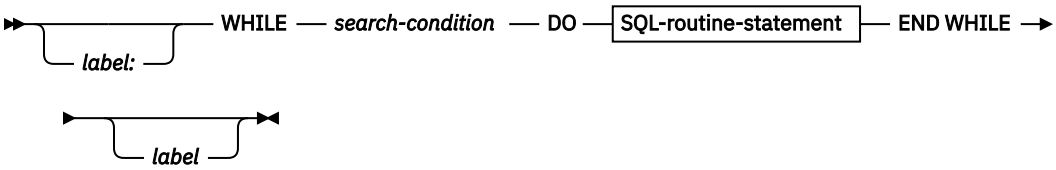
- SQL procedure definition
- Compound SQL (compiled) statement
- Compound SQL (inlined) statement

The compound statements can be embedded in an SQL procedure definition, SQL function definition, or SQL trigger definition. It is not an executable statement and cannot be dynamically prepared.

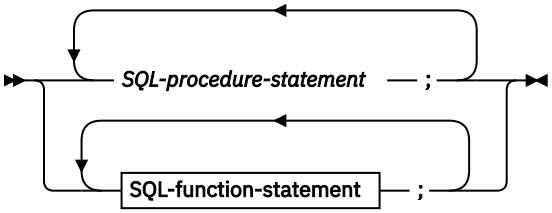
Authorization

No privileges are required to invoke the WHILE statement. However, the authorization ID of the statement must hold the necessary privileges to invoke the SQL statements and search condition that are embedded in the WHILE statement.

Syntax



SQL-routine-statement



Description

label
Specifies the label for the WHILE statement. If the beginning label is specified, it can be specified in LEAVE and ITERATE statements. If the ending label is specified, it must be the same as the beginning label.

search-condition
Specifies a condition that is evaluated before each execution of the loop. If the condition is true, the SQL-procedure-statements in the loop are processed.

SQL-procedure-statement
Specifies the SQL statements to execute within the loop. *SQL-procedure-statement* is only applicable when in the context of an SQL procedure or compound SQL (compiled) statement. See *SQL-procedure-statement* in "Compound SQL (compiled)" statement.

SQL-function-statement
Specifies the SQL statements to execute within the loop. *SQL-function-statement* is only applicable in an SQL function or a compound SQL (inlined) statement which can be embedded in an SQL trigger, SQL function or SQL method. See *SQL-function-statement* in "FOR".

Example

This example uses a WHILE statement to iterate through FETCH and SET statements. While the value of SQL variable *v_counter* is less than half of number of employees in the department identified by the IN parameter *deptNumber*, the WHILE statement continues to perform the FETCH and SET statements. When the condition is no longer true, the flow of control leaves the WHILE statement and closes the cursor.

```
CREATE PROCEDURE DEPT_MEDIAN
  (IN deptNumber SMALLINT, OUT medianSalary DOUBLE)
  LANGUAGE SQL
  BEGIN
    DECLARE v_numRecords INTEGER DEFAULT 1;
    DECLARE v_counter INTEGER DEFAULT 0;
    DECLARE c1 CURSOR FOR
      SELECT CAST(salary AS DOUBLE)
        FROM staff
        WHERE DEPT = deptNumber
        ORDER BY salary;
    DECLARE EXIT HANDLER FOR NOT FOUND
      SET medianSalary = 6666;
    SET medianSalary = 0;
    SELECT COUNT(*) INTO v_numRecords
      FROM staff
      WHERE DEPT = deptNumber;
    OPEN c1;
    WHILE v_counter < (v_numRecords / 2 + 1) DO
      FETCH c1 INTO medianSalary;
      SET v_counter = v_counter + 1;
    END WHILE;
    CLOSE c1;
  END
```

Catalog views

The database manager creates and maintains two sets of catalog views that are defined on top of the base system catalog tables.

- SYSCAT views are read-only catalog views that are found in the SYSCAT schema. The RESTRICT option on CREATE DATABASE statement determines how SELECT privilege is granted. When the RESTRICT option is not specified, SELECT privilege is granted to PUBLIC.
- SYSSTAT views are updatable catalog views that are found in the SYSSTAT schema. The updatable views contain statistical information that is used by the optimizer. The values in some columns in these views can be changed to test performance. (Before changing any statistics, it is recommended that the RUNSTATS command be invoked so that all the statistics reflect the current state.)

Applications should be written to the SYSCAT and SYSSTAT views rather than the base catalog tables.

All the catalog views are created at database creation time. The catalog views cannot be explicitly created or dropped. In a Unicode database, the catalog views are created with IDENTITY collation. In non-Unicode databases, the catalog views are created with the database collation. The views are updated during normal operation in response to SQL data definition statements, environment routines, and certain utilities. Data in the catalog views is available through normal SQL query facilities. The catalog views (with the exception of some updatable catalog views) cannot be modified using normal SQL data manipulation statements.

An object table, statistical view, column, or index object appears in a user's updatable SYSSTAT catalog view only if that user holds explicit CONTROL privilege on the object, or holds explicit DATAACCESS authority. An object table or statistical view also appears in a user's updatable SYSSTAT.TABLES catalog view if the user is a direct or indirect member of a role that has CONTROL privilege on the object, or a role that has DATAACCESS authority. Role privileges and authorities are not considered when determining the objects that appear in the other SYSSTAT catalog views. A routine object appears in a user's updatable SYSSTAT.ROUTINES catalog view if that user owns the routine or holds explicit SQLADM authority. Group privileges and authorities are not considered when determining the objects that appear in a user's updatable SYSSTAT catalog views.

The order of columns in the views may change from release to release. To prevent this from affecting programming logic, specify the columns in a select list explicitly, and avoid using SELECT *. Columns have consistent names based on the types of objects that they describe.

Table 155. Samples of consistent column names for objects they describe

Described Object	**Column Names**
Table	TABSCHEMA, TABNAME
Index	INDSCHEMA, INDNAME
Index extension	IESCHEMA, IENAME
View	VIEWSCHEMA, VIEWNAME
Constraint	CONSTSCHEMA, CONSTNAME
Control	CONTROLSCHEMA, CONTROLNAME, CONTROLID
Trigger	TRIGSCHEMA, TRIGNAME
Package	PKGSCHEMA, PKGNAME
Type	TYPESCHEMA, TYPENAME, TYPEID
Function	ROUTINESCHEMA, ROUTINEMODULENAME, ROUTINENAME, ROUTINEID
Method	ROUTINESCHEMA, ROUTINEMODULENAME, ROUTINENAME, ROUTINEID
Procedure	ROUTINESCHEMA, ROUTINEMODULENAME, ROUTINENAME, ROUTINEID
Column	COLNAME
Schema	SCHEMANAME
Table Space	TBSPACE
Database partition group	DBPGNAME
Audit policy	AUDITPOLICYNAME, AUDITPOLICYID
Buffer pool	BPNAME
Event Monitor	EVMONNAME
Condition	CONDSCHEMA, CONDMODULENAME, CONDNAME, CONDMODULEID
Data source	SERVERNAME, SERVERTYPE, SERVERVERSION
Global variable	VARSCHEMA, VARMODULENAME, VARNAME, VARMODULEID
Histogram template	TEMPLATENAME, TEMPLATEID
Module	MODULESCHEMA, MODULENAME, MODULEID
Period	PERIODNAME
Role	ROLENAME, ROLEID
Security label	SECLABELNAME, SECLABELID
Security policy	SECPOLICYNAME, SECPOLICYID
Sequence	SEQSCHEMA, SEQNAME
Threshold	THRESHOLDNAME, THRESHOLDID

Table 155. Samples of consistent column names for objects they describe (continued)	
Described Object	**Column Names**
Trusted context	CONTEXTNAME, CONTEXTID
Usage list	USAGELISTSCHEMA, USAGELISTNAME, USAGELISTID
Work action	ACTIONNAME, ACTIONID
Work action set	ACTIONSETNAME, ACTIONSETID
Work class	WORKCLASSNAME, WORKCLASSID
Work class set	WORKCLASSSETNAME, WORKCLASSSETID
Workload	WORKLOADID, WORKLOADNAME
Wrapper	WRAPNAME
Alteration Timestamp	ALTER_TIME
Creation Timestamp	CREATE_TIME

Road map to the catalog views

This topic lists the catalog views, grouped by object or functionality.

Table 156. Road map to the read-only catalog views

Description	Catalog View
attributes of structured data types	“SYSCAT.ATTRIBUTES ” on page 1934
audit policies	“SYSCAT.AUDITPOLICIES ” on page 1936 “SYSCAT.AUDITUSE ” on page 1938
authorities on database	“SYSCAT.DBAUTH ” on page 1967
buffer pool configuration on database partition group	“SYSCAT.BUFFERPOOLS ” on page 1939
buffer pool size exceptions for database partitions	“SYSCAT.BUFFERPOOLDBPARTITIONS ” on page 1938
buffer pool size exceptions for members	“SYSCAT.BUFFERPOOLEXCEPTIONS ” on page 1939
cast functions	“SYSCAT.CASTFUNCTIONS ” on page 1940
check constraints	“SYSCAT.CHECKS ” on page 1941
column masks	“SYSCAT.CONTROLS ” on page 1957
column mask dependences	“SYSCAT.CONTROLDEP ” on page 1956
column privileges	“SYSCAT.COLAUTH ” on page 1942
columns	“SYSCAT.COLUMNS ” on page 1947
columns referenced by check constraints	“SYSCAT.COLCHECKS ” on page 1943
columns used in dimensions	“SYSCAT.COLUSE ” on page 1953

Table 156. Road map to the read-only catalog views (continued)

Description	Catalog View
columns used in keys	“SYSCAT.KEYCOLUSE ” on page 2001
conditions	“SYSCAT.CONDITIONS ” on page 1954
constraint dependencies	“SYSCAT.CONSTDEP ” on page 1954
controls	“SYSCAT.CONTROLS ” on page 1957
database partition group database partitions	“SYSCAT.DBPARTITIONGROUPDEF ” on page 1969
database partition group definitions	“SYSCAT.DBPARTITIONGROUPS ” on page 1970
data partitions	“SYSCAT.DATAPARTITIONEXPRESSION ” on page 1959 “SYSCAT.DATAPARTITIONS ” on page 1959
data type dependencies	“SYSCAT.DATATYPEDEP ” on page 1962
data types	“SYSCAT.DATATYPES ” on page 1963
detailed column group statistics	“SYSCAT.COLGROUPCOLS ” on page 1944 “SYSCAT.COLGROUPDIST ” on page 1945 “SYSCAT.COLGROUPDISTCOUNTS ” on page 1945 “SYSCAT.COLGROUPS ” on page 1946
detailed column options	“SYSCAT.COLOPTIONS ” on page 1947
detailed column statistics	“SYSCAT.COLDIST ” on page 1943
distribution maps	“SYSCAT.PARTITIONMAPS ” on page 2020
event monitor definitions	“SYSCAT.EVENTMONITORS ” on page 1971
events currently monitored	“SYSCAT.EVENTS ” on page 1973 “SYSCAT.EVENTTABLES ” on page 1973
external tables	“SYSCAT.EXTERNALTABLEOPTIONS ” on page 1975
fields of row data types	“SYSCAT.ROWFIELDS ” on page 2043
function dependencies[1]	“SYSCAT.ROUTINEDEP ” on page 2025
function mapping	“SYSCAT.FUNCMAPPINGS ” on page 1978
function mapping options	“SYSCAT.FUNCMAPOPTIONS ” on page 1978
function parameter mapping options	“SYSCAT.FUNCMAPPARMOPTIONS ” on page 1978
function parameters[1]	“SYSCAT.ROUTINEPARMS ” on page 2028
functions[1]	“SYSCAT.ROUTINES ” on page 2030
global variables	“SYSCAT.VARIABLEAUTH ” on page 2092 “SYSCAT.VARIABLEDEP ” on page 2093 “SYSCAT.VARIABLES ” on page 2094
hierarchies (types, tables, views)	“SYSCAT.HIERARCHIES ” on page 1979 “SYSCAT.FULLHIERARCHIES ” on page 1977
identity columns	“SYSCAT.COLIDENTATTRIBUTES ” on page 1946
index columns	“SYSCAT.INDEXCOLUSE ” on page 1982
index data partitions	“SYSCAT.INDEXPARTITIONS ” on page 1996

Table 156. Road map to the read-only catalog views (continued)

Description	Catalog View
index dependencies	“SYSCAT.INDEXDEP ” on page 1983
index exploitation	“SYSCAT.INDEXEXPLOITRULES ” on page 1992
index extension dependencies	“SYSCAT.INDEXEXTENSIONDEP ” on page 1993
index extension parameters	“SYSCAT.INDEXEXTENSIONPARMS ” on page 1994
index extension search methods	“SYSCAT.INDEXEXTENSIONMETHODS ” on page 1994
index extensions	“SYSCAT.INDEXEXTENSIONS ” on page 1995
index options	“SYSCAT.INDEXOPTIONS ” on page 1996
index privileges	“SYSCAT.INDEXAUTH ” on page 1981
indexes	“SYSCAT.INDEXES ” on page 1985
invalid objects	“SYSCAT.INVALIDOBJECTS ” on page 2000
member susbsets	“SYSCAT.MEMBERSUBSETATTRS ” on page 2001 “SYSCAT.MEMBERSUBSETMEMBERS ” on page 2002 “SYSCAT.MEMBERSUBSETS ” on page 2002
method dependencies[1]	“SYSCAT.ROUTINEDEP ” on page 2025
method parameters[1]	“SYSCAT.ROUTINES ” on page 2030
methods[1]	“SYSCAT.ROUTINES ” on page 2030
module objects	“SYSCAT.MODULEOBJECTS ” on page 2003
module privileges	“SYSCAT.MODULEAUTH ” on page 2003
modules	“SYSCAT.MODULES ” on page 2004
nicknames	“SYSCAT.NICKNAMES ” on page 2005
object mapping	“SYSCAT.NAMEMAPPINGS ” on page 2005
package dependencies	“SYSCAT.PACKAGEDEP ” on page 2009
package privileges	“SYSCAT.PACKAGEAUTH ” on page 2008
packages	“SYSCAT.PACKAGES ” on page 2011
partitioned tables	“SYSCAT.TABDETACHEDDEP ” on page 2066
pass-through privileges	“SYSCAT.PASSTHRUAUTH ” on page 2021
periods	“SYSCAT.PERIODS ” on page 2021
predicate specifications	“SYSCAT.PREDICATESPECS ” on page 2021
procedure options	“SYSCAT.ROUTINEOPTIONS ” on page 2027
procedure parameter options	“SYSCAT.ROUTINEPARMOPTIONS ” on page 2027
procedure parameters[1]	“SYSCAT.ROUTINEPARMS ” on page 2028
procedures[1]	“SYSCAT.ROUTINES ” on page 2030

Table 156. Road map to the read-only catalog views (continued)

Description	Catalog View
protected tables	"SYSCAT.SECURITYLABELACCESS " on page 2048 "SYSCAT.SECURITYLABELCOMPONENTELEMENTS " on page 2049 "SYSCAT.SECURITYLABELCOMPONENTS " on page 2049 "SYSCAT.SECURITYLABELS " on page 2049 "SYSCAT.SECURITYPOLICIES " on page 2050 "SYSCAT.SECURITYPOLICYCOMPONENTRULES " on page 2051 "SYSCAT.SECURITYPOLICYEXEMPTIONS " on page 2051 "SYSCAT.SURROGATEAUTHIDS " on page 2061
provides Db2 for z/OS compatibility	"SYSIBM.SYSDUMMY1 " on page 2114
referential constraints	"SYSCAT.REFERENCES " on page 2022
remote table options	"SYSCAT.TABOPTIONS " on page 2078
roles	"SYSCAT.ROLEAUTH " on page 2023 "SYSCAT.ROLES " on page 2023
routine dependencies	"SYSCAT.ROUTINEDEP " on page 2025
routine parameters[1]	"SYSCAT.ROUTINEPARMS " on page 2028
routine privileges	"SYSCAT.ROUTINEAUTH " on page 2024
routines[1]	"SYSCAT.ROUTINES " on page 2030 "SYSCAT.ROUTINESFEDERATED " on page 2041
row permisssions	"SYSCAT.CONTROLS " on page 1957
row permission dependeencies	"SYSCAT.CONTROLDEP " on page 1956
schema privileges	"SYSCAT.SCHEMAAUTH " on page 2044
schemas	"SYSCAT.SCHEMATA " on page 2046
sequence privileges	"SYSCAT.SEQUENCEAUTH " on page 2052
sequences	"SYSCAT.SEQUENCES " on page 2052
server options	"SYSCAT.SERVEROPTIONS " on page 2055
server-specific user options	"SYSCAT.USEROPTIONS " on page 2091
statements	"SYSCAT.STATEMENTS " on page 2059 "SYSCAT.STATEMENTTEXTS " on page 2060
storage groups	"SYSCAT.STOGROUPS " on page 2060
procedures	"SYSCAT.ROUTINES " on page 2030
system servers	"SYSCAT.SERVERS " on page 2055
table constraints	"SYSCAT.TABCONST " on page 2063
table dependencies	"SYSCAT.TABDEP " on page 2064
table privileges	"SYSCAT.TABAUTH " on page 2061
table space use privileges	"SYSCAT.TBSPACEAUTH " on page 2079
table spaces	"SYSCAT.TABLESPACES " on page 2076
tables	"SYSCAT.TABLES " on page 2066
transforms	"SYSCAT.TRANSFORMS " on page 2082

Table 156. Road map to the read-only catalog views (continued)

Description	Catalog View
trigger dependencies	"SYSCAT.TRIGDEP " on page 2083
triggers	"SYSCAT.TRIGGERS " on page 2085
trusted contexts	"SYSCAT.CONTEXTATTRIBUTES " on page 1955 "SYSCAT.CONTEXTS " on page 1955
type mapping	"SYSCAT.TYPEMAPPINGS " on page 2087
usage lists	"SYSCAT.USAGELISTS " on page 2091
user-defined functions	"SYSCAT.ROUTINES " on page 2030
view dependencies	"SYSCAT.TABDEP " on page 2064
views	"SYSCAT.TABLES " on page 2066 "SYSCAT.VIEWS " on page 2096
workload management	"SYSCAT.HISTOGRAMTEMPLATEBINS " on page 1980 "SYSCAT.HISTOGRAMTEMPLATES " on page 1980 "SYSCAT.HISTOGRAMTEMPLATEUSE " on page 1981 "SYSCAT.SCPREFTBSPACES " on page 2047 "SYSCAT.SERVICECLASSES " on page 2055 "SYSCAT.THRESHOLDS " on page 2079 "SYSCAT.WORKACTIONS " on page 2097 "SYSCAT.WORKACTIONSETS " on page 2100 "SYSCAT.WORKCLASSATTRIBUTES " on page 2101 "SYSCAT.WORKCLASSES " on page 2103 "SYSCAT.WORKCLASSSETS " on page 2103 "SYSCAT.WORKLOADAUTH " on page 2103 "SYSCAT.WORKLOADCONNATTR " on page 2104 "SYSCAT.WORKLOADS " on page 2104
wrapper options	"SYSCAT.WRAPOPTIONS " on page 2108
wrappers	"SYSCAT.WRAPPERS " on page 2108
XML strings	"SYSCAT.XMLSTRINGS " on page 2109
Index on XML column	"SYSCAT.INDEXXMLPATTERNS " on page 1999
XSR objects	"SYSCAT.XDBMAPGRAPHS " on page 2109 "SYSCAT.XDBMAPSHREDTREES " on page 2109 "SYSCAT.XSROBJECTAUTH " on page 2110 "SYSCAT.XSROBJECTCOMPONENTS " on page 2110 "SYSCAT.XSROBJECTDEP " on page 2111 "SYSCAT.XSROBJECTDETAILS " on page 2112 "SYSCAT.XSROBJECTHIERARCHIES " on page 2112 "SYSCAT.XSROBJECTS " on page 2113

[1] The following catalog views for functions, methods, and procedures defined in Db2 Version 7.1 and earlier are still available:

```
Functions:  SYSCAT.FUNCTIONS, SYSCAT.FUNCDEP, SYSCAT.FUNCPARMS
Methods:    SYSCAT.FUNCTIONS, SYSCAT.FUNCDEP, SYSCAT.FUNCPARMS
Procedures: SYSCAT.PROCEDURES, SYSCAT.PROCPARMS
```

However, these views have not been updated since Db2 Version 7.1. Use the SYSCAT.ROUTINES, SYSCAT.ROUTINEDEP, or SYSCAT.ROUTINEPARMS catalog view instead.

Table 157. Road map to the updatable catalog views

Description	Catalog View
columns	“SYSSTAT.COLUMNS ” on page 2116
detailed column group statistics	“SYSSTAT.COLGROUPDIST ” on page 2115 “SYSSTAT.COLGROUPDISTCOUNTS ” on page 2115 “SYSSTAT.COLGROUPS ” on page 2116
detailed column statistics	“SYSSTAT.COLDIST ” on page 2114
indexes	“SYSSTAT.INDEXES ” on page 2118
routines[1]	“SYSSTAT.ROUTINES ” on page 2122
tables	“SYSSTAT.TABLES ” on page 2123

[1] The SYSSTAT.FUNCTIONS catalog view still exists for updating the statistics for functions and methods. This view, however, does not reflect any changes since Db2 Version 7.1.

SYSCAT.ATTRIBUTES

Each row represents an attribute that is defined for a user-defined structured data type. Includes inherited attributes of subtypes.

Table 158. SYSCAT.ATTRIBUTES Catalog View

Column Name	Data Type	Nullable	Description
TYPESCHEMA	VARCHAR (128)		Schema name of the structured data type that includes the attribute.
TYPEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the structured data type belongs. The null value if not a module structured data type.
TYPENAME	VARCHAR (128)		Unqualified name of the structured data type that includes the attribute.
ATTR_NAME	VARCHAR (128)		Attribute name.
ATTR_TYPESCHEMA	VARCHAR (128)		Schema name of the data type of an attribute.
ATTR_TYPEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the data type of an attribute belongs. The null value if not a module attribute.
ATTR_TYPENAME	VARCHAR (128)		Unqualified name of the data type of an attribute.
TARGET_TYPESCHEMA	VARCHAR (128)	Y	Schema name of the target row type. Applies to reference types only; null value otherwise.
TARGET_TYPEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the target row type belongs. The null value if not a module row type. Applies to reference types only; null value otherwise.
TARGET_TYPENAME	VARCHAR (128)	Y	Unqualified name of the target row type. Applies to reference types only; null value otherwise.

Table 158. SYSCAT.ATTRIBUTES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SOURCE_TYPESCHEMA	VARCHAR (128)		For inherited attributes, the schema name of the data type with which the attribute was first defined. For non-inherited attributes, this column is the same as TYPESCHEMA.
SOURCE_TYPEMODULENAME	VARCHAR (128)	Y	For inherited attributes, the unqualified name of the module to which the data type with which the attribute was first defined belongs. For non-inherited attributes, this column is the same as TYPEMODULEID. The null value if not a module data type.
SOURCE_TYPENAME	VARCHAR (128)		For inherited attributes, the unqualified name of the data type with which the attribute was first defined. For non-inherited attributes, this column is the same as TYPENAME.
ORDINAL	SMALLINT		Position of the attribute in the definition of the structured data type, starting with 0.
LENGTH	INTEGER		Length of the attribute data type. 0 if the attribute is a user-defined type.
SCALE	SMALLINT		Scale if the attribute data type is DECIMAL or distinct type based on DECIMAL; the number of digits of fractional seconds if the attribute data type is TIMESTAMP or distinct type based on TIMESTAMP; 0 otherwise.
TYPESTRINGUNITS	VARCHAR (11)	Y	In a Unicode database, the string units that apply to a character string or graphic string data type. Otherwise, the null value.
STRINGUNITSLENGTH	INTEGER	Y	In a Unicode database, the declared number of string units for a character string or graphic string data type. Otherwise, the null value.
CODEPAGE	SMALLINT		For string types, denotes the code page; 0 indicates FOR BIT DATA; 0 for non-string types.
COLLATIONSCHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the attribute; the null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the attribute; the null value otherwise.
LOGGED	CHAR (1)		Applies to LOB types only; blank otherwise. • N = Changes are not logged • Y = Changes are logged

Table 158. SYSCAT.ATTRIBUTES Catalog View (continued)

Column Name	Data Type	Nullable	Description
COMPACT	CHAR (1)		Applies to LOB types only; blank otherwise. • N = Stored in non-compact format • Y = Stored in compact format
DL_FEATURES	CHAR (10)		This column is no longer used and will be removed in a future release.
JAVA_FIELDNAME	VARCHAR (256)	Y	Reserved for future use.
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.

SYSCAT.AUDITPOLICIES

Each row represents an audit policy.

Table 159. SYSCAT.AUDITPOLICIES Catalog View

Column Name	Data Type	Nullable	Description
AUDITPOLICYNAME	VARCHAR (128)		Name of the audit policy.
AUDITPOLICYID	INTEGER		Identifier for the audit policy.
CREATE_TIME	TIMESTAMP		Time at which the audit policy was created.
ALTER_TIME	TIMESTAMP		Time at which the audit policy was last altered.
AUDITSTATUS	CHAR (1)		Status for the AUDIT category. • B = Both • F = Failure • N = None • S = Success
CONTEXTSTATUS	CHAR (1)		Status for the CONTEXT category. • B = Both • F = Failure • N = None • S = Success
VALIDATESTATUS	CHAR (1)		Status for the VALIDATE category. • B = Both • F = Failure • N = None • S = Success

Table 159. SYSCAT.AUDITPOLICIES Catalog View (continued)

Column Name	Data Type	Nullable	Description
CHECKINGSTATUS	CHAR (1)		Status for the CHECKING category. • B = Both • F = Failure • N = None • S = Success
SECMAINTSTATUS	CHAR (1)		Status for the SECMAINT category. • B = Both • F = Failure • N = None • S = Success
OBJMAINTSTATUS	CHAR (1)		Status for the OBJMAINT category. • B = Both • F = Failure • N = None • S = Success
SYSADMINSTATUS	CHAR (1)		Status for the SYSADMIN category. • B = Both • F = Failure • N = None • S = Success
EXECUTESTATUS	CHAR (1)		Status for the EXECUTE category. • B = Both • F = Failure • N = None • S = Success
EXECUTEWITHDATA	CHAR (1)		Host variables and parameter markers logged with EXECUTE category. • N = No • Y = Yes
ERRORTYPE	CHAR (1)		The audit error type. • A = Audit • N = Normal
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.AUDITUSE

Each row represents an audit policy that is associated with an object.

Table 160. SYSCAT.AUDITUSE Catalog View

Column Name	Data Type	Nullable	Description
AUDITPOLICYNAME	VARCHAR (128)		Name of the audit policy.
AUDITPOLICYID	INTEGER		Identifier for the audit policy.
OBJECTTYPE	CHAR (1)		The type of object with which this audit policy is associated. • S = MQT • T = Table • g = Authority • i = Authorization ID • x = Trusted context • Blank = Database
SUBOBJECTTYPE	CHAR (1)		If OBJECTTYPE is 'i', this is the type that the authorization ID represents. • G = Group • R = Role • U = User • Blank = Not applicable
OBJECTSCHEMA	VARCHAR (128)		Schema name of the object for which the audit policy is in use. OBJECTSCHEMA is null if OBJECTTYPE identifies an object to which a schema does not apply.
OBJECTNAME	VARCHAR (128)		Unqualified name of the object for which this audit policy is in use.
AUDITEXCEPTIONENABLED	CHAR (1)		Reserved for future use.

SYSCAT.BUFFERPOOLDBPARTITIONS

Each row represents a combination of a buffer pool and a member, in which the size of the buffer pool on that member is different from its default size for other members in the same database partition group (as represented in SYSCAT.BUFFERPOOLS).

Table 161. SYSCAT.BUFFERPOOLDBPARTITIONS Catalog View

Column Name	Data Type	Nullable	Description
BUFFERPOOLID	INTEGER		Internal buffer pool identifier.
DBPARTITIONNUM	SMALLINT		Member number.
NPAGES	INTEGER		Number of pages in this buffer pool on this member.

SYSCAT.BUFFERPOOLEXCEPTIONS

Each row represents a combination of a buffer pool and a member, in which the size of the buffer pool on that member is different from its default size for other members in the same database partition group (as represented in SYSBUFFERPOOLS).

Table 162. SYSCAT.BUFFERPOOLEXCEPTIONS Catalog View

Column Name	Data Type	Nullable	Description
BUFFERPOOLID	INTEGER		Internal buffer pool identifier.
MEMBER	SMALLINT		Member number.
NPAGES	INTEGER		The number of pages in this buffer pool on this member.

SYSCAT.BUFFERPOOLS

Each row represents the configuration of a buffer pool on one database partition group of a database, or on all database partitions of a database.

Table 163. SYSCAT.BUFFERPOOLS Catalog View

Column Name	Data Type	Nullable	Description
BPNAME	VARCHAR (128)		Name of the buffer pool.
BUFFERPOOLID	INTEGER		Identifier for the buffer pool.
DBPGNAME	VARCHAR (128)	Y	Name of the database partition group (the null value if the buffer pool exists on all database partitions in the database).
NPAGES	INTEGER		Default number of pages in this buffer pool on database partitions in this database partition group. -1(Computed) and -2(Automatic).
PAGESIZE	INTEGER		Page size for this buffer pool on database partitions in this database partition group.
ESTORE	INTEGER		Always 'N'. Extended storage no longer applies.
NUMBLOCKPAGES	INTEGER		Number of pages of the buffer pool that are to be in a block-based area. A block-based area of the buffer pool is only used by prefetchers doing a sequential prefetch.
BLOCKSIZE	INTEGER		Number of pages in a *block*.
NGNAME[1]	VARCHAR (128)	Y	Name of the database partition group (the null value if the buffer pool exists on all database partitions in the database).

Note:

1. The NGNAME column is included for backwards compatibility. See DBPGNAME.

SYSCAT.CASTFUNCTIONS

Each row represents a cast function, not including built-in cast functions.

Table 164. SYSCAT.CASTFUNCTIONS Catalog View

Column Name	Data Type	Nullable	Description
FROM_TYPESCHEMA	VARCHAR (128)		Schema name of the data type of the parameter.
FROM_TYPEMODULENAME	VARCHAR (128)		Unqualified name of the module to which the data type of the parameter belongs. The null value if not a module data type.
FROM_TYPENAME	VARCHAR (128)		Name of the data type of the parameter.
FROM_TYPEMODULEID	INTEGER	Y	Identifier for the module to which the data type of the parameter belongs. The null value if not a module data type.
TO_TYPESCHEMA	VARCHAR (128)		Schema name of the data type of the result after casting.
TO_TYPEMODULENAME	VARCHAR (128)		Unqualified name of the module to which the data type of the result after casting belongs. The null value if not a module data type.
TO_TYPENAME	VARCHAR (128)		Name of the data type of the result after casting.
TO_TYPEMODULEID	INTEGER	Y	Identifier for the module to which the data type of the result after casting belongs. The null value if not a module data type.
FUNCSCHEMA	VARCHAR (128)		Schema name of the function.
FUNCMODULENAME	VARCHAR (128)		Unqualified name of the module to which the function belongs. The null value if not a module function.
FUNCNAME	VARCHAR (128)		Unqualified name of the function.
SPECIFICNAME	VARCHAR (128)		Name of the routine instance (might be system-generated).
FUNCMODULEID	INTEGER	Y	Identifier for the module to which the function belongs. The null value if not a module function.
ASSIGN_FUNCTION	CHAR (1)		• N = Not an assignment function • Y = Implicit assignment function

SYSCAT.CHECKS

Each row represents a check constraint or a derived column in a materialized query table. For table hierarchies, each check constraint is recorded only at the level of the hierarchy where the constraint was created.

Table 165. SYSCAT.CHECKS Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR (128)		Name of the check constraint.
OWNER	VARCHAR (128)		Authorization ID of the owner of the constraint.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
TABSCHEMA	VARCHAR (128)		Schema name of the table to which this constraint applies.
TABNAME	VARCHAR (128)		Name of the table to which this constraint applies.
CREATE_TIME	TIMESTAMP		Time at which the constraint was defined. Used in resolving functions that are part of this constraint. Functions that were created after the constraint was defined are not chosen.
QUALIFIER	VARCHAR (128)		Value of the default schema at the time of object definition. Used to complete any unqualified references.
TYPE	CHAR (1)		Type of check constraint: • C = Check constraint • F = Functional dependency • O = Constraint is an object property • S = System-generated check constraint for a GENERATED ALWAYS column
FUNC_PATH	CLOB (2K)		SQL path in effect when the constraint was defined.
TEXT	CLOB (2M)		Text of the check condition or definition of the derived column.[1]
PERCENTVALID	SMALLINT		Number of rows for which the informational constraint is valid, expressed as a percentage of the total.
COLLATIONSCHEMA	VARCHAR (128)		Schema name of the collation for the constraint.

Table 165. SYSCAT.CHECKS Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLLATIONNAME	VARCHAR (128)		Unqualified name of the collation for the constraint.
COLLATIONSCHEMA_ORDERBY	VARCHAR (128)		Schema name of the collation for ORDER BY clauses in the constraint.
COLLATIONNAME_ORDERBY	VARCHAR (128)		Unqualified name of the collation for ORDER BY clauses in the constraint.
DEFINER[2]	VARCHAR (128)		Authorization ID of the owner of the constraint.
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.

Note:

1. In the catalog view, the text of the check condition is always shown in the database code page and can contain substitution characters. The check constraint will always be applied in the code page of the target table, and will not contain any substitution characters when applied. (The check constraint will be applied based on the original text in the code page of the target table, which might not include the substitution characters.)
2. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.COLAUTH

Each row represents a user, group, or role that has been granted one or more privileges on a column.

Table 166. SYSCAT.COLAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of a privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of a privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
TABSCHEMA	VARCHAR (128)		Schema name of the table or view on which the privilege is held.
TABNAME	VARCHAR (128)		Unqualified name of the table or view on which the privilege is held.
COLNAME	VARCHAR (128)		Name of the column to which this privilege applies.
COLNO	SMALLINT		Column number of this column within the table (starting with 0).

Table 166. SYSCAT.COLAUTH Catalog View (continued)

Column Name	Data Type	Nullable	Description
PRIVTYPE	CHAR (1)		• R = Reference privilege • U = Update privilege
GRANTABLE	CHAR (1)		• G = Privilege is grantable • N = Privilege is not grantable

Note:

1. Privileges can be granted by column, but can be revoked only on a table-wide basis.

SYSCAT.COLCHECKS

Each row represents a column that is referenced by a check constraint or by the definition of a materialized query table. For table hierarchies, each check constraint is recorded only at the level of the hierarchy where the constraint was created.

Table 167. SYSCAT.COLCHECKS Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR (128)		Name of the check constraint.
TABSCHEMA	VARCHAR (128)		Schema name of the table containing the referenced column.
TABNAME	VARCHAR (128)		Unqualified name of the table containing the referenced column.
COLNAME	VARCHAR (128)		Name of the column.
USAGE	CHAR (1)		• D = Column is the child in a functional dependency • P = Column is the parent in a functional dependency • R = Column is referenced in the check constraint • S = Column is a source in the system-generated column check constraint that supports a materialized query table • T = Column is a target in the system-generated column check constraint that supports a materialized query table

SYSCAT.COLDIST

Each row represents the *n*th most frequent value of some column, or the *n*th quantile (cumulative distribution) value of the column. Applies to columns of real tables only (not views). No statistics are recorded for inherited columns of typed tables.

Table 168. SYSCAT.COLDIST Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the table to which the statistics apply.

Table 168. SYSCAT.COLDIST Catalog View (continued)

Column Name	Data Type	Nullable	Description
TABNAME	VARCHAR (128)		Unqualified name of the table to which the statistics apply.
COLNAME	VARCHAR (128)		Name of the column to which the statistics apply.
TYPE	CHAR (1)		• F = Frequency value • Q = Quantile value
SEQNO	SMALLINT		If TYPE = "F", *n* in this column identifies the *n*th most frequent value. If TYPE = "Q", *n* in this column identifies the *n*th quantile value.
COLVALUE[1]	VARCHAR (254)	Y	Data value as a character literal or a null value.
VALCOUNT[2]	BIGINT		If TYPE = "F", VALCOUNT is the number of occurrences of COLVALUE in the column. If TYPE = "Q", VALCOUNT is the number of rows whose value is less than or equal to COLVALUE.
DISTCOUNT[3]	BIGINT	Y	If TYPE = "Q", this column records the number of distinct values that are less than or equal to COLVALUE (the null value if unavailable).

Note:

1. In the catalog view, the value of COLVALUE is always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column's table, and will therefore use actual column values when applied during query optimization.
2. VALCOUNT is estimated, regardless of the sampling method that is used for collecting statistics. VALCOUNT might not be estimated if the given column is a leading column of an index that is defined on the table.
3. DISTCOUNT is collected only for columns that are the first key column in an index.

SYSCAT.COLGROUPCOLS

Each row represents a column that makes up a column group.

Table 169. SYSCAT.COLGROUPCOLS Catalog View

Column Name	Data Type	Nullable	Description
COLGROUPID	INTEGER		Identifier for the column group.
COLNAME	VARCHAR (128)		Name of the column in the column group.
TABSCHEMA	VARCHAR (128)		Schema name of the table for the column in the column group.
TABNAME	VARCHAR (128)		Unqualified name of the table for the column in the column group.
ORDINAL	SMALLINT		Ordinal number of the column in the column group.

SYSCAT.COLGROUPDIST

Each row represents the value of the column in a column group that makes up the *n*th most frequent value of the column group or the *n*th quantile value of the column group.

Table 170. SYSCAT.COLGROUPDIST Catalog View

Column Name	Data Type	Nullable	Description
COLGROUPID	INTEGER		Identifier for the column group.
TYPE	CHAR (1)		• F = Frequency value • Q = Quantile value
ORDINAL	SMALLINT		Ordinal number of the column in the column group.
SEQNO	SMALLINT		If TYPE = 'F', *n* in this column identifies the *n*th most frequent value. If TYPE = 'Q', *n* in this column identifies the *n*th quantile value.
COLVALUE[1]	VARCHAR (254)		Data value as a character literal or a null value.

Note:

1. In the catalog view, the value of COLVALUE is always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column's table, and will therefore use actual column values when applied during query optimization.

SYSCAT.COLGROUPDISTCOUNTS

Each row represents the distribution statistics that apply to the *n*th most frequent value of a column group or the *n*th quantile of a column group.

Table 171. SYSCAT.COLGROUPDISTCOUNTS Catalog View

Column Name	Data Type	Nullable	Description
COLGROUPID	INTEGER		Identifier for the column group.
TYPE	CHAR (1)		• F = Frequency value • Q = Quantile value
SEQNO	SMALLINT		Sequence number *n* representing the *n*th TYPE value.
VALCOUNT	BIGINT		If TYPE = "F", VALCOUNT is the number of occurrences of COLVALUE for the column group with this SEQNO. If TYPE = "Q", VALCOUNT is the number of rows whose value is less than or equal to COLVALUE for the column group with this SEQNO.
DISTCOUNT	BIGINT		If TYPE = "Q", this column records the number of distinct values that are less than or equal to COLVALUE for the column group with this SEQNO (the null value if unavailable).

SYSCAT.COLGROUPS

Each row represents a column group and statistics that apply to the entire column group.

Table 172. SYSCAT.COLGROUPS Catalog View

Column Name	Data Type	Nullable	Description
COLGROUPSCHEMA	VARCHAR (128)		Schema name of the column group.
COLGROUPNAME	VARCHAR (128)		Unqualified name of the column group.
COLGROUPID	INTEGER		Identifier for the column group.
COLGROUPCARD	BIGINT		Cardinality of the column group.
NUMFREQ_VALUES	SMALLINT		Number of frequent values collected for the column group.
NUMQUANTILES	SMALLINT		Number of quantiles collected for the column group.

SYSCAT.COLIDENTATTRIBUTES

Each row represents an identity column that is defined for a table.

Table 173. SYSCAT.COLIDENTATTRIBUTES Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the table or view that contains the column.
TABNAME	VARCHAR (128)		Unqualified name of the table or view that contains the column.
COLNAME	VARCHAR (128)		Name of the column.
START	DECIMAL (31,0)	Y	Start value of the sequence. The null value if the sequence is an alias.
INCREMENT	DECIMAL (31,0)	Y	Increment value. The null value if the sequence is an alias.
MINVALUE	DECIMAL (31,0)	Y	Minimum value of the sequence. The null value if the sequence is an alias.
MAXVALUE	DECIMAL (31,0)	Y	Maximum value of the sequence. The null value if the sequence is an alias.
CYCLE	CHAR (1)		Indicates whether or not the sequence can continue to generate values after reaching its maximum or minimum value. • N = Sequence cannot cycle • Y = Sequence can cycle • Blank = Sequence is an alias.
CACHE	INTEGER		Number of sequence values to pre-allocate in memory for faster access. 0 indicates that values of the sequence are not to be preallocated. In a partitioned database, this value applies to each database partition. -1 if the sequence is an alias.

Table 173. SYSCAT.COLIDENTATTRIBUTES Catalog View (continued)

Column Name	Data Type	Nullable	Description
ORDER	CHAR (1)		Indicates whether or not the sequence numbers must be generated in order of request. • N = Sequence numbers are not required to be generated in order of request • Y = Sequence numbers must be generated in order of request • Blank = Sequence is an alias.
NEXTCACHEFIRSTVALUE	DECIMAL (31,0)	Y	The first value available to be assigned in the next cache block. If no caching, the next value available to be assigned.
SEQID	INTEGER		Identifier for the sequence or alias.

SYSCAT.COLOPTIONS

Each row contains column-specific option values.

Table 174. SYSCAT.COLOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the nickname.
TABNAME	VARCHAR (128)		Nickname for the column for which options are set.
COLNAME	VARCHAR (128)		Local column name.
OPTION	VARCHAR (128)		Name of the column option.
SETTING	CLOB (32K)		Value.

SYSCAT.COLUMNS

Each row represents a column defined for a table, view, or nickname.

Table 175. SYSCAT.COLUMNS Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the table, view, or nickname that contains the column.
TABNAME	VARCHAR (128)		Unqualified name of the table, view, or nickname that contains the column.
COLNAME	VARCHAR (128)		Name of the column.
COLNO	SMALLINT		Number of this column in the table (starting with 0).
TYPESCHEMA	VARCHAR (128)		Schema name of the data type for the column.
TYPENAME	VARCHAR (128)		Unqualified name of the data type for the column.

Table 175. SYSCAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description
LENGTH	INTEGER		Maximum length of the data; 0 for distinct types. The LENGTH column indicates precision for DECIMAL fields, and indicates the number of bytes of storage required for decimal floating-point columns; that is, 8 and 16 for DECFLOAT(16) and DECFLOAT(34), respectively.
SCALE	SMALLINT		Scale if the column type is DECIMAL or number of digits of fractional seconds if the column type is TIMESTAMP; 0 otherwise.
TYPESTRINGUNITS	VARCHAR (11)	Y	In a Unicode database, the string units that apply to a character string or graphic string data type. Otherwise, the null value.
STRINGUNITSLENGTH	INTEGER	Y	In a Unicode database, the declared number of string units for a character string or graphic string data type. Otherwise, the null value.
DEFAULT	CLOB (64K)	Y	Default value for the column of a table expressed as a constant, special register, or cast-function appropriate for the data type of the column. Can also be the keyword NULL. Values might be converted from what was specified as a default value. For example, date and time constants are shown in ISO format, cast-function names are qualified with schema names, and identifiers are delimited. Null value if a DEFAULT clause was not specified or the column is a view column.
NULLS	CHAR (1)		Nullability attribute for the column. • N = Column is not nullable • Y = Column is nullable The value can be "N" for a view column that is derived from an expression or function. Nevertheless, such a column allows null values when the statement using the view is processed with warnings for arithmetic errors.
CODEPAGE	SMALLINT		Code page used for data in this column; 0 if the column is defined as FOR BIT DATA or is not a string type.
COLLATIONSCHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the column; the null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the column; the null value otherwise.

Table 175. SYSCAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description
LOGGED	CHAR (1)		Applies only to columns whose type is LOB or distinct based on LOB; blank otherwise. • N = Column is not logged • Y = Column is logged
COMPACT	CHAR (1)		Applies only to columns whose type is LOB or distinct based on LOB; blank otherwise. • N = Column is not compacted • Y = Column is compacted in storage
COLCARD	BIGINT		Number of distinct values in the column; -1 if statistics are not collected; -2 for inherited columns and columns of hierarchy tables.
HIGH2KEY[1]	VARCHAR (254)	Y	Second-highest data value. Representation of numeric data changed to character literals. Empty if statistics are not collected. Empty for inherited columns and columns of hierarchy tables.
LOW2KEY[1]	VARCHAR (254)	Y	Second-lowest data value. Representation of numeric data changed to character literals. Empty if statistics are not collected. Empty for inherited columns and columns of hierarchy tables.
AVGCOLLEN	INTEGER		Average space in bytes when the column is stored in database memory or a temporary table. For LOB data types that are not inlined, LONG data types, and XML documents, the value used to calculate the average column length is the length of the data descriptor. An extra byte is required if the column is nullable; -1 if statistics have not been collected; -2 for inherited columns and columns of hierarchy tables. Note: The average space required to store the column on disk may be different than the value represented by this statistic.
KEYSEQ	SMALLINT	Y	The column's numerical position within the table's primary key. The null value for columns of subtables and hierarchy tables.
PARTKEYSEQ	SMALLINT	Y	The column's numerical position within the table's distribution key; 0 or the null value if the column is not in the distribution key. The null value for columns of subtables and hierarchy tables.
NQUANTILES	SMALLINT		Number of quantile values recorded in SYSCAT.COLDIST for this column; -1 if statistics are not gathered; -2 for inherited columns and columns of hierarchy tables.

Table 175. SYSCAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description
NMOSTFREQ	SMALLINT		Number of most-frequent values recorded in SYSCAT.COLDIST for this column; -1 if statistics are not gathered; -2 for inherited columns and columns of hierarchy tables.
NUMNULLS	BIGINT		Number of null values in the column; -1 if statistics are not collected.
TARGET_TYPESCHEMA	VARCHAR (128)	Y	Schema name of the target row type, if the type of this column is REFERENCE; null value otherwise.
TARGET_TYPENAME	VARCHAR (128)	Y	Unqualified name of the target row type, if the type of this column is REFERENCE; null value otherwise.
SCOPE_TABSCHEMA	VARCHAR (128)	Y	Schema name of the scope (target table), if the type of this column is REFERENCE; null value otherwise.
SCOPE_TABNAME	VARCHAR (128)	Y	Unqualified name of the scope (target table), if the type of this column is REFERENCE; null value otherwise.
SOURCE_TABSCHEMA	VARCHAR (128)	Y	For columns of typed tables or views, the schema name of the table or view in which the column was first introduced. For non-inherited columns, this is the same as TABSCHEMA. The null value for columns of non-typed tables and views.
SOURCE_TABNAME	VARCHAR (128)	Y	For columns of typed tables or views, the unqualified name of the table or view in which the column was first introduced. For non-inherited columns, this is the same as TABNAME. The null value for columns of non-typed tables and views.
DL_FEATURES	CHAR (10)	Y	This column is no longer used and will be removed in a future release.
SPECIAL_PROPS	CHAR (8)	Y	Applies to REFERENCE type columns only; blanks otherwise. Each byte position is defined as follows: • 1 = Object identifier (OID) column ("Y" for yes; "N" for no) • 2 = User-generated or system-generated ("U" for user; "S" for system) Bytes 3 through 8 are reserved for future use.

Table 175. SYSCAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description
HIDDEN	CHAR (1)		Type of hidden column. • I = Column is defined as IMPLICITLY HIDDEN • S = System-managed hidden column • Blank = Column is not hidden
INLINE_LENGTH	INTEGER		Maximum size in bytes of the internal representation of an instance of an XML document, a structured type, or a LOB data type, that can be stored in the base table; 0 when not applicable.
PCTINLINED	SMALLINT		Percentage of inlined data for columns with VARCHAR, VARGRAPHIC, LOB, or XML data types. -1 if statistics have not been collected or the column data type does not support storing data outside the row. Also -1 for VARCHAR and VARGRAPHIC column if the table is organized by column or the table is organized by row and the row size of the table does not exceed the maximum record length for the page size of the table space.
IDENTITY	CHAR (1)		• N = Not an identity column • Y = Identity column
ROWCHANGETIMESTAMP	CHAR (1)		• N = Not a row change timestamp column • Y = Row change timestamp column
GENERATED	CHAR (1)		Type of generated column. • A = Column value is always generated • D = Column value is generated by default • Blank = Column is not generated
TEXT	CLOB (2M)	Y	For columns defined as generated as expression, this field contains the text of the generated column expression, starting with the keyword AS.
COMPRESS	CHAR (1)		• O = Compress off • S = Compress system default values
AVGDISTINCTPERPAGE	DOUBLE	Y	For future use.
PAGEVARIANCERATIO	DOUBLE		Reserved for future use.
SUB_COUNT	SMALLINT		Average number of sub-elements in the column. Applicable to character string columns only.

Table 175. SYSCAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description
SUB_DELIM_LENGTH	SMALLINT		Average length of the delimiters that separate each sub-element in the column. Applicable to character string columns only.
AVGCOLLENCHAR	INTEGER		Average number of characters (based on the collation in effect for the column) required for the column; -1 if the data type of the column is long, LOB, or XML or if statistics have not been collected; -2 for inherited columns and columns of hierarchy tables.
IMPLICITVALUE[2]	VARCHAR (254)	Y	For a column that was added to a table after the table was created, stores the default value at the time the column was added. For a column that was defined when the table was created, stores the null value.
SECLABELNAME	VARCHAR (128)	Y	Name of the security label that is associated with the column if it is a protected column; the null value otherwise.
ROWBEGIN	CHAR (1)		• N = Not a row begin column • Y = Row begin column
ROWEND	CHAR (1)		• N = Not a row end column • Y = Row end column
TRANSACTIONSTARTID	CHAR (1)		• N = Not a transaction start ID column • Transaction start ID column
RANDDISTKEY	CHAR (1)		• N = Column of a table not using random distribution with random by generation method or not the distribution key • Y = Distribution key of a random distribution table that uses random by generation method
PCTENCODED	SMALLINT		Percentage of values that are encoded as a result of compression for a column in a column-organized table; -1 if the table is not organized by column or if statistics are not collected; -2 for inherited columns and columns of hierarchy tables.
AVGENCODEDCOLLEN	DOUBLE		Average space in bytes when the column is stored in database memory, taking into account that some of the column values might be compressed; -1 if the table is not organized by column or if statistics are not collected; -2 for inherited columns and columns of hierarchy tables.
QUALIFIER	VARCHAR (128)	Y	Reserved for future use.

Table 175. SYSCAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description
FUNC_PATH	CLOB (2K)	Y	Reserved for future use.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. In the catalog view, the values of HIGH2KEY and LOW2KEY are always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column's table, and will therefore use actual column values when applied during query optimization.
2. Attaching a data partition is allowed unless IMPLICITVALUE for a specific column is a non-null value for both the source column and the target column, and the values do not match. In this case, you must drop the source table and then re-create it. A column can have a non-null value in the IMPLICITVALUE field if one of the following conditions is met:
   - The column is created as the result of an ALTER TABLE...ADD COLUMN statement
   - The IMPLICITVALUE field is propagated from a source table during attach
   - The IMPLICITVALUE field is inherited from a source table during detach
   - The IMPLICITVALUE field is set during database upgrade from Version 8 to Version 9, where it is determined to be an added column, or might be an added column. If the database is not certain whether the column is added or not, it is treated as added. An added column is a column that was created as the result of an ALTER TABLE...ADD COLUMN statement.

   To avoid these inconsistencies during non-migration scenarios, it is recommended that you always create the tables that you are going to attach with all the columns already defined. That is, never use the ALTER TABLE statement to add columns to a table before attaching it.

SYSCAT.COLUSE

Each row represents a column that is referenced in the DIMENSIONS clause of a CREATE TABLE statement.

Table 176. SYSCAT.COLUSE Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the table containing the column.
TABNAME	VARCHAR (128)		Unqualified name of the table containing the column.
COLNAME	VARCHAR (128)		Name of the column.
DIMENSION	SMALLINT		Dimension number, based on the reverse order of dimensions specified in the DIMENSIONS clause (initial position is 1). For a composite dimension, this value will be the same for each component of the dimension.
COLSEQ	SMALLINT		Numeric position of the column in the dimension to which it belongs (initial position is 1). The value is 1 for the single column in a noncomposite dimension.

Table 176. SYSCAT.COLUSE Catalog View (continued)

Column Name	Data Type	Nullable	Description
TYPE	CHAR (1)		Type of dimension. • C = Clustering or multidimensional clustering • P = Partitioning

SYSCAT.CONDITIONS

Each row represents a condition defined in a module.

Table 177. SYSCAT.CONDITIONS Catalog View

Column Name	Data Type	Nullable	Description
CONDSCHEMA	VARCHAR (128)		Schema name of the condition.
CONDMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the condition belongs.
CONDNAME	VARCHAR (128)		Unqualified name of the condition.
CONDID	INTEGER		Identifier for the condition.
CONDMODULEID	INTEGER	Y	Identifier of the module to which the condition belongs.
SQLSTATE	CHAR (5)	Y	SQLSTATE value associated with the condition.
OWNER	VARCHAR (128)		Authorization ID of the owner of the condition.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
CREATE_TIME	TIMESTAMP		Time at which the condition was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.CONSTDEP

Each row represents a dependency of a constraint on some other object. The constraint depends on the object of type BTYPE of name BNAME, so a change to the object affects the constraint.

Table 178. SYSCAT.CONSTDEP Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR (128)		Unqualified name of the constraint.
TABSCHEMA	VARCHAR (128)		Schema name of the table to which the constraint applies.
TABNAME	VARCHAR (128)		Unqualified name of the table to which the constraint applies.

Table 178. SYSCAT.CONSTDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
BTYPE	CHAR (1)		Type of object on which the constraint depends. Possible values are: • F = Routine • I = Index • R = User-defined structured type • u = Module alias
BSCHEMA	VARCHAR (128)		Schema name of the object on which the constraint depends.
BMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.
BNAME	VARCHAR (128)		Unqualified name of the object on which the constraint depends.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which the constraint depends.

SYSCAT.CONTEXTATTRIBUTES

Each row represents a trusted context attribute.

Table 179. SYSCAT.CONTEXTATTRIBUTES Catalog View

Column Name	Data Type	Nullable	Description
CONTEXTNAME	VARCHAR (128)		Name of the trusted context.
ATTR_NAME	VARCHAR (128)		Name of the attribute. One of: • ADDRESS • ENCRYPTION
ATTR_VALUE	VARCHAR (128)		Value of the attribute.
ATTR_OPTIONS	VARCHAR (128)	Y	If ATTR_NAME is 'ADDRESS', specifies the level of encryption required for this specific address. A null value indicates that the global ENCRYPTION attribute applies.

SYSCAT.CONTEXTS

Each row represents a trusted context.

Table 180. SYSCAT.CONTEXTS Catalog View

Column Name	Data Type	Nullable	Description
CONTEXTNAME	VARCHAR (128)		Name of the trusted context.
CONTEXTID	INTEGER		Identifier for the trusted context.
SYSTEMAUTHID	VARCHAR (128)		The system authorization ID associated with the trusted context.
DEFAULTCONTEXTROLE	VARCHAR (128)	Y	The default role for the context.

Table 180. SYSCAT.CONTEXTS Catalog View (continued)

Column Name	Data Type	Nullable	Description
CREATE_TIME	TIMESTAMP		Time at which the trusted context was created.
ALTER_TIME	TIMESTAMP		Time at which the trusted context was last altered.
ENABLED	CHAR (1)		Trusted context state. • N = Disabled • Y = Enabled
AUDITPOLICYID	INTEGER	Y	Identifier for the audit policy.
AUDITPOLICYNAME	VARCHAR (128)	Y	Name of the audit policy.
AUDITEXCEPTIONENABLED	CHAR (1)		Reserved for future use.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.CONTROLDEP

Each row represents a dependency of a row permission or column mask on some other object.

Table 181. SYSCAT.CONTROLDEP Catalog View

Column Name	Data Type	Nullable	Description
DSCHEMA	VARCHAR (128)		Schema name of the row permission or column mask.
DNAME	VARCHAR (128)		Unqualified name of the row permission or column mask.
DTYPE	CHAR (1)		Type of the depending object. • y = Row Permission • 2 = Column Mask

Table 181. SYSCAT.CONTROLDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
BTYPE	CHAR (1)		Type of object on which there is a dependency: • A = Table alias • C = Column • F = Routine • H = Hierarchy table • I = Index • L = Detached table • S = Materialized query table • T = Table (not typed) • U = Typed table • V = View (not typed) • W = Typed view • m = Module • s = Statistical Table • u = Module alias • v = Global variable
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BMODULENAME	VARCHAR (128)	Y	Unqualified name for the module of the object on which the control depends. The null value if not a module object.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which the control depends.
BCOLNAME	VARCHAR (128)	Y	If BTYPE = 'C', the column name on which there is a dependency; the null value otherwise.

SYSCAT.CONTROLS

Each row represents a row permission or column mask.

Table 182. SYSCAT.CONTROLS Catalog View

Column Name	Data Type	Nullable	Description
CONTROLSCHEMA	VARCHAR (128)		Schema of the row permission or column mask.
CONTROLNAME	VARCHAR (128)		Name of the row permission or column mask.
OWNER	VARCHAR (128)		Owner of the row permission or column mask.

Table 182. SYSCAT.CONTROLS Catalog View (continued)

Column Name	Data Type	Nullable	Description
OWNERTYPE	CHAR (1)		• S = System • U = User
TABSCHEMA	VARCHAR (128)		Schema of the table on which the row permission or column mask is defined.
TABNAME	VARCHAR (128)		Name of the table on which the row permission or column mask is defined.
COLNAME	VARCHAR (128)		Column name on which the column mask is defined. Blank if this is a row permission.
CONTROLID	INTEGER		Identifier of the control.
CONTROLTYPE	CHAR (1)		• C = Column mask • R = Row permission
ENFORCED	CHAR (1)		• A = All access
IMPLICIT	CHAR (1)		• N = The row permission was explicitly created or this is a column mask • Y = The row permission was implicitly created
ENABLE	CHAR (1)		• N = Not enabled • Y = Enabled
VALID	CHAR (1)		• N = The control is invalid • Y = The control is valid
RULETEXT	CLOB (2M)		The source text of the search condition or case expression portion of the CREATE PERMISSION or CREATE MASK statement.
TABCORRELATION	VARCHAR (128)		The correlation name of the table on which the row permission or column mask is defined. Blank if not specified.
QUALIFIER	VARCHAR (128)		Value of the default schema at the time of object definition. Used to complete any unqualified references.
FUNC_PATH	CLOB (2K)		SQL path in effect when the control was defined.
COLLATIONSCHEMA	VARCHAR (128)		Schema name of the collation for the control.
COLLATIONNAME	VARCHAR (128)		Unqualified name of the collation for the control.

Table 182. SYSCAT.CONTROLS Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLLATIONSCHEMA_ORDERBY	VARCHAR (128)		Schema name of the collation for the ORDER BY clauses in the control.
COLLATIONNAME_ORDERBY	VARCHAR (128)		Unqualified name of the collation for the ORDER BY clauses in the control.
CREATE_TIME	TIMESTAMP		Time at which the row permission or column mask was created.
ALTER_TIME	TIMESTAMP		Time at which the row permission or column mask was last altered.
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.DATAPARTITIONEXPRESSION

Each row represents an expression for that part of the table partitioning key.

Table 183. SYSCAT.DATAPARTITIONEXPRESSION Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the partitioned table.
TABNAME	VARCHAR (128)		Unqualified name of the partitioned table.
DATAPARTITIONKEYSEQ	INTEGER		Expression key part sequence ID, starting from 1.
DATAPARTITIONEXPRESSION	CLOB (32K)		Expression for this entry in the sequence, in SQL syntax.
NULLSFIRST	CHAR (1)		• N = Null values in this expression compare high • Y = Null values in this expression compare low

SYSCAT.DATAPARTITIONS

Each row represents a data partition. Note that the data partition statistics represent one database partition if the table is created on multiple database partitions.

Table 184. SYSCAT.DATAPARTITIONS Catalog View

Column Name	Data Type	Nullable	Description
DATAPARTITIONNAME	VARCHAR (128)		Name of the data partition.
TABSCHEMA	VARCHAR (128)		Schema name of the table to which this data partition belongs.
TABNAME	VARCHAR (128)		Unqualified name of the table to which this data partition belongs.
DATAPARTITIONID	INTEGER		Identifier for the data partition.

Table 184. SYSCAT.DATAPARTITIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
TBSPACEID	INTEGER	Y	Identifier for the table space in which this data partition is stored. The null value when STATUS is 'I'.
PARTITIONOBJECTID	INTEGER	Y	Identifier for the data partition within the table space.
LONG_TBSPACEID	INTEGER	Y	Identifier for the table space in which long data is stored. The null value when STATUS is 'I'.
ACCESS_MODE	CHAR (1)		Access restriction state of the data partition. These states only apply to objects that are in set integrity pending state or to objects that were processed by a SET INTEGRITY statement. Possible values are: • D = No data movement • F = Full access • N = No access • R = Read-only access
STATUS	VARCHAR (32)		• A = Data partition is newly attached • D = Data partition is detached and detached dependents are to be incrementally maintained with respect to the content of this partition • I = Detached data partition whose entry in the catalog is maintained only during asynchronous index cleanup; rows with a STATUS value of 'I' are removed when all index records referring to the detached partition have been deleted • L = Data partition is logically detached • Empty string = Data partition is visible (normal status) Bytes 2 through 32 are reserved for future use.
SEQNO	INTEGER		Data partition sequence number (starting from 0).
LOWINCLUSIVE	CHAR (1)		• N = Low key value is not inclusive • Y = Low key value is inclusive
LOWVALUE	VARCHAR (512)		Low key value (a string representation of an SQL value) for this data partition.
HIGHINCLUSIVE	CHAR (1)		• N = High key value is not inclusive • Y = High key value is inclusive
HIGHVALUE	VARCHAR (512)		High key value (a string representation of an SQL value) for this data partition.

Table 184. SYSCAT.DATAPARTITIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
CARD	BIGINT		Total number of rows in the data partition; -1 if statistics are not collected.
OVERFLOW	BIGINT		Total number of overflow records in the data partition; -1 if statistics are not collected.
NPAGES	BIGINT		Total number of pages on which the rows of the data partition exist; -1 if statistics are not collected.
FPAGES	BIGINT		Total number of pages in the data partition; -1 if statistics are not collected.
ACTIVE_BLOCKS	BIGINT		Total number of active blocks in the data partition, or -1. Applies to multidimensional clustering (MDC) tables only.
INDEX_TBSPACEID	INTEGER		Identifier for the table space which holds all partitioned indexes for this data partition.
AVGROWSIZE	SMALLINT		Average length (in bytes) of both compressed and uncompressed rows in this data partition; -1 if statistics are not collected.
PCTROWSCOMPRESSED	REAL		Compressed rows as a percentage of the total number of rows in the data partition; -1 if statistics are not collected.
PCTPAGESAVED	SMALLINT		Approximate percentage of pages saved in the data partition as a result of row compression. This value includes overhead bytes for each user data row in the data partition, but does not include the space that is consumed by dictionary overhead; -1 if statistics are not collected.
AVGCOMPRESSEDROWSIZE	SMALLINT		Average length (in bytes) of compressed rows in this data partition; -1 if statistics are not collected.
AVGROWCOMPRESSIONRATIO	REAL		For compressed rows in the data partition, this is the average compression ratio by row; that is, the average uncompressed row length divided by the average compressed row length; -1 if statistics are not collected.
STATS_TIME	TIMESTAMP	Y	Time at which any change was last made to recorded statistics for this object. Null if statistics are not collected.

Table 184. SYSCAT.DATAPARTITIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
LASTUSED	DATE		Date when the data partition was last used by any DML statement or the LOAD command. If the table is not partitioned, only the LASTUSED value in SYSCAT.TABLES is updated. This column is not updated when the data partition is used on an HADR standby database. The default value is '0001-01-01'. This value is updated asynchronously not more than once within a 24 hour period and might not reflect usage within the last 15 minutes.
COLDICT_EXISTS	CHAR(1)		Indicates the existence and status of a column compression dictionary for a column-organized table. Possible values are: • A = Common dictionary exists on all members of the table's database partition group • N = Common dictionary does not exist • S = Common dictionary exists on some members of the table's database partition group • X = Not applicable
COLDICT_CREATE_TIME	TIMESTAMP	Y	Time at which the column compression dictionary was created. Null if COLDICT_EXISTS is 'N' or 'X'.
COLDICT_ALTER_TIME	TIMESTAMP	Y	Time at which the column compression dictionary was last altered. At the time of dictionary creation, this value matches the value of COLDICT_CREATE_TIME. Null if COLDICT_EXISTS is 'N' or 'X'.

SYSCAT.DATATYPEDEP

Each row represents a dependency of a user-defined data type on some other object.

Table 185. SYSCAT.DATATYPEDEP Catalog View

Column Name	Data Type	Nullable	Description
TYPESCHEMA	VARCHAR (128)		Schema name of the data type.
TYPEMODULENAME	VARCHAR (128)	Y	Module name of the data type.
TYPENAME	VARCHAR (128)		Unqualified name of the data type.
TYPEMODULEID	INTEGER	Y	Identifier for the module of the data type.

Table 185. SYSCAT.DATATYPEDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: • A = Table alias • F = Routine • G = Global temporary table • H = Hierarchy table • I = Index • N = Nickname • R = User-defined data type • S = Materialized query table • T = Table (not typed) • U = Typed table • V = View (not typed) • W = Typed view • m = Module • q = Sequence alias • u = Module alias • v = Global variable • * = Anchored to the row of a base table
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BMODULENAME	VARCHAR (128)	Y	Module name of the object on which there is a dependency.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which there is a dependency.
TABAUTH	SMALLINT	Y	If BTYPE = 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by the dependent data type; the null value otherwise.

SYSCAT.DATATYPES

Each row represents a built-in or user-defined data type.

Table 186. SYSCAT.DATATYPES Catalog View

Column Name	Data Type	Nullable	Description
TYPESCHEMA	VARCHAR (128)		Schema name of the data type if TYPEMODULEID is null; otherwise schema name of the module to which the data type belongs.

Table 186. SYSCAT.DATATYPES Catalog View (continued)

Column Name	Data Type	Nullable	Description
TYPEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the user-defined type belongs. The null value if not a module user-defined type.
TYPENAME	VARCHAR (128)		Unqualified name of the data type.
OWNER	VARCHAR (128)		Authorization ID of the owner of the type.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
SOURCESCHEMA	VARCHAR (128)	Y	For distinct types or array types, the schema name of the source data type. For user-defined structured types, the schema name of the built-in type of the reference representation type. Null for other data types.
SOURCEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the source data type belongs. The null value if not a module source data type.
SOURCENAME	VARCHAR (128)	Y	For distinct types or array types, the unqualified name of the source data type. For user-defined structured types, the unqualified built-in type name of the reference representation type. Null for other data types.
METATYPE	CHAR (1)		• A = User-defined array type • C = User-defined cursor type • F = User-defined row type • L = User-defined associative array type • R = User-defined structured type • S = System predefined type • T = User-defined distinct type
TYPERULES	CHAR(1)		Indicates the type rules for the user-defined type. • S = Strong typing • W = Weak typing • Blank = Not applicable
TYPEID	SMALLINT		Identifier for the data type.
TYPEMODULEID	INTEGER	Y	Identifier for the module to which the user-defined type belongs. The null value if not a module user-defined type.
SOURCETYPEID	SMALLINT	Y	Identifier for the source type (the null value for built-in types). For user-defined structured types, this is the identifier of the reference representation type.

Table 186. SYSCAT.DATATYPES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SOURCEMODULEID	INTEGER	Y	Identifier for the module to which the source data type belongs. The null value if not a module source data type.
PUBLISHED	CHAR (1)		Indicates whether the module user-defined type can be referenced outside its module. • N = The module user-defined type is not published • Y = The module user-defined type is published • Blank = Not applicable
LENGTH	INTEGER		Maximum length of the type. 0 for built-in parameterized types (for example, DECIMAL and VARCHAR). For user-defined structured types, this is the length of the reference representation type.
SCALE	SMALLINT		Scale for distinct types or reference representation types based on the built-in DECIMAL type; the number of digits of fractional seconds for distinct types based on the built-in TIMESTAMP type; 6 for the built-in TIMESTAMP type; 0 for all other types (including DECIMAL itself).
TYPESTRINGUNITS	VARCHAR (11)	Y	In a Unicode database, the string units that apply to a character string or graphic string data type. Otherwise, the null value.
STRINGUNITSLENGTH	INTEGER	Y	In a Unicode database, the declared number of string units for a character string or graphic string data type. Otherwise, the null value.
CODEPAGE	SMALLINT		Database code page for string types, distinct types based on string types, or reference representation types; 0 otherwise.
COLLATIONSCHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the data type; the null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the data type; the null value otherwise.
ARRAY_LENGTH	INTEGER	Y	Maximum cardinality of the array. The null value if METATYPE is not 'A'.
ARRAYINDEXTYPESCHEMA	VARCHAR (128)	Y	Schema of the data type of the array index. The null value if METATYPE is not 'L'.
ARRAYINDEXTYPENAME	VARCHAR (128)	Y	Name of the data type of the array index. The null value if METATYPE is not 'L'.

Table 186. SYSCAT.DATATYPES Catalog View (continued)

Column Name	Data Type	Nullable	Description
ARRAYINDEXTYPEID	SMALLINT	Y	Identifier for the array index type. The null value if METATYPE is not 'L'.
ARRAYINDEXTYPELENGTH	INTEGER	Y	Maximum length of the array index data type. The null value if METATYPE is not 'L'.
ARRAYINDEXTYPE_ STRINGUNITS	VARCHAR (11)	Y	In a Unicode database, the string units that apply to a character string array index data type. Otherwise, the null value.
ARRAYINDEXTYPE_ STRINGUNITSLENGTH	INTEGER	Y	In a Unicode database, the declared number of string units for a character string array index data type. Otherwise, the null value.
CREATE_TIME	TIMESTAMP		Creation time of the data type.
VALID	CHAR (1)		• N = The data type is invalid • Y = The data type is valid
ATTRCOUNT	SMALLINT		Number of attributes in the data type.
INSTANTIABLE	CHAR (1)		• N = Type cannot be instantiated • Y = Type can be instantiated
WITH_FUNC_ACCESS	CHAR (1)		• N = Methods for this type cannot be invoked using function notation. • Y = All the methods for this type can be invoked using function notation.
FINAL	CHAR (1)		• N = The user-defined type can have subtypes. • Y = The user-defined type cannot have subtypes.
INLINE_LENGTH	INTEGER		Maximum length of a structured type that can be kept with a base table row; 0 otherwise.
NATURAL_ INLINE_LENGTH	INTEGER	Y	System-generated natural inline length of a structured type instance. The null value if this type is not a structured type.
JARSCHEMA	VARCHAR (128)	Y	Schema name of the JAR_ID that identifies the Jar file containing the Java class that implements the SQL type. The null value if the EXTERNAL NAME clause is not specified.
JAR_ID	VARCHAR (128)	Y	Identifier for the Jar file that contains the Java class that implements the SQL type. The null value if the EXTERNAL NAME clause is not specified.

Table 186. SYSCAT.DATATYPES Catalog View (continued)

Column Name	Data Type	Nullable	Description
CLASS	VARCHAR (384)	Y	Java class that implements the SQL type. The null value if the EXTERNAL NAME clause is not specified.
SQLJ_REPRESENTATION	CHAR (1)	Y	SQLJ "representation_spec" of the Java class that implements the SQL type. The null value if the EXTERNAL NAME ... LANGUAGE JAVA REPRESENTATION SPEC clause is not specified. • D = SQL data • S = Serializable
ALTER_TIME	TIMESTAMP		Time at which the data type was last altered.
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the type.
NULLS	CHAR (1)		Nullability attribute for the type. • N = Type is not nullable • Y = Type is nullable
FUNC_PATH	CLOB (2K)	Y	SQL path in effect when the type was created. The null value if no data type check constraint exists.
CONSTRAINT_TEXT	CLOB (64K)	Y	Predicate text which constrains the values allowed for the data type. The null value if no data type check constraint exists.
LAST_REGEN_TIME	TIMESTAMP		Time at which the data type check constraint was last regenerated. The same value as CREATE_TIME if no data type check constraint exists.
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.DBAUTH

Each row represents a user, group, or role that has been granted one or more database-level authorities.

Table 187. SYSCAT.DBAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the authority.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of the authority.

Table 187. SYSCAT.DBAUTH Catalog View (continued)

Column Name	Data Type	Nullable	Description
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
BINDADDAUTH	CHAR (1)		Authority to create packages. • N = Not held • Y = Held
CONNECTAUTH	CHAR (1)		Authority to connect to the database. • N = Not held • Y = Held
CREATETABAUTH	CHAR (1)		Authority to create tables. • N = Not held • Y = Held
DBADMAUTH	CHAR (1)		DBADM authority. • N = Not held • Y = Held
EXTERNALROUTINEAUTH	CHAR (1)		Authority to create external routines. • N = Not held • Y = Held
IMPLSCHEMAAUTH	CHAR (1)		Authority to implicitly create schemas by creating objects in non-existent schemas. • N = Not held • Y = Held
LOADAUTH	CHAR (1)		Authority to use the Db2 load utility. • N = Not held • Y = Held
NOFENCEAUTH	CHAR (1)		Authority to create non-fenced user-defined functions. • N = Not held • Y = Held
QUIESCECONNECTAUTH	CHAR (1)		Authority to access the database when it is quiesced. • N = Not held • Y = Held
LIBRARYADMAUTH	CHAR (1)		Reserved for future use.

Table 187. SYSCAT.DBAUTH Catalog View (continued)

Column Name	Data Type	Nullable	Description
SECURITYADMAUTH	CHAR (1)		Authority to administer database security. • N = Not held • Y = Held
SQLADMAUTH	CHAR (1)		Authority to monitor and tune SQL statements. • N = Not held • Y = Held
WLMADMAUTH	CHAR (1)		Authority to manage WLM objects. • N = Not held • Y = Held
EXPLAINAUTH	CHAR (1)		Authority to explain SQL statements without requiring actual privileges on the objects in the statement. • N = Not held • Y = Held
DATAACCESSAUTH	CHAR (1)		Authority to access data. • N = Not held • Y = Held
ACCESSCTRLAUTH	CHAR (1)		Authority to grant and revoke database object privileges. • N = Not held • Y = Held
CREATESECUREAUTH	CHAR (1)		Authority to create secure objects. • N = Not held • Y = Held

SYSCAT.DBPARTITIONGROUPDEF

Each row represents a database partition that is contained in a database partition group.

Table 188. SYSCAT.DBPARTITIONGROUPDEF Catalog View

Column Name	Data Type	Nullable	Description
DBPGNAME	VARCHAR (128)		Name of the database partition group that contains the database partition.
DBPARTITIONNUM	SMALLINT		Partition number of a database partition that is contained in the database partition group. A valid partition number is between 0 and 999, inclusive.

Table 188. SYSCAT.DBPARTITIONGROUPDEF Catalog View (continued)

Column Name	Data Type	Nullable	Description
IN_USE	CHAR (1)		Status of the database partition. • A = The newly added database partition is not in the distribution map, but the containers for the table spaces in the database partition group have been created; the database partition is added to the distribution map when a redistribute database partition group operation has completed successfully. • D = The database partition will be dropped when a redistribute database partition group operation has completed successfully. • T = The newly added database partition is not in the distribution map, and it was added using the WITHOUT TABLESPACES clause; containers must be added to the table spaces in the database partition group. • Y = The database partition is in the distribution map.

SYSCAT.DBPARTITIONGROUPS

Each row represents a database partition group.

Table 189. SYSCAT.DBPARTITIONGROUPS Catalog View

Column Name	Data Type	Nullable	Description
DBPGNAME	VARCHAR (128)		Name of the database partition group.
OWNER	VARCHAR (128)		Authorization ID of the owner of the database partition group.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
PMAP_ID	SMALLINT		Identifier for the distribution map in the SYSCAT.PARTITIONMAPS catalog view.
REDISTRIBUTE_PMAP_ID	SMALLINT		Identifier for the distribution map currently being used for redistribution; -1 if redistribution is currently not in progress.
CREATE_TIME	TIMESTAMP		Creation time of the database partition group.
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the database partition group.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.EVENTMONITORS

Each row represents an event monitor.

Table 190. SYSCAT.EVENTMONITORS Catalog View

Column Name	Data Type	Nullable	Description
EVMONNAME	VARCHAR (128)		Name of the event monitor.
OWNER	VARCHAR (128)		Authorization ID of the owner of the event monitor.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
TARGET_TYPE	CHAR (1)		Type of target to which event data is written. • F = File • P = Pipe • T = Table • U = Unformatted event table
TARGET	VARCHAR (762)		Name of the target to which file or pipe event monitor data is written. For files, it can be either an absolute path name or a relative path name (relative to the database path for the database; this can be seen using the LIST ACTIVE DATABASES command). For pipes, it can be an absolute path name.
MAXFILES	INTEGER	Y	Maximum number of event files that this event monitor permits in an event path. The null value if there is no maximum, or if TARGET_TYPE is not 'F' (file).
MAXFILESIZE	INTEGER	Y	Maximum size (in 4K pages) that each event file can attain before the event monitor creates a new file. The null value if there is no maximum, or if TARGET_TYPE is not 'F' (file).
BUFFERSIZE	INTEGER	Y	Size of the buffer (in 4K pages) that is used by event monitors with file targets; null value otherwise.
IO_MODE	CHAR (1)	Y	Mode of file input/output (I/O). • B = Blocked • N = Not blocked • Null value = TARGET_TYPE is not 'F' (file) or 'T' (table)

Table 190. SYSCAT.EVENTMONITORS Catalog View (continued)

Column Name	Data Type	Nullable	Description
WRITE_MODE	CHAR (1)	Y	Indicates how this event monitor handles existing event data when the monitor is activated. • A = Append • R = Replace • Null value = TARGET_TYPE is not 'F' (file)
AUTOSTART	CHAR (1)		Indicates whether this event monitor is to be activated automatically when the database starts. • N = No • Y = Yes
DBPARTITIONNUM[1]	SMALLINT		This column is deprecated and will be removed in a future release. Replaced by MEMBER.
MONSCOPE	CHAR (1)		Monitoring scope. • G = Global • L = Local • T = Each database partition on which the table space exists • Blank = WRITE TO TABLE event monitor
EVMON_ACTIVATES	INTEGER		Number of times the event monitor has been activated.
NODENUM[1]	SMALLINT		This column is deprecated and will be removed in a future release. Replaced by MEMBER.
DEFINER[2]	VARCHAR (128)		Authorization ID of the owner of the event monitor.
VERSIONNUMBER	INTEGER		Version, release, and modification level in which the event monitor was created or last upgraded.
MEMBER	SMALLINT		Number of the member where the event monitor runs and logs events.
REMARKS	VARCHAR (254)	Y	Reserved for future use.

Note:

1. The NODENUM column is included for backwards compatibility. See DBPARTITIONNUM.
2. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.EVENTS

Each row represents an event that is being monitored. An event monitor, in general, monitors multiple events.

Table 191. SYSCAT.EVENTS Catalog View

Column Name	Data Type	Nullable	Description
EVMONNAME	VARCHAR (128)		Name of the event monitor that is monitoring this event.
TYPE	VARCHAR (128)		Type of event being monitored. Possible values are: • ACTIVITIES • CHANGEHISTORY • CONNECTIONS • DATABASE • DEADLOCKS • DETAILDEADLOCKS • DLOCKWHIST • DLOCKWHISTAVAL • LOCKING • PKGCACHEBASE • PKGCACHEDETAILED • STATEMENTS • TABLES • TABLESPACES • THRESHOLDVIOLATIONS • TRANSACTIONS • STATISTICS • UOW
FILTER	CLOB (64K)	Y	Full text of the WHERE clause that applies to this event.

SYSCAT.EVENTTABLES

Each row represents the target table of an event monitor that writes to SQL tables.

Table 192. SYSCAT.EVENTTABLES Catalog View

Column Name	Data Type	Nullable	Description
EVMONNAME	VARCHAR (128)		Name of the event monitor.

Table 192. SYSCAT.EVENTTABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
LOGICAL_GROUP	VARCHAR (128)		Name of the logical data group. Possible values are: • ACTIVITYHISTORY • BUFFERPOOL • CHANGESUMMARY • CONN • CONNHEADER • CONTROL • DATAVAL • DB • DBDBMCFG • DDLSTMTEXEC • DEADLOCK • DLCONN • DLLOCK • EVMONSTART • LOCKING • PKGCACHEBASE • PKGCACHEDETAILED • REGVAR • SCMETRICS • SCSTATS • STMT • STMTHIST • STMTVALS • SUBSECTION • SUPERCLASSMETRICS • SUPERCLASSSTATS • TABLE • TABLESPACE • THRESHOLDVIOLATIONS • TXNCOMPLETION • UOW • UTILLOCATION • UTILPHASE • UTILSTART • UTILSTOP • WCSTATS • WLMETRICS • WLSTATS • XACT

Table 192. SYSCAT.EVENTTABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the target table.
TABNAME	VARCHAR (128)		Unqualified name of the target table.
PCTDEACTIVATE	SMALLINT		A percent value that specifies how full a DMS table space must be before an event monitor automatically deactivates. Set to 100 for SMS table spaces.
TABOPTIONS	VARCHAR (32)		String indicating the logical data group options for the target table. Each character in the string represents an option. Possible values are: • E = EXCLUDES • I = INCLUDES • T = TRUNC • Blank = Not applicable

SYSCAT.EXTERNALTABLEOPTIONS

Each row represents a named external table.

Table 193. SYSCAT.EXTERNALTABLEOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
TABLENAME	VARCHAR(128)	No	Name of the external table.
FILENAME	CLOB(4K)	No	Fully-qualified name of the file that contains the data for this external table.
FIELDDELIMITER	CHAR(1)	Yes	Character that indicates the end of a field.
RECORDDELIMITER	CHAR(4)	Yes	Character string that indicates the end of a record.
DECIMALDELIMITER	CHAR(1)	No	Character to represent the decimal delimiter.
DATEDELIMITER	CHAR(1)	No	Character to separate date components.
TIMEDELIMITER	CHAR(1)	No	Character to separate time components.
DATESTYLE	CHAR(12)	No	Format that determines how a date is represented.
TIMESTYLE	CHAR(6)	No	Time format. Possible values are '24HOUR' and '12HOUR'
BOOLEANSTYLE	CHAR(32)	No	Boolean style. Possible values are '1_0', 'TRUE_FALSE', and 'YES_NO'.
NULLVALUE	CHAR(8)	No	String that is used to indicate a null value. The default is 'NULL'.

Table 193. SYSCAT.EXTERNALTABLEOPTIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
QUOTEDVALUE	CHAR(12)	Yes	Type of quotation marks that are to be stripped away from data values that are enclosed by them. Possible values are 'YES' or 'SINGLE' (for single quotation marks), 'DOUBLE' (for double quotation marks), and 'NO' (if no quotation marks are to be stripped).
REQUIREQUOTES	CHAR(5)	Yes	Whether all data values are enclosed in quotation marks. Possible values are: 'TRUE' or 'FALSE'. If REQUIREQUOTES is set to 'TRUE', QUOTEDVALUE must be set to 'YES', 'SINGLE', or 'DOUBLE'.
RECORDLENGTH	INTEGER	Yes	Length of each record of a fixed-format file.
MAXERRORS	BIGINT	No	Maximum number of errors before an external table operation is rolled back.
MAXROWS	BIGINT	No	Maximum number of rows to load. If this value is exceeded, the load operation fails.
Y2BASE	SMALLINT	No	The hundreds component of a year that is specified as only 2 digits. For example, 19 if the 2-digit year 15 represents 1915; 20 if it represents 2015.
FORMAT	CHAR(8)	No	A character string that indicates the data format. Possible values are 'TEXT', 'INTERNAL', 'FIXED', 'BINARY', 'GENERIC', 'NZ_REPL', 'DB2Z_BRF' and 'DB2Z_RRF'.
ENCODING	CHAR(20)	Yes	Code set of the external data file.
REMOTESOURCE	CHAR(10)	Yes	Remote source type. Possible values are: 'ODBC', 'JDBC', 'LOCAL', 'OLEDB', or 'NZ_REPLSRV'.
SOCKETBUFFERSIZE	BIGINT	No	Chunk size, in bytes, at which data is read from the source file.
SKIPROWS	DECFLOAT	No	When reading a table, the number of rows from the top that are to be skipped.
ISFILLRECORD	CHAR(5)	Yes	Whether all fields must be specified. Possible values are: 'TRUE' or 'FALSE'.
ISESCAPE	CHAR(1)	Yes	A character in the range ASCII 32 to ASCII 127 that is interpreted as an escape character.
ISCRINSTRING	CHAR(5)	No	Whether a carriage return character is to be regarded as part of string. Possible values are: 'TRUE' or 'FALSE'.

Table 193. SYSCAT.EXTERNALTABLEOPTIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
ISTRUNCSTRING	CHAR(5)	Yes	Whether to truncate a string if it exceeds the size of the column of the external table into which it is to be loaded. Possible values are: 'TRUE' (a string that is too large to fit in a column is truncated) or 'FALSE' (a string that is too large terminates and rolls back the load operation).
ISCONTROLCHARACTERS	CHAR(5)	No	Whether ASCII characters 1 to 31 are to be allowed in a string. Possible values are: 'TRUE' or 'FALSE'.
ISIGNOREZERO	CHAR(5)	Yes	Whether all occurrences of ASCII character 0 in a string are to be discarded. Possible values are: 'TRUE' or 'FALSE'.
ISTIMEROUNDNANOS	CHAR(5)	Yes	Whether time values are to be rounded to the nearest microsecond. Possible values are: 'TRUE' or 'FALSE'.
ISCOMPRESS	CHAR(5)	Yes	Whether data that is being read from an external table is compressed, and data being written to an external table is to be compressed. Possible values are: 'TRUE' or 'FALSE'.
ISINCLUDEHEADER	CHAR(5)	No	Whether column names are to be included during an unload operation ('TRUE') or ignored ('FALSE').
ISINCLUDEZEROSECONDS	CHAR(5)	Yes	During an unload operation, whether values of 00 seconds are to be unloaded ('TRUE') or ignored ('FALSE').
LOGFILEPATH	CLOB(4K)	No	Name of the file in which to log external table operations.

SYSCAT.FULLHIERARCHIES

Each row represents the relationship between a subtable and a supertable, a subtype and a supertype, or a subview and a superview. All hierarchical relationships, including immediate ones, are included in this view.

Table 194. SYSCAT.FULLHIERARCHIES Catalog View

Column Name	Data Type	Nullable	Description
METATYPE	CHAR (1)		Relationship type. • R = Between structured types • U = Between typed tables • W = Between typed views
SUB_SCHEMA	VARCHAR (128)		Schema name of the subtype, subtable, or subview.
SUB_NAME	VARCHAR (128)		Unqualified name of the subtype, subtable, or subview.

Table 194. SYSCAT.FULLHIERARCHIES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SUPER_SCHEMA	VARCHAR (128)	Y	Schema name of the supertype, supertable, or superview.
SUPER_NAME	VARCHAR (128)	Y	Unqualified name of the supertype, supertable, or superview.
ROOT_SCHEMA	VARCHAR (128)		Schema name of the table, view, or type that is at the root of the hierarchy.
ROOT_NAME	VARCHAR (128)		Unqualified name of the table, view, or type that is at the root of the hierarchy.

SYSCAT.FUNCMAPOPTIONS

Each row represents a function mapping option value.

Table 195. SYSCAT.FUNCMAPOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
FUNCTION_MAPPING	VARCHAR (128)		Name of the function mapping.
OPTION	VARCHAR (128)		Name of the function mapping option.
SETTING	VARCHAR (2048)		Value of the function mapping option.

SYSCAT.FUNCMAPPARMOPTIONS

Each row represents a function mapping parameter option value.

Table 196. SYSCAT.FUNCMAPPARMOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
FUNCTION_MAPPING	VARCHAR (128)		Name of the function mapping.
ORDINAL	SMALLINT		Position of the parameter.
LOCATION	CHAR (1)		Location of the parameter. • L = Local parameter • R = Remote parameter
OPTION	VARCHAR (128)		Name of the function mapping parameter option.
SETTING	VARCHAR (2048)		Value of the function mapping parameter option.

SYSCAT.FUNCMAPPINGS

Each row represents a function mapping.

Table 197. SYSCAT.FUNCMAPPINGS Catalog View

Column Name	Data Type	Nullable	Description
FUNCTION_MAPPING	VARCHAR (128)		Name of the function mapping (might be system-generated).

Table 197. SYSCAT.FUNCMAPPINGS Catalog View (continued)

Column Name	Data Type	Nullable	Description
FUNCSCHEMA	VARCHAR (128)	Y	Schema name of the function. If the null value, the function is assumed to be a built-in function.
FUNCNAME	VARCHAR (1024)	Y	Unqualified name of the user-defined or built-in function.
FUNCID	INTEGER	Y	Identifier for the function.
SPECIFICNAME	VARCHAR (128)	Y	Name of the routine instance (might be system-generated).
OWNER	VARCHAR (128)		Authorization ID of the owner of the mapping. 'SYSIBM' indicates that this is a built-in function.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
WRAPNAME	VARCHAR (128)	Y	Wrapper to which this mapping applies.
SERVERNAME	VARCHAR (128)	Y	Name of the data source.
SERVERTYPE	VARCHAR (30)	Y	Type of data source to which this mapping applies.
SERVERVERSION	VARCHAR (18)	Y	Version of the server type to which this mapping applies.
CREATE_TIME	TIMESTAMP		Time at which the mapping was created.
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the mapping. 'SYSIBM' indicates that this is a built-in function.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.HIERARCHIES

Each row represents the relationship between a subtable and its immediate supertable, a subtype and its immediate supertype, or a subview and its immediate superview. Only immediate hierarchical relationships are included in this view.

Table 198. SYSCAT.HIERARCHIES Catalog View

Column Name	Data Type	Nullable	Description
METATYPE	CHAR (1)		Relationship type. • R = Between structured types • U = Between typed tables • W = Between typed views
SUB_SCHEMA	VARCHAR (128)		Schema name of the subtype, subtable, or subview.

Table 198. SYSCAT.HIERARCHIES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SUB_NAME	VARCHAR (128)		Unqualified name of the subtype, subtable, or subview.
SUPER_SCHEMA	VARCHAR (128)		Schema name of the supertype, supertable, or superview.
SUPER_NAME	VARCHAR (128)		Unqualified name of the supertype, supertable, or superview.
ROOT_SCHEMA	VARCHAR (128)		Schema name of the table, view, or type that is at the root of the hierarchy.
ROOT_NAME	VARCHAR (128)		Unqualified name of the table, view, or type that is at the root of the hierarchy.

SYSCAT.HISTOGRAMTEMPLATEBINS

Each row represents a histogram template bin.

Table 199. SYSCAT.HISTOGRAMTEMPLATEBINS Catalog View

Column Name	Data Type	Nullable	Description
TEMPLATENAME	VARCHAR (128)	Y	Name of the histogram template.
TEMPLATEID	INTEGER		Identifier for the histogram template.
BINID	INTEGER		Identifier for the histogram template bin.
BINUPPERVALUE	BIGINT		The upper value for a single bin in the histogram template.

SYSCAT.HISTOGRAMTEMPLATES

Each row represents a histogram template.

Table 200. SYSCAT.HISTOGRAMTEMPLATES Catalog View

Column Name	Data Type	Nullable	Description
TEMPLATEID	INTEGER		Identifier for the histogram template.
TEMPLATENAME	VARCHAR (128)		Name of the histogram template.
CREATE_TIME	TIMESTAMP		Time at which the histogram template was created.
ALTER_TIME	TIMESTAMP		Time at which the histogram template was last altered.
NUMBINS	INTEGER		Number of bins in the histogram template, including the last bin that has an unbounded top value.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.HISTOGRAMTEMPLATEUSE

Each row represents a relationship between a workload management object that can use histogram templates and a histogram template.

Table 201. SYSCAT.HISTOGRAMTEMPLATEUSE Catalog View

Column Name	Data Type	Nullable	Description
TEMPLATENAME	VARCHAR (128)	Y	Name of the histogram template.
TEMPLATEID	INTEGER		Identifier for the histogram template.
HISTOGRAMTYPE	CHAR (1)		The type of information collected by histograms based on this template. • C = Activity estimated cost histogram • E = Activity execution time histogram • I = Activity interarrival time histogram • L = Activity life time histogram • Q = Activity queue time histogram • R = Request execution time histogram • U = Unit of work life time histogram
OBJECTTYPE	CHAR (1)		The type of WLM object. • b = Service class • k = Work action • w = Workload
OBJECTID	INTEGER		Identifier of the WLM object.
SERVICECLASSNAME	VARCHAR (128)	Y	Name of the service class.
PARENTSERVICECLASSNAME	VARCHAR (128)	Y	The name of the parent service class of the service subclass that uses the histogram template.
WORKACTIONNAME	VARCHAR (128)	Y	The name of the work action that uses the histogram template.
WORKACTIONSETNAME	VARCHAR (128)	Y	The name of the work action set containing the work action that uses the histogram template.
WORKLOADNAME	VARCHAR (128)	Y	The name of the workload that uses the histogram template.

SYSCAT.INDEXAUTH

Each row represents a user, group, or role that has been granted CONTROL privilege on an index.

Table 202. SYSCAT.INDEXAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user

Table 202. SYSCAT.INDEXAUTH Catalog View (continued)

Column Name	Data Type	Nullable	Description
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
INDSCHEMA	VARCHAR (128)		Schema name of the index.
INDNAME	VARCHAR (128)		Unqualified name of the index.
CONTROLAUTH	CHAR (1)		CONTROL privilege. • N = Not held • Y = Held

SYSCAT.INDEXCOLUSE

Each row represents a column that participates in an index.

Table 203. SYSCAT.INDEXCOLUSE Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR (128)		Schema name of the index.
INDNAME	VARCHAR (128)		Unqualified name of the index.
COLNAME	VARCHAR (128)		Name of the column.
COLSEQ	SMALLINT		Numeric position of the column in the index (initial position is 1).
COLORDER	CHAR (1)		Order of the values in this index column. Possible values are: • A = Ascending • D = Descending • I = INCLUDE column (ordering ignored) • R = Random
COLLATIONSCHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the column; the null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the column; the null value otherwise.

Table 203. SYSCAT.INDEXCOLUSE Catalog View (continued)

Column Name	Data Type	Nullable	Description
VIRTUAL	CHAR (1)	Y	• N = Column exists in the table on which this index is defined. • S = Virtual index column found in statistical view associated with the index. The expression for this part of the index key is stored in the TEXT column. • Y = Virtual index column that does not exist in the table on which this index is defined.
TEXT	CLOB (64K)	Y	The expression text for this part of the index key. The null value if this part of the index key is not based on an expression.

SYSCAT.INDEXDEP

Each row represents a dependency of an index on some other object. The index depends on an object of type BTYPE and name BNAME, so a change to the object affects the index.

Table 204. SYSCAT.INDEXDEP Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR (128)		Schema name of the index.
INDNAME	VARCHAR (128)		Unqualified name of the index.

Table 204. SYSCAT.INDEXDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: • A = Table alias • B = Trigger • C = Column • F = Routine • G = Global temporary table • H = Hierachy table • I = Index • K = Package • L = Detached table • N = Nickname • O = Privilege dependency on all subtables or subviews in a table or view hierarchy • Q = Sequence • R = User-defined data type • S = Materialized query table • T = Table (not typed) • U = Typed table • V = View (not typed) • W = Typed view • X = Index extension • Z = XSR object • m = Module • q = Sequence alias • u = Module alias • v = Global variable • * = Anchored to the row of a base table
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which there is a dependency.
TABAUTH	SMALLINT	Y	If BTYPE = 'O', 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by the dependent index; the null value otherwise.

SYSCAT.INDEXES

Each row represents an index. Indexes on typed tables are represented by two rows: one for the "logical index" on the typed table, and one for the "H-index" on the hierarchy table.

Table 205. SYSCAT.INDEXES Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR (128)		Schema name of the index.
INDNAME	VARCHAR (128)		Unqualified name of the index.
OWNER	VARCHAR (128)		Authorization ID of the owner of the index.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
TABSCHEMA	VARCHAR (128)		Schema name of the table or nickname on which the index is defined.
TABNAME	VARCHAR (128)		Unqualified name of the table or nickname on which the index is defined.
COLNAMES	VARCHAR (640)		This column is no longer used and will be removed in the next release. Use SYSCAT.INDEXCOLUSE for this information.
UNIQUERULE	CHAR (1)		Unique rule. • D = Permits duplicates • U = Unique • P = Implements primary key
MADE_UNIQUE	CHAR (1)		• N = Index remains as it was created. • Y = Index was originally non-unique but was converted to a unique index to support a unique or primary key constraint. If the constraint is dropped, the index reverts to being non-unique.
COLCOUNT	SMALLINT		Number of columns in the key, plus the number of include columns, if any.
UNIQUE_COLCOUNT	SMALLINT		Number of columns required for a unique key. It is always <= COLCOUNT, and < COLCOUNT only if there are include columns; -1 if the index has no unique key (that is, it permits duplicates).

Table 205. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
INDEXTYPE[5]	CHAR (4)		Type of index. • BLOK = Block index • CLUS = Clustering index (controls the physical placement of newly inserted rows) • CPMA = Page map index for a column-organized table • DIM = Dimension block index • MDST = Modification state index • RCT = Key sequence index for a range-clustered table • REG = Regular index • TEXT = Text index • XPTH = XML path index • XRGN = XML region index • XVIL = Index over XML column (logical) • XVIP = Index over XML column (physical)
ENTRYTYPE	CHAR (1)		• H = This row represents an index on a hierarchy table • L = This row represents a logical index on a typed table • Blank = This row represents an index on an untyped table
PCTFREE	SMALLINT		Percentage of each index page to be reserved during the initial building of the index. This space is available for data insertions after the index has been built.
IID	SMALLINT		Identifier for the index.
NLEAF	BIGINT		Number of leaf pages; -1 if statistics are not collected.
NLEVELS	SMALLINT		Number of index levels; -1 if statistics are not collected.
FIRSTKEYCARD	BIGINT		Number of distinct first-key values; -1 if statistics are not collected.
FIRST2KEYCARD	BIGINT		Number of distinct keys using the first two columns of the index; -1 if statistics are not collected, or if not applicable.
FIRST3KEYCARD	BIGINT		Number of distinct keys using the first three columns of the index; -1 if statistics are not collected, or if not applicable.
FIRST4KEYCARD	BIGINT		Number of distinct keys using the first four columns of the index; -1 if statistics are not collected, or if not applicable.

Table 205. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
FULLKEYCARD	BIGINT		Number of distinct full-key values; -1 if statistics are not collected.
CLUSTERRATIO[3]	SMALLINT		Degree of data clustering with the index; -1 if statistics are not collected or if detailed index statistics are collected (in which case, CLUSTERFACTOR will be used instead).
CLUSTERFACTOR[3]	DOUBLE		Finer measurement of the degree of clustering; -1 if statistics are not collected or if the index is defined on a nickname.
SEQUENTIAL_PAGES	BIGINT		Number of leaf pages located on disk in index key order with few or no large gaps between them; -1 if statistics are not collected.
DENSITY	INTEGER		Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percent (integer between 0 and 100); -1 if statistics are not collected.
USER_DEFINED	SMALLINT		1 if this index was defined by a user and has not been dropped; 0 otherwise.
SYSTEM_REQUIRED	SMALLINT		• 1 if one or the other of the following conditions is met: – This index is required for a primary or unique key constraint, or this index is a dimension block index or composite block index for a multidimensional clustering (MDC) table or an insert time clustering (ITC) table. – This is the index on the object identifier (OID) column of a typed table. • 2 if both of the following conditions are met: – This index is required for a primary or unique key constraint, or this index is a dimension block index or composite block index for an MDC table or an ITC table. – This is the index on the OID column of a typed table. • 0 otherwise.
CREATE_TIME	TIMESTAMP		Time when the index was created.
STATS_TIME	TIMESTAMP	Y	Last time that any change was made to the recorded statistics for this index. The null value if no statistics are available.

Table 205. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
PAGE_FETCH_PAIRS[3]	VARCHAR (520)		A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the table with this index using that hypothetical buffer. Zero-length string if no data is available.
MINPCTUSED	SMALLINT		A non-zero integer value indicates that the index is enabled for online defragmentation, and represents the minimum percentage of used space on a page before a page merge can be attempted. A zero value indicates that no page merge is attempted.
REVERSE_SCANS	CHAR (1)		• N = Index does not support reverse scans • Y = Index supports reverse scans
INTERNAL_FORMAT	SMALLINT		Possible values are: • 1 = Index does not have backward pointers • 2 or greater = Index has backward pointers • 6 = Index is a composite block index
COMPRESSION	CHAR (1)		Specifies whether index compression is activated • N = Not activated • Y = Activated
IESCHEMA	VARCHAR (128)	Y	Schema name of the index extension. The null value for ordinary indexes.
IENAME	VARCHAR (128)	Y	Unqualified name of the index extension. The null value for ordinary indexes.
IEARGUMENTS	CLOB (64K)	Y	External information of the parameter specified when the index is created. The null value for ordinary indexes.
INDEX_OBJECTID	INTEGER		Identifier for the index object.
NUMRIDS	BIGINT		Total number of row identifiers (RIDs) or block identifiers (BIDs) in the index; -1 if not known.
NUMRIDS_DELETED	BIGINT		Total number of row identifiers (or block identifiers) in the index that are marked deleted, excluding those identifiers on leaf pages on which all the identifiers are marked deleted.

Table 205. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
NUM_EMPTY_LEAFS	BIGINT		Total number of index leaf pages that have all of their row identifiers (or block identifiers) marked deleted.
AVERAGE_RANDOM_FETCH_PAGES[1,2]	DOUBLE		Average number of random table pages between sequential page accesses when fetching using the index; -1 if not known.
AVERAGE_RANDOM_PAGES[2]	DOUBLE		Average number of random table pages between sequential page accesses; -1 if not known.
AVERAGE_SEQUENCE_GAP[2]	DOUBLE		Gap between index page sequences. Detected through a scan of index leaf pages, each gap represents the average number of index pages that must be randomly fetched between sequences of index pages; -1 if not known.
AVERAGE_SEQUENCE_FETCH_GAP[1,2]	DOUBLE		Gap between table page sequences when fetching using the index. Detected through a scan of index leaf pages, each gap represents the average number of table pages that must be randomly fetched between sequences of table pages; -1 if not known.
AVERAGE_SEQUENCE_PAGES[2]	DOUBLE		Average number of index pages that are accessible in sequence (that is, the number of index pages that the prefetchers would detect as being in sequence); -1 if not known.
AVERAGE_SEQUENCE_FETCH_PAGES[1,2]	DOUBLE		Average number of table pages that are accessible in sequence (that is, the number of table pages that the prefetchers would detect as being in sequence) when fetching using the index; -1 if not known.
TBSPACEID	INTEGER		Identifier for the index table space.
LEVEL2PCTFREE	SMALLINT		Percentage of each index level 2 page to be reserved during initial building of the index. This space is available for future inserts after the index has been built.
PAGESPLIT	CHAR (1)		Index page split behavior. • H = High • L = Low • S = Symmetric • Blank = Not applicable

Table 205. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
AVGPARTITION_ CLUSTERRATIO[3]	SMALLINT		Degree of data clustering within a single data partition. -1 if the table is not partitioned, if statistics are not collected, or if detailed statistics are collected (in which case AVGPARTITION_ CLUSTERFACTOR will be used instead).
AVGPARTITION_ CLUSTERFACTOR[3]	DOUBLE		Finer measurement of the degree of clustering within a single data partition. -1 if the table is not partitioned, if statistics are not collected, or if the index is defined on a nickname.
AVGPARTITION_PAGE_FETCH_ PAIRS[3]	VARCHAR (520)		A list of paired integers in character form. Each pair represents a potential buffer pool size and the corresponding page fetches required to access a single data partition from the table. Zero-length string if no data is available, or if the table is not partitioned.
PCTPAGESSAVED	SMALLINT		Approximate percentage of pages saved in the index as a result of index compression. -1 if statistics are not collected.
DATAPARTITION_ CLUSTERFACTOR	DOUBLE		A statistic measuring the "clustering" of the index keys with regard to data partitions. It is a number between 0 and 1, with 1 representing perfect clustering and 0 representing no clustering.
INDCARD	BIGINT		Cardinality of the index. This might be different from the cardinality of the table for indexes that do not have a one-to-one relationship between the table rows and the index entries.
AVGLEAFKEYSIZE	INTEGER		Average index key size for keys on leaf pages in the index.
AVGNLEAFKEYSIZE	INTEGER		Average index key size for keys on non-leaf pages in the index.
OS_PTR_SIZE	INTEGER		Platform word size with which the index was created. • 32 = 32-bit • 64 = 64-bit
COLLECTSTATISTCS	CHAR (1)		Specifies how statistics were collected at index creation time. • D = Collect detailed index statistics • S = Collect sampled detailed index statistics • Y = Collect basic index statistics • Blank = Do not collect index statistics

Table 205. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
DEFINER[4]	VARCHAR (128)		Authorization ID of the owner of the index.
LASTUSED	DATE		Date when the index was last used by any DML statement to perform a scan, or used to enforce referential integrity constraints. This column is not updated when the index is used on an HADR standby database, nor is it updated when rows are inserted into the table on which the index is defined. The default value is '0001-01-01'. This value is updated asynchronously not more than once within a 24 hour period and might not reflect usage within the last 15 minutes.
PERIODNAME	VARCHAR(128)	Y	Name of the period used to define this index.
PERIODPOLICY	CHAR (1)		If a period name was specified, the index uses this period policy. • N = Not applicable • O = Period overlaps not allowed
MADE_WITHOUTOVERLAPS	CHAR (1)		• N = Index remains as it was created. • Y = Index was converted to enforce WITHOUT OVERLAPS on the application period to support a primary or unique constraint. If the constraint is dropped, the index reverts to the original state.
NULLKEYS	CHAR (1)		• N = Keys that contain all null values are not indexed (not considering columns or expressions from the INCLUDE clause) • Y = Keys that contain all null values are indexed (not considering columns or expressions from the INCLUDE clause)
FUNC_PATH	CLOB (2K)	Y	SQL path in effect when the index was defined with an expression in the key. The null value if the key does not include any expressions.
VIEWSCHEMA	VARCHAR(128)	Y	Schema name of the statistical view associated with the index key, if the key includes at least one expression. The null value if there are no expressions in the key.
VIEWNAME	VARCHAR(128)	Y	Unqualified name of the statistical view associated with the index key, if the key includes at least one expression. The null value if there are no expressions in the key.
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Table 205. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description

Note:

1. When using DMS table spaces, this statistic cannot be computed.
2. Prefetch statistics are not gathered during a LOAD...STATISTICS USE PROFILE, or a CREATE INDEX...COLLECT STATISTICS operation, or when the database configuration parameter *seqdetect* is turned off.
3. AVGPARTITION_CLUSTERRATIO, AVGPARTITION_CLUSTERFACTOR, and AVGPARTITION_PAGE_FETCH_PAIRS measure the degree of clustering within a single data partition (local clustering). CLUSTERRATIO, CLUSTERFACTOR, and PAGE_FETCH_PAIRS measure the degree of clustering in the entire table (global clustering). Global clustering and local clustering values can diverge significantly if the table partitioning key is not a prefix of the index key, or when the table partitioning key and the index key are logically independent of each other.
4. The DEFINER column is included for backwards compatibility. See OWNER.
5. The XPTH, XRGN, and XVIP indexes are not recognized by any application programming interface that returns index metadata.

SYSCAT.INDEXEXPLOITRULES

Each row represents an index exploitation rule.

Table 206. SYSCAT.INDEXEXPLOITRULES Catalog View

Column Name	Data Type	Nullable	Description
FUNCID	INTEGER		Identifier for the function.
SPECID	SMALLINT		Number of the predicate specification.
IESCHEMA	VARCHAR (128)		Schema name of the index extension.
IENAME	VARCHAR (128)		Unqualified name of the index extension.
RULEID	SMALLINT		Identifier for the exploitation rule.
SEARCHMETHODID	SMALLINT		Identifier for the search method in the specific index extension.
SEARCHKEY	VARCHAR (640)		Key used to exploit the index.
SEARCHARGUMENT	VARCHAR (2700)		Search arguments used to exploit the index.
EXACT	CHAR (1)		• N = Index lookup is not exact in terms of predicate evaluation • Y = Index lookup is exact in terms of predicate evaluation

SYSCAT.INDEXEXTENSIONDEP

Each row represents a dependency of an index extension on some other object. The index extension depends on the object of type BTYPE of name BNAME, so a change to the object affects the index extension.

Table 207. SYSCAT.INDEXEXTENSIONDEP Catalog View

Column Name	Data Type	Nullable	Description
IESCHEMA	VARCHAR (128)		Schema name of the index extension.
IENAME	VARCHAR (128)		Unqualified name of the index extension.
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: • A = Table alias • B = Trigger • C = Column • F = Routine • G = Global temporary table • H = Hierachy table • I = Index • K = Package • L = Detached table • N = Nickname • O = Privilege dependency on all subtables or subviews in a table or view hierarchy • Q = Sequence • R = User-defined data type • S = Materialized query table • T = Table (not typed) • U = Typed table • V = View (not typed) • W = Typed view • X = Index extension • Z = XSR object • m = Module • q = Sequence alias • u = Module alias • v = Global variable • * = Anchored to the row of a base table
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.

Table 207. SYSCAT.INDEXEXTENSIONDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which there is a dependency.
TABAUTH	SMALLINT	Y	If BTYPE = 'O', 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by the dependent index extension; the null value otherwise.

SYSCAT.INDEXEXTENSIONMETHODS

Each row represents a search method. An index extension can contain more than one search method.

Table 208. SYSCAT.INDEXEXTENSIONMETHODS Catalog View

Column Name	Data Type	Nullable	Description
METHODNAME	VARCHAR (128)		Name of the search method.
METHODID	SMALLINT		Number of the method in the index extension.
IESCHEMA	VARCHAR (128)		Schema name of the index extension on which this method is defined.
IENAME	VARCHAR (128)		Unqualified name of the index extension on which this method is defined.
RANGEFUNCSCHEMA	VARCHAR (128)		Schema name of the range-through function.
RANGEFUNCNAME	VARCHAR (128)		Unqualified name of the range-through function.
RANGESPECIFICNAME	VARCHAR (128)		Function-specific name of the range-through function.
FILTERFUNCSCHEMA	VARCHAR (128)	Y	Schema name of the filter function.
FILTERFUNCNAME	VARCHAR (128)	Y	Unqualified name of the filter function.
FILTERSPECIFICNAME	VARCHAR (128)	Y	Function-specific name of the filter function.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.INDEXEXTENSIONPARMS

Each row represents an index extension instance parameter or source key column.

Table 209. SYSCAT.INDEXEXTENSIONPARMS Catalog View

Column Name	Data Type	Nullable	Description
IESCHEMA	VARCHAR (128)		Schema name of the index extension.
IENAME	VARCHAR (128)		Unqualified name of the index extension.

Table 209. SYSCAT.INDEXEXTENSIONPARMS Catalog View (continued)

Column Name	Data Type	Nullable	Description
ORDINAL	SMALLINT		Sequence number of the parameter or key column.
PARMNAME	VARCHAR (128)		Name of the parameter or key column.
TYPESCHEMA	VARCHAR (128)		Schema name of the data type of the parameter or key column.
TYPENAME	VARCHAR (128)		Unqualified name of the data type of the parameter or key column.
LENGTH	INTEGER		Data type length of the parameter or key column.
SCALE	SMALLINT		Data type scale of the parameter or key column; 0 if not applicable.
PARMTYPE	CHAR (1)		• K = Source key column • P = Index extension instance parameter
CODEPAGE	SMALLINT		Code page of the index extension instance parameter; 0 if not a string type.
COLLATIONSCHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the parameter; the null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the parameter; the null value otherwise.

SYSCAT.INDEXEXTENSIONS

Each row represents an index extension.

Table 210. SYSCAT.INDEXEXTENSIONS Catalog View

Column Name	Data Type	Nullable	Description
IESCHEMA	VARCHAR (128)		Schema name of the index extension.
IENAME	VARCHAR (128)		Unqualified name of the index extension.
OWNER	VARCHAR (128)		Authorization ID of the owner of the index extension.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
CREATE_TIME	TIMESTAMP		Time at which the index extension was defined.
KEYGENFUNCSCHEMA	VARCHAR (128)		Schema name of the key generation function.
KEYGENFUNCNAME	VARCHAR (128)		Unqualified name of the key generation function.
KEYGENSPECIFICNAME	VARCHAR (128)		Name of the key generation function instance (might be system-generated).

Table 210. SYSCAT.INDEXEXTENSIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
TEXT	CLOB (2M)		Full text of the CREATE INDEX EXTENSION statement.
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the index extension.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.INDEXOPTIONS

Each row represents an index-specific option value.

Table 211. SYSCAT.INDEXOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR (128)		Schema name of the index.
INDNAME	VARCHAR (128)		Unqualified name of the index.
OPTION	VARCHAR (128)		Name of the index option.
SETTING	VARCHAR (2048)		Value of the index option.

SYSCAT.INDEXPARTITIONS

Each row represents a partitioned index piece located on one data partition. Note that the index partition statistics represent one database partition if the table is created on multiple database partitions.

Table 212. SYSCAT.INDEXPARTITIONS Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR (128)		Schema name of the index.
INDNAME	VARCHAR (128)		Unqualified name of the index.
TABSCHEMA	VARCHAR (128)		Schema name of the table or nickname on which the index is defined.
TABNAME	VARCHAR (128)		Unqualified name of the table or nickname on which the index is defined.
IID	SMALLINT		Identifier for the index.
INDPARTITIONTBSPACEID	INTEGER		Identifier for the index partition table space.
INDPARTITIONOBJECTID	INTEGER		Identifier for the index partition object.
DATAPARTITIONID	INTEGER		This corresponds to the DATAPARTITIONID found in the SYSCAT.DATAPARTITIONS view.

Table 212. SYSCAT.INDEXPARTITIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
INDCARD	BIGINT		Cardinality of the index partition. This might be different from the cardinality of the corresponding data partition for partitioned indexes that do not have a one-to-one relationship between the data partition rows and the index entries.
NLEAF	BIGINT		Number of leaf pages in the index partition; -1 if statistics are not collected.
NUM_EMPTY_LEAFS	BIGINT		Total number of index leaf pages in the index partition that have all of their row identifiers (RIDs) or block identifiers (BIDs) marked deleted.
NUMRIDS	BIGINT		Total number of row identifiers (RIDs) or block identifiers (BIDs) in the index partition; -1 if not known.
NUMRIDS_DELETED	BIGINT		Total number of row identifiers (RIDs) or block identifiers (BIDs) in the index partition that are marked deleted, excluding those identifiers on leaf pages on which all the identifiers are marked deleted.
FULLKEYCARD	BIGINT		Number of distinct full-key values in the index partition; -1 if statistics are not collected.
NLEVELS	SMALLINT		Number of index levels in the index partition; -1 if statistics are not collected.
CLUSTERRATIO	SMALLINT		Degree of data clustering with the index partition; -1 in either of the following situations: • Statistics are not collected • Detailed index statistics are collected. In this situation, CLUSTERFACTOR will be used instead.
CLUSTERFACTOR	DOUBLE		Finer measurement of the degree of clustering; -1 if statistics are not collected.
FIRSTKEYCARD	BIGINT		Number of distinct first-key values; -1 if statistics are not collected.
FIRST2KEYCARD	BIGINT		Number of distinct keys using the first two columns of the index key; -1 if statistics are not collected, or if not applicable.
FIRST3KEYCARD	BIGINT		Number of distinct keys using the first three columns of the index key; -1 if statistics are not collected, or if not applicable.
FIRST4KEYCARD	BIGINT		Number of distinct keys using the first four columns of the index key; -1 if statistics are not collected, or if not applicable.

Table 212. SYSCAT.INDEXPARTITIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
AVGLEAFKEYSIZE	INTEGER		Average index key size for keys on leaf pages in the index partition; -1 if statistics are not collected.
AVGNLEAFKEYSIZE	INTEGER		Average index key size for keys on non-leaf pages in the index partition; -1 if statistics are not collected.
PCTFREE	SMALLINT		Percentage of each index page to be reserved during the initial building of the index partition. This space is available for data insertions after the index partition has been built.
PAGE_FETCH_PAIRS	VARCHAR (520)		A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the data partition with this index using that hypothetical buffer. Zero-length string if not data is available.
SEQUENTIAL_PAGES	BIGINT		Number of leaf pages located on disk in index key order with few or no large gaps between them; -1 if statistics are not collected.
DENSITY	INTEGER		Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index partition, expressed as a percent (integer between 0 and 100); -1 if statistics are not collected.
AVERAGE_SEQUENCE_GAP	DOUBLE		Gap between index page sequences within the index partition. Detected through a scan of index leaf pages, each gap represents the average number of index pages that must be randomly fetched between sequences of index pages; -1 if not known.
AVERAGE_SEQUENCE_ FETCH_GAP	DOUBLE		Gap between table page sequences when fetching using the index partition. Detected through a scan of index leaf pages, each gap represents the average number of data partition pages that must be randomly fetched between sequences of data partition pages; -1 if not known.
AVERAGE_SEQUENCE_ PAGES	DOUBLE		Average number of index pages that are accessible in sequence (that is, the number of index pages that the prefetchers would detect as being in sequence); -1 if not known.

Table 212. SYSCAT.INDEXPARTITIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
AVERAGE_SEQUENCE_ FETCH_PAGES	DOUBLE		Average number of data partition pages that are accessible in sequence (that is, the number of data partition pages that the prefetchers would detect as being in sequence) when fetching using the index; -1 if not known.
AVERAGE_RANDOM_PAGES	DOUBLE		Average number of random data partition pages between sequential page accesses; -1 if not known.
AVERAGE_RANDOM_FETCH_ PAGES	DOUBLE		Average number of random data partition pages between sequential page accesses when fetching using the index partition; -1 if not known.
STATS_TIME	TIMESTAMP	Y	Last time that any change was made to the recorded statistics for this index partition. The null value if no statistics are available.
COMPRESSION	CHAR (1)		Specifies whether index compression is activated • N = Not activated • Y = Activated
PCTPAGESSAVED	SMALLINT		Approximate percentage of pages saved in the index as a result of index compression. -1 if statistics are not collected.

SYSCAT.INDEXXMLPATTERNS

Each row represents a pattern clause in an index over an XML column.

Table 213. SYSCAT.INDEXXMLPATTERNS Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR (128)		Schema name of the logical index.
INDNAME	VARCHAR (128)		Unqualified name of the logical index.
PINDNAME	VARCHAR (128)		Unqualified name of the physical index.
PINDID	SMALLINT		Identifier for the physical index.
TYPEMODEL	CHAR (1)		• Q = SQL DATA TYPE (Ignore invalid values) • R = SQL DATA TYPE (Reject invalid values)
DATATYPE	VARCHAR (128)		Name of the data type.
HASHED	CHAR (1)		Indicates whether or not the value is hashed. • N = Not hashed • Y = Hashed

Table 213. SYSCAT.INDEXXMLPATTERNS Catalog View (continued)

Column Name	Data Type	Nullable	Description
LENGTH	SMALLINT		Length if DATATYPE = 'VARCHAR' and HASHED = 'N'; precision if DATATYPE = 'DECIMAL'; 0 otherwise.
SCALE	SMALLINT		Scale if DATATYPE = 'DECIMAL'; 0 otherwise.
PATTERNID	SMALLINT		Identifier for the pattern.
PATTERN	CLOB (2M)	Y	Definition of the pattern.

Note:

1. When indexes over XML columns are created, logical indexes that use XML pattern information are created, resulting in the creation of physical B-tree indexes with key columns generated by the database manager to support the logical indexes. A physical index is created to support the data type that is specified in the xmltype-clause of the CREATE INDEX statement.

SYSCAT.INVALIDOBJECTS

Each row represents an invalid object.

Table 214. SYSCAT.INVALIDOBJECTS Catalog View

Column Name	Data Type	Nullable	Description
OBJECTSCHEMA	VARCHAR (128)		Schema name of the object being created or revalidated.
OBJECTMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object being created or revalidated belongs. The null value if the object does not belong to a module.
OBJECTNAME	VARCHAR (128)		Unqualified name of the object being created or revalidated. For routines (OBJECTTYPE = 'F'), this is the specific name.
ROUTINENAME	VARCHAR (128)	Y	Unqualified name of the routine.
OBJECTTYPE	CHAR (1)		Type of the object being created or revalidated. Possible values are: • B = Trigger • F = Routine • R = User-defined data type • V = View • v = Global variable • y = Row permission • 2 = Column mask • 3 = Usage list
SQLCODE	INTEGER	Y	SQLCODE returned in CREATE with errors or revalidation. The null value if the object has never been revalidated.

Table 214. SYSCAT.INVALIDOBJECTS Catalog View (continued)

Column Name	Data Type	Nullable	Description
SQLSTATE	CHAR (5)	Y	SQLSTATE returned in CREATE with errors or revalidation. The null value if the object has never been revalidated.
ERRORMESSAGE	VARCHAR (4000)	Y	Short text for the message associated with SQLCODE. The null value if the object has never been revalidated.
LINENUMBER	INTEGER	Y	Line number where the error occurred in compiled objects. The null value if the object is not a compiled object.
INVALIDATE_TIME	TIMESTAMP		Time at which the object was last invalidated.
LAST_REGEN_TIME	TIMESTAMP	Y	Time at which the object was last revalidated. The null value if the object has never been revalidated.

SYSCAT.KEYCOLUSE

Each row represents a column that participates in a key defined by a unique, primary key, or foreign key constraint.

Table 215. SYSCAT.KEYCOLUSE Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR (128)		Name of the constraint.
TABSCHEMA	VARCHAR (128)		Schema name of the table containing the column.
TABNAME	VARCHAR (128)		Unqualified name of the table containing the column.
COLNAME	VARCHAR (128)		Name of the column.
COLSEQ	SMALLINT		Numeric position of the column in the key for the constraint (initial position is 1). If a constraint uses an existing index, this value is the numeric position of the column in the index.

SYSCAT.MEMBERSUBSETATTRS

Each row represents a member subset attribute.

Table 216. SYSCAT.MEMBERSUBSETATTRS Catalog View

Column Name	Data Type	Nullable	Description
SUBSETID	INTEGER		Subset identifier.
ATTRID	SMALLINT		Attribute identifier. • 1 = DBALIAS
ATTRVALUE	VARCHAR(1000)	Y	Value of attribute.

SYSCAT.MEMBERSUBSETMEMBERS

Each row represents a member that is associated with a member subset.

Table 217. SYSCAT.MEMBERSUBSETMEMBERS Catalog View

Column Name	Data Type	Nullable	Description
SUBSETID	INTEGER		Subset identifier.
MEMBER	SMALLINT		Member ID as defined in db2nodes.cfg.
FAILOVER_PRIORITY	SMALLINT		Stores the failover priority value of the members.

SYSCAT.MEMBERSUBSETS

Each row represents a member subset.

Table 218. SYSCAT.MEMBERSUBSETS Catalog View

Column Name	Data Type	Nullable	Description
SUBSETNAME	VARCHAR(128)		Name of subset.
SUBSETID	INTEGER		Subset identifier.
CREATE_TIME	TIMESTAMP		Time at which subset was defined.
ALTER_TIME	TIMESTAMP		Time at which subset was last altered.
ENABLED	CHAR(1)		State of the member subset. • N = subset is currently disabled • Y = subset is currently enabled
MEMBERPRIORITYBASIS	CHAR(1)		Basis for member priorities: • E = Equal priorities • L = Member load
INCLUSIVESUBSET	CHAR(1)		Inclusiveness of member subset: • N = The member subset is not inclusive • Y = The member subset is inclusive
ALTERNATESERVER	CHAR(1)		Alternate server's inclusion in the member subsets server list: • N = The alternate server is not included in the member subsets server list • Y = The alternate server is included in the member subsets server list
CATALOGDATABASEALIAS	CHAR(1)		Database alias cataloged for member subset: • N = No database alias cataloged explicitly for this member subset • Y = Database alias cataloged explicitly for this member subset
REMARKS	VARCHAR(254)	Y	User provided comments, or the null value.

SYSCAT.MODULEAUTH

Each row represents a user, group, or role that has been granted a privilege on a module.

Table 219. SYSCAT.MODULEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of a privilege
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of a privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
MODULEID	INTEGER		Identifier for the module to which this privilege applies.
MODULESCHEMA	VARCHAR (128)		Schema name of the module to which this privilege applies.
MODULENAME	VARCHAR (128)		Unqualified name of the module to which this privilege applies.
EXECUTEAUTH	CHAR (1)		Privilege to execute objects in the identified module. • G = Held and grantable • N = Not held • Y = Held

SYSCAT.MODULEOBJECTS

Each row represents a function, procedure, global variable, condition, or user-defined type that belongs to a module.

Table 220. SYSCAT.MODULEOBJECTS Catalog View

Column Name	Data Type	Nullable	Description
OBJECTSCHEMA	VARCHAR (128)	N	Schema name of the module.
OBJECTMODULENAME	VARCHAR (128)	N	Unqualified name of the module to which the object belongs.
OBJECTNAME	VARCHAR (128)	N	Unqualified name of the object.
OBJECTTYPE	VARCHAR (9)	N	• CONDITION = The object is a condition • FUNCTION = The object is a function • PROCEDURE = The object is a procedure • TYPE = The object is a data type • VARIABLE = The object is a variable

Table 220. SYSCAT.MODULEOBJECTS Catalog View (continued)

Column Name	Data Type	Nullable	Description
PUBLISHED	CHAR (1)	N	Indicates whether the object can be referenced outside its module. • N = The object is not published • Y = The object is published
SPECIFICNAME	VARCHAR (128)	N	Routine specific name if OBJECTTYPE is 'FUNCTION', 'METHOD' or 'PROCEDURE'; the null value otherwise.
USERDEFINED	CHAR (1)	N	Indicates whether the object is generated by the system or defined by a user. • N = The object is system generated • Y = The object is defined by a user

SYSCAT.MODULES

Each row represents a module.

Table 221. SYSCAT.MODULES Catalog View

Column Name	Data Type	Nullable	Description
MODULESCHEMA	VARCHAR (128)		Schema name of the module.
MODULENAME	VARCHAR (128)		Unqualified name of the module.
MODULEID	INTEGER		Identifier for the module.
DIALECT	VARCHAR (10)		The source dialect of the SQL module. Possible values are: • DB2 SQL PL • PL/SQL • Blank = Not applicable for an alias
OWNER	VARCHAR (128)		Authorization ID of the owner of the module.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
MODULETYPE	CHAR (1)		Type of module. • A = Alias • M = Module • P = PL/SQL package
BASE_MODULESCHEMA	VARCHAR (128)	Y	If MODULETYPE is 'A', contains the schema name of the module or alias that is referenced by this alias; the null value otherwise.

Table 221. SYSCAT.MODULES Catalog View (continued)

Column Name	Data Type	Nullable	Description
BASE_MODULENAME	VARCHAR (128)	Y	If MODULETYPE is 'A', contains the unqualified name of the module or alias that is referenced by this alias; the null value otherwise.
CREATE_TIME	TIMESTAMP		Time at which the module was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.NAMEMAPPINGS

Each row represents the mapping between a "logical" object (typed table or view and its columns and indexes, including inherited columns) and the corresponding "implementation" object (hierarchy table or hierarchy view and its columns and indexes) that implements the logical object.

Table 222. SYSCAT.NAMEMAPPINGS Catalog View

Column Name	Data Type	Nullable	Description
TYPE	CHAR (1)		• C = Column • I = Index • U = Typed table
LOGICAL_SCHEMA	VARCHAR (128)		Schema name of the logical object.
LOGICAL_NAME	VARCHAR (128)		Unqualified name of the logical object.
LOGICAL_COLNAME	VARCHAR (128)	Y	Name of the logical column if TYPE = 'C'; null value otherwise.
IMPL_SCHEMA	VARCHAR (128)		Schema name of the implementation object that implements the logical object.
IMPL_NAME	VARCHAR (128)		Unqualified name of the implementation object that implements the logical object.
IMPL_COLNAME	VARCHAR (128)	Y	Name of the implementation column if TYPE = 'C'; null value otherwise.

SYSCAT.NICKNAMES

Each row represents a nickname.

Table 223. SYSCAT.NICKNAMES Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the nickname.
TABNAME	VARCHAR (128)		Unqualified name of the nickname.
OWNER	VARCHAR (128)		Authorization ID of the owner of the table, view, alias, or nickname.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user

Table 223. SYSCAT.NICKNAMES Catalog View (continued)

Column Name	Data Type	Nullable	Description
STATUS	CHAR (1)		Status of the object. • C = Set integrity pending • N = Normal • X = Inoperative
CREATE_TIME	TIMESTAMP		Time at which the object was created.
STATS_TIME	TIMESTAMP	Y	Time at which any change was last made to recorded statistics for this object. The null value if statistics are not collected.
COLCOUNT	SMALLINT		Number of columns, including inherited columns (if any).
TABLEID	SMALLINT		Internal logical object identifier.
TBSPACEID	SMALLINT		Internal logical identifier for the primary table space for this object.
CARD	BIGINT		Total number of rows in the table; -1 if statistics are not collected.
NPAGES	BIGINT		Total number of pages on which the rows of the nickname exist; -1 if statistics are not gathered.
FPAGES	BIGINT		Total number of pages; -1 if statistics are not gathered.
OVERFLOW	BIGINT		Total number of overflow records; -1 if statistics are not gathered.
PARENTS	SMALLINT	Y	Number of parent tables for this object; that is, the number of referential constraints in which this object is a dependent.
CHILDREN	SMALLINT	Y	Number of dependent tables for this object; that is, the number of referential constraints in which this object is a parent.
SELFREFS	SMALLINT	Y	Number of self-referencing referential constraints for this object; that is, the number of referential constraints in which this object is both a parent and a dependent.
KEYCOLUMNS	SMALLINT	Y	Number of columns in the primary key.
KEYINDEXID	SMALLINT	Y	Index identifier for the primary key index; 0 or the null value if there is no primary key.
KEYUNIQUE	SMALLINT		Number of unique key constraints (other than the primary key constraint) defined on this object.
CHECKCOUNT	SMALLINT		Number of check constraints defined on this object.

Table 223. SYSCAT.NICKNAMES Catalog View (continued)

Column Name	Data Type	Nullable	Description
DATACAPTURE	CHAR (1)		• L = Nickname participates in data replication, including replication of LONG VARCHAR and LONG VARGRAPHIC columns • N = Nickname does not participate in data replication • Y = Nickname participates in data replication
CONST_CHECKED	CHAR (32)		• Byte 1 represents foreign key constraint. • Byte 2 represents check constraint. • Byte 5 represents materialized query table. • Byte 6 represents generated column. • Byte 7 represents staging table. • Byte 8 represents data partitioning constraint. • Other bytes are reserved for future use. Possible values are: • F = In byte 5, the materialized query table cannot be refreshed incrementally. In byte 7, the content of the staging table is incomplete and cannot be used for incremental refresh of the associated materialized query table. • N = Not checked • U = Checked by user • W = Was in 'U' state when the table was placed in set integrity pending state • Y = Checked by system
PARTITION_MODE	CHAR (1)		Reserved for future use.
STATISTICS_PROFILE	CLOB (10M)	Y	RUNSTATS command used to register a statistical profile for the object.
ACCESS_MODE	CHAR (1)		Access restriction state of the object. These states only apply to objects that are in set integrity pending state or to objects that were processed by a SET INTEGRITY statement. Possible values are: • D = No data movement • F = Full access • N = No access • R = Read-only access

Table 223. SYSCAT.NICKNAMES Catalog View (continued)

Column Name	Data Type	Nullable	Description
CODEPAGE	SMALLINT		Code page of the object. This is the default code page used for all character columns, triggers, check constraints, and expression-generated columns.
REMOTE_TABLE	VARCHAR (128)	Y	Unqualified name of the specific data source object (such as a table or a view) for which the nickname was created.
REMOTE_SCHEMA	VARCHAR (128)	Y	Schema name of the specific data source object (such as a table or a view) for which the nickname was created.
SERVERNAME	VARCHAR (128)	Y	Name of the data source that contains the table or view for which the nickname was created.
REMOTE_TYPE	CHAR (1)	Y	Type of object at the data source. • A = Alias • N = Nickname • S = Materialized query table • T = Table (untyped) • V = View (untyped)
CACHINGALLOWED	VARCHAR (1)		• N = Caching is not allowed • Y = Caching is allowed
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the table, view, alias, or nickname.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.PACKAGEAUTH

Each row represents a user, group, or role that has been granted one or more privileges on a package.

Table 224. SYSCAT.PACKAGEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user

Table 224. SYSCAT.PACKAGEAUTH Catalog View (continued)

Column Name	Data Type	Nullable	Description
PKGSCHEMA	VARCHAR (128)		Schema name of the package.
PKGNAME	VARCHAR (128)		Unqualified name of the package.
CONTROLAUTH	CHAR (1)		CONTROL privilege. • N = Not held • Y = Held
BINDAUTH	CHAR (1)		Privilege to bind the package. • G = Held and grantable • N = Not held • Y = Held
EXECUTEAUTH	CHAR (1)		Privilege to execute the package. • G = Held and grantable • N = Not held • Y = Held

SYSCAT.PACKAGEDEP

Each row represents a dependency of a package on some other object. The package depends on the object of type BTYPE of name BNAME, so a change to the object affects the package.

Table 225. SYSCAT.PACKAGEDEP Catalog View

Column Name	Data Type	Nullable	Description
PKGSCHEMA	VARCHAR (128)		Schema name of the package.
PKGNAME	VARCHAR (128)		Unqualified name of the package.
BINDER	VARCHAR (128)		Binder of the package.
BINDERTYPE	CHAR (1)		• U = Binder is an individual user

Table 225. SYSCAT.PACKAGEDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: • A = Table alias • B = Trigger • D = Server definition • F = Routine • G = Global temporary table • I = Index • M = Function mapping • N = Nickname • O = Privilege dependency on all subtables or subviews in a table or view hierarchy • P = Page size • Q = Sequence object • R = User-defined data type • S = Materialized query table • T = Table (untyped) • U = Typed table • V = View (untyped) • W = Typed view • Z = XSR object • m = Module • n = Database partition group • q = Sequence alias • u = Module alias • v = Global variable • 4 = Application-period temporal table • 5 = System-period temporal table
BSCHEMA	VARCHAR (128)		Schema name of an object on which the package depends.
BMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.
BNAME	VARCHAR (128)		Unqualified name of an object on which the package depends.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which there is a dependency.
TABAUTH	SMALLINT	Y	If BTYPE is 'O', 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges that are required by this package (SELECT, INSERT, UPDATE, or DELETE).

Table 225. SYSCAT.PACKAGEDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
VARAUTH	SMALLINT	Y	If BTYPE is 'v', encodes the privileges that are required by this package (READ or WRITE).
UNIQUE_ID	CHAR (8) FOR BIT DATA		Identifier for a specific package when multiple packages having the same name exist.
PKGVERSION	VARCHAR (64)	Y	Version identifier for the package.

Note:

1. If a function instance with dependencies is dropped, the package is put into an "inoperative" state, and it must be explicitly rebound. If any other object with dependencies is dropped, the package is put into an "invalid" state, and the system will attempt to rebind the package automatically when it is first referenced.

SYSCAT.PACKAGES

Each row represents a package that has been created by binding an application program.

Table 226. SYSCAT.PACKAGES Catalog View

Column Name	Data Type	Nullable	Description
PKGSCHEMA	VARCHAR (128)		Schema name of the package.
PKGNAME	VARCHAR (128)		Unqualified name of the package.
BOUNDBY	VARCHAR (128)		Authorization ID of the binder and owner of the package.
BOUNDBYTYPE	CHAR (1)		• U = The binder and owner is an individual user
OWNER	VARCHAR (128)		Authorization ID of the binder and owner of the package.
OWNERTYPE	CHAR (1)		• U = The binder and owner is an individual user
DEFAULT_SCHEMA	VARCHAR (128)		Default schema name used for unqualified names in static SQL statements.
VALID[1]	CHAR (1)		• N = Needs rebinding • V = Validate at run time • X = Package is inoperative because some function instance on which it depends has been dropped; explicit rebind is needed • Y = Valid
UNIQUE_ID	CHAR (8) FOR BIT DATA		Identifier for a specific package when multiple packages having the same name exist.
TOTAL_SECT	SMALLINT		Number of sections in the package.

Table 226. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
FORMAT	CHAR (1)		Date and time format associated with the package. • 0 = Format associated with the territory code of the client • 1 = USA • 2 = EUR • 3 = ISO • 4 = JIS • 5 = LOCAL
ISOLATION	CHAR (2)	Y	Isolation level. • CS = Cursor Stability • RR = Repeatable Read • RS = Read Stability • UR = Uncommitted Read
CONCURRENTACCESSRESOLUTION	CHAR (1)	Y	The value of the CONCURRENTACCESSRESOLUTION bind option: • U = USE CURRENTLY COMMITTED • W = WAIT FOR OUTCOME • Blank = Not specified
BLOCKING	CHAR (1)	Y	Cursor blocking option. • B = Block all cursors • N = No blocking • U = Block unambiguous cursors
INSERT_BUF	CHAR (1)		Setting of the INSERT bind option (applies to partitioned database systems). • N = Inserts are not buffered • Y = Inserts are buffered at the coordinator member to minimize traffic among members
LANG_LEVEL	CHAR (1)	Y	Setting of the LANGLEVEL bind option. • 0 = SAA1 • 1 = MIA • 2 = SQL92E
FUNC_PATH	CLOB (2K)		SQL path in effect when the package was bound.
QUERYOPT	INTEGER		Optimization class under which this package was bound. Used for rebind operations.

Table 226. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
EXPLAIN_LEVEL	CHAR (1)		Indicates whether Explain was requested using the EXPLAIN or EXPLSNAP bind option. • P = Package selection level • Blank = No Explain requested
EXPLAIN_MODE	CHAR (1)		Value of the EXPLAIN bind option. • A = ALL • N = No • R = REOPT • Y = Yes
EXPLAIN_SNAPSHOT	CHAR (1)		Value of the EXPLSNAP bind option. • A = ALL • N = No • R = REOPT • Y = Yes
SQLWARN	CHAR (1)		Indicates whether or not positive SQLCODEs resulting from dynamic SQL statements are returned to the application. • N = No, they are suppressed • Y = Yes
SQLMATHWARN	CHAR (1)		Value of the *dft_sqlmathwarn* database configuration parameter at bind time. Indicates whether arithmetic and retrieval conversion errors return warnings and null values (indicator -2), allowing query processing to continue whenever possible. • N = No, errors are returned • Y = Yes, warnings are returned
CREATE_TIME	TIMESTAMP		Time at which the package was first bound.
EXPLICIT_BIND_TIME	TIMESTAMP		Time at which this package was last changed by: • BIND • REBIND (explicit)

Table 226. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
LAST_BIND_TIME	TIMESTAMP		Time at which the package was last changed by: • BIND • REBIND (explicit) • REBIND (implicit)
ALTER_TIME	TIMESTAMP		Time at which this package was last changed by: • BIND • REBIND (explicit) • REBIND (implicit) • ALTER PACKAGE
CODEPAGE	SMALLINT		Application code page at bind time; -1 if not known.
COLLATIONSCHEMA	VARCHAR (128)		Schema name of the collation for the package.
COLLATIONNAME	VARCHAR (128)		Unqualified name of the collation for the package.
COLLATIONSCHEMA_ORDERBY	VARCHAR (128)		Schema name of the collation for ORDER BY clauses in the package.
COLLATIONNAME_ORDERBY	VARCHAR (128)		Unqualified name of the collation for ORDER BY clauses in the package.
DEGREE	CHAR (5)		Degree of intrapartition parallelism that was specified when the package was bound. • 1 = No parallelism • 2-32767 = User-specified limit • ANY = Degree determined by the system (no limit specified)
MULTINODE_PLANS	CHAR (1)		• N = Package was not bound in a partitioned database environment • Y = Package was bound in a partitioned database environment

Table 226. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
INTRA_PARALLEL	CHAR (1)		Use of intrapartition parallelism by static SQL statements within the package. • F = One or more static SQL statements in this package can use intrapartition parallelism; this parallelism has been disabled for use on a system that is not configured for intrapartition parallelism • N = No static SQL statement uses intrapartition parallelism • Y = One or more static SQL statements in the package use intrapartition parallelism
VALIDATE	CHAR (1)		Indicates whether validity checking can be deferred until run time. • B = All checking must be performed at bind time • R = Validation of tables, views, and privileges that do not exist at bind time is performed at run time

Table 226. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
DYNAMICRULES	CHAR (1)		• B = BIND; dynamic SQL statements are executed with DYNAMICRULES BIND behavior • D = DEFINERBIND; when the package is run within a routine context, dynamic SQL statements in the package are executed with DEFINE behavior; when the package is not run within a routine context, dynamic SQL statements in the package are executed with BIND behavior • E = DEFINERRUN; when the package is run within a routine context, dynamic SQL statements in the package are executed with DEFINE behavior; when the package is not run within a routine context, dynamic SQL statements in the package are executed with RUN behavior • H = INVOKEBIND; when the package is run within a routine context, dynamic SQL statements in the package are executed with INVOKE behavior; when the package is not run within a routine context, dynamic SQL statements in the package are executed with BIND behavior • I = INVOKERUN; when the package is run within a routine context, dynamic SQL statements in the package are executed with INVOKE behavior; when the package is not run within a routine context, dynamic SQL statements in the package are executed with RUN behavior • R = RUN; dynamic SQL statements are executed with RUN behavior; this is the default
SQLERROR	CHAR (1)		SQLERROR option on the most recent subcommand that bound or rebound the package. • C = CONTINUE; creates a package, even if errors occur while binding SQL statements • N = NOPACKAGE; does not create a package or a bind file if an error occurs

Table 226. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
REFRESHAGE	DECIMAL (20,6)		Timestamp duration indicating the maximum length of time between execution of a REFRESH TABLE statement for a materialized query table (MQT) and when that MQT is used in place of a base table.
FEDERATED	CHAR (1)		• N = FEDERATED bind or prep option is turned off • U = FEDERATED bind or prep option was not specified • Y = FEDERATED bind or prep option is turned on
TRANSFORMGROUP	VARCHAR (1024)	Y	Value of the TRANSFORM GROUP bind option; the null value if a transform group is not specified.
REOPTVAR	CHAR (1)		Indicates whether the access path is determined again at execution time using input variable values. • A = Access path is reoptimized for every OPEN or EXECUTE request • N = Access path is determined at bind time • O = Access path is reoptimized only at the first OPEN or EXECUTE request; it is subsequently cached
OS_PTR_SIZE	INTEGER		Word size for the platform on which the package was created. • 32 = Package is a 32-bit package • 64 = Package is a 64-bit package
PKGVERSION	VARCHAR (64)		Version identifier for the package.

Table 226. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
STATICREADONLY	CHAR (1)		Indicates whether or not static cursors will be treated as READ ONLY. Possible values are: • I = Any static cursor that does not contain the FOR UPDATE clause is considered READ ONLY and INSENSITIVE • N = Static cursors take on the attributes that would normally be generated for the given statement text and the setting of the LANGLEVEL precompile option • Y = Any static cursor that does not contain the FOR UPDATE or the FOR READ ONLY clause is considered READ ONLY
FEDERATED_ASYNCHRONY	INTEGER		Indicates the limit on asynchrony (the number of ATQs in the plan) as a bind option when the package was bound. • 0 = No asynchrony • *n* = User-specified limit (32 767 maximum) • -1 = Degree of asynchrony determined by the system • -2 = Degree of asynchrony not specified For a non-federated system, the value is 0.
ANONBLOCK	CHAR (1)		• N = The package is not associated with an anonymous block • Y = The package is associated with an anonymous block
OPTPROFILESCHEMA	VARCHAR (128)	Y	Value of the optimization profile schema specified as part of the OPTPROFILE bind option.
OPTPROFILENAME	VARCHAR (128)	Y	Value of the optimization profile name specified as part of the OPTPROFILE bind option.
PKGID	BIGINT		Identifier for the package.
DBPARTITIONNUM	SMALLINT		Number of the database partition where the package was bound.
DEFINER[2]	VARCHAR (128)		Authorization ID of the binder and owner of the package.
PKG_CREATE_TIME[3]	TIMESTAMP		Time at which the package was first bound.

Table 226. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
APREUSE	CHAR (1)		• N = The query compiler will not attempt to reuse access plans • Y = The access plans in this package should be reused, meaning that at rebind time the query compiler will attempt to choose plans like the ones currently in the package
EXTENDEDINDICATOR	CHAR (1)		• N = Extended indicator variable values are not recognized • Y = Extended indicator variable values are recognized
LASTUSED	DATE		Date when any statement in the package was last executed. This column is not updated for a package associated with an anonymous block. This column is not updated when a statement in the package is executed on an HADR standby database. The default value is '0001-01-01'. This value is updated asynchronously not more than once within a 24 hour period and might not reflect usage within the last 15 minutes.
BUSTIMESENSITIVE	CHAR (1)		• N = Statements that reference an application period temporal table (ATT) will not be affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register • Y = Statements that reference an ATT will be affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register
SYSTIMESENSITIVE	CHAR (1)		• N = Statements that reference a system period temporal table (STT) will not be affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register • Y = Statements that reference an STT will be affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register

Table 226. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
KEEPDYNAMIC	CHAR (1)		Specifies whether dynamic SQL statements are kept after commit or rollback. • N = Inactive dynamic SQL statements need to be prepared again after commit or rollback • Y = Dynamic SQL statements are kept across transactions
STATICASDYNAMIC	CHAR (1)		• N = All static SQL statements in the package are compiled at bind time, using static SQL semantics • Y = All static SQL statements in the package are compiled at execution time, using dynamic SQL semantics
MEMBER	SMALLINT		Number of the member where the package was bound.
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. If a function instance with dependencies is dropped, the package is put into an "inoperative" state, and it must be explicitly rebound. If any other object with dependencies is dropped, the package is put into an "invalid" state, and the system will attempt to rebind the package automatically when it is first referenced.
2. The DEFINER column is included for backwards compatibility. See OWNER.
3. The PKG_CREATE_TIME column is included for backwards compatibility. See CREATE_TIME.

SYSCAT.PARTITIONMAPS

Each row represents a distribution map that is used to distribute the rows of a table among the database partitions in a database partition group, based on hashing the table's distribution.

Table 227. SYSCAT.PARTITIONMAPS Catalog View

Column Name	Data Type	Nullable	Description
PMAP_ID	SMALLINT		Identifier for the distribution map.
PARTITIONMAP	BLOB (65536)		Distribution map, a vector of 32768 two-byte integers for a multiple partition database partition group. For a single partition database partition group, there is one entry denoting the partition number of the single partition.

SYSCAT.PASSTHRUAUTH

Each row represents a user, group, or role that has been granted pass-through authorization to query a data source.

Table 228. SYSCAT.PASSTHRUAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
SERVERNAME	VARCHAR (128)		Name of the data source to which authorization is being granted.

SYSCAT.PERIODS

Each row represents the definition of a period for use with a temporal table.

Table 229. SYSCAT.PERIODS Catalog View

Column Name	Data Type	Nullable	Description
PERIODNAME	VARCHAR (128)		Name of the period.
TABSCHEMA	VARCHAR (128)		Schema name of the table.
TABNAME	VARCHAR (128)		Unqualified name of the table.
BEGINCOLNAME	VARCHAR (128)		Period begin column name.
ENDCOLNAME	VARCHAR (128)		Period end column name.
PERIODTYPE	CHAR (1)		Type of period. • A = Application period • S = System period
HISTORYTABSCHEMA	VARCHAR (128)		Schema name of the history table.
HISTORYTABNAME	VARCHAR (128)		Unqualified name of the history table.

SYSCAT.PREDICATESPECS

Each row represents a predicate specification.

Table 230. SYSCAT.PREDICATESPECS Catalog View

Column Name	Data Type	Nullable	Description
FUNCSCHEMA	VARCHAR (128)		Schema name of the function.
FUNCNAME	VARCHAR (128)		Unqualified name of the function.
SPECIFICNAME	VARCHAR (128)		Name of the function instance.
FUNCID	INTEGER		Identifier for the function.

Table 230. SYSCAT.PREDICATESPECS Catalog View (continued)

Column Name	Data Type	Nullable	Description
SPECID	SMALLINT		Number of this predicate specification.
CONTEXTOP	CHAR (8)		Comparison operator, one of the built-in relational operators (=, <, >, >=, and so on).
CONTEXTEXP	CLOB (2M)		Constant, or an SQL expression.
FILTERTEXT	CLOB (32K)	Y	Text of the data filter expression.

SYSCAT.REFERENCES

Each row represents a referential integrity (foreign key) constraint.

Table 231. SYSCAT.REFERENCES Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR (128)		Name of the constraint.
TABSCHEMA	VARCHAR (128)		Schema name of the dependent table.
TABNAME	VARCHAR (128)		Unqualified name of the dependent table.
OWNER	VARCHAR (128)		Authorization ID of the owner of the constraint.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
REFKEYNAME	VARCHAR (128)		Name of the parent key.
REFTABSCHEMA	VARCHAR (128)		Schema name of the parent table.
REFTABNAME	VARCHAR (128)		Unqualified name of the parent table.
COLCOUNT	SMALLINT		Number of columns in the foreign key.
DELETERULE	CHAR (1)		Delete rule. • A = NO ACTION • C = CASCADE • N = SET NULL • R = RESTRICT
UPDATERULE	CHAR (1)		Update rule. • A = NO ACTION • R = RESTRICT
CREATE_TIME	TIMESTAMP		Time at which the constraint was defined.
FK_COLNAMES	VARCHAR (640)		This column is no longer used and will be removed in a future release. Use SYSCAT.KEYCOLUSE for this information.
PK_COLNAMES	VARCHAR (640)		This column is no longer used and will be removed in a future release. Use SYSCAT.KEYCOLUSE for this information.

Table 231. SYSCAT.REFERENCES Catalog View (continued)

Column Name	Data Type	Nullable	Description
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the constraint.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.ROLEAUTH

Each row represents a role granted to a user, group, role, or PUBLIC.

Table 232. SYSCAT.ROLEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Authorization ID that granted the role.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Authorization ID to which the role was granted.
GRANTEETYPE	CHAR (1)		• G = The grantee is a group • R = The grantee is a role • U = The grantee is an individual user
ROLENAME	VARCHAR (128)		Name of the role.
ROLEID	INTEGER		Identifier for the role.
ADMIN	CHAR (1)		Privilege to grant or revoke the role to or from others, or to comment on the role. • N = Not held • Y = Held

SYSCAT.ROLES

Each row represents a role.

Table 233. SYSCAT.ROLES Catalog View

Column Name	Data Type	Nullable	Description
ROLENAME	VARCHAR (128)		Name of the role.
ROLEID	INTEGER		Identifier for the role.
CREATE_TIME	TIMESTAMP		Time when the role was created.
AUDITPOLICYID	INTEGER	Y	Identifier for the audit policy.
AUDITPOLICYNAME	VARCHAR (128)	Y	Name of the audit policy.
AUDITEXCEPTIONENABLED	CHAR (1)		Reserved for future use.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.ROUTINEAUTH

Each row represents a user, group, or role that has been granted EXECUTE privilege on either a particular routine (function, method, or procedure) in the database that is not defined in a module or all routines in a particular schema in the database that are not defined in a module.

Table 234. SYSCAT.ROUTINEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege. "SYSIBM" if the privilege was granted by the system.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
SCHEMA	VARCHAR (128)		Schema name of the routine.
SPECIFICNAME	VARCHAR (128)	Y	Specific name of the routine. If SPECIFICNAME is the null value and ROUTINETYPE is not "M", the privilege applies to all routines of the type specified in ROUTINETYPE in the schema specified in SCHEMA. If SPECIFICNAME is the null value and ROUTINETYPE is "M", the privilege applies to all methods for the subject type specified by TYPENAME in the schema specified by TYPESCHEMA. If SPECIFICNAME is the null value, ROUTINETYPE is "M", and both TYPENAME and TYPESCHEMA are null values, the privilege applies to all methods for all types in the schema.
TYPESCHEMA	VARCHAR (128)	Y	Schema name of the type for the method. The null value if ROUTINETYPE is not "M".
TYPENAME	VARCHAR (128)	Y	Unqualified name of the type for the method. The null value if ROUTINETYPE is not "M". If TYPENAME is the null value and ROUTINETYPE is "M", the privilege applies to all methods for any subject type if they are in the schema specified by SCHEMA.
ROUTINETYPE	CHAR (1)		Type of the routine. • F = Function • M = Method • P = Procedure

Table 234. SYSCAT.ROUTINEAUTH Catalog View (continued)

Column Name	Data Type	Nullable	Description
EXECUTEAUTH	CHAR (1)		Privilege to execute the routine. • G = Held and grantable • N = Not held • Y = Held
GRANT_TIME	TIMESTAMP		Time at which the privilege was granted.

SYSCAT.ROUTINEDEP

Each row represents a dependency of a routine on some other object. The routine depends on the object of type BTYPE of name BNAME, so a change to the object affects the routine.

Table 235. SYSCAT.ROUTINEDEP Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESCHEMA	VARCHAR (128)		Schema name of the routine that has dependencies on another object.
ROUTINEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module.
SPECIFICNAME	VARCHAR (128)		Specific name of the routine that has dependencies on another object.
ROUTINEMODULEID	INTEGER	Y	Identifier for the module of the object that has dependencies on another object.

Table 235. SYSCAT.ROUTINEDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: • A = Table alias • B = Trigger • C = Column • F = Routine • G = Global temporary table • H = Hierachy table • I = Index • K = Package • L = Detached table • N = Nickname • O = Privilege dependency on all subtables or subviews in a table or view hierarchy • Q = Sequence • R = User-defined data type • S = Materialized query table • T = Table (not typed) • U = Typed table • V = View (not typed) • W = Typed view • X = Index extension • Z = XSR object • m = Module • q = Sequence alias • u = Module alias • v = Global variable • * = Anchored to the row of a base table
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which there is a dependency.
TABAUTH	SMALLINT	Y	If BTYPE = 'O', 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by the dependent routine; the null value otherwise.

Table 235. SYSCAT.ROUTINEDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
ROUTINENAME	VARCHAR (128)		This column is no longer used and will be removed in a future release. See SPECIFICNAME.

SYSCAT.ROUTINEOPTIONS

Each row represents a routine-specific option value.

Table 236. SYSCAT.ROUTINEOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESCHEMA	VARCHAR (128)		Schema name of the routine if ROUTINEMODULEID is null; otherwise schema name of the module to which the routine belongs.
ROUTINEMODULENAME	VARCHAR (128)		Unqualified name of the module to which the routine belongs. The null value if not a module routine.
ROUTINENAME	VARCHAR (128)		Unqualified name of the routine.
SPECIFICNAME	VARCHAR (128)		Name of the routine instance (might be system-generated).
OPTION	VARCHAR (128)		Federated routine option name or routine environment variable name.
SETTING	VARCHAR (2048)		Federated routine option value or routine environment variable value.

SYSCAT.ROUTINEPARMOPTIONS

Each row represents a routine parameter-specific option value.

Table 237. SYSCAT.ROUTINEPARMOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESCHEMA	VARCHAR (128)		Schema name of the routine if ROUTINEMODULEID is null; otherwise schema name of the module to which the routine belongs.
ROUTINENAME	VARCHAR (128)		Unqualified name of the routine.
SPECIFICNAME	VARCHAR (128)		Name of the routine instance (might be system-generated).
ORDINAL	SMALLINT		Position of the parameter within the routine signature.
OPTION	VARCHAR (128)		Name of the federated routine option.
SETTING	VARCHAR (2048)		Value of the federated routine option.

SYSCAT.ROUTINEPARMS

Each row represents a parameter, an aggregation state variable, or the result of a routine defined in SYSCAT.ROUTINES.

Table 238. SYSCAT.ROUTINEPARMS Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESCHEMA	VARCHAR (128)	Y	Schema name of the routine if ROUTINEMODULEID is null; otherwise schema name of the module to which the routine belongs.
ROUTINEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the routine belongs. The null value if not a module routine.
ROUTINENAME	VARCHAR (128)	Y	Unqualified name of the routine.
ROUTINEMODULEID	INTEGER	Y	Identifier for the module to which the routine belongs. The null value if not a module routine.
SPECIFICNAME	VARCHAR (128)	Y	Name of the routine instance (might be system-generated).
PARMNAME	VARCHAR (128)	Y	Name of the parameter, result column, or aggregation state variable; the null value if no name exists.
ROWTYPE	CHAR (1)	Y	• B = Both input and output parameter • C = Result after casting • O = Output parameter • P = Input parameter • R = Result before casting • S = Aggregation state variable
ORDINAL	SMALLINT	Y	If ROWTYPE = "B", "O", or "P", numerical position of the parameter within the routine signature, starting with 1; if ROWTYPE = "R" and the routine returns a table, numerical position of a named column in the result table, starting with 1; if ROWTYPE = "S", numerical position of an aggregation state variable within the routine definition, starting with 1; 0 otherwise.
TYPESCHEMA	VARCHAR (128)	Y	Schema name of the data type if TYPEMODULEID is null; otherwise schema name of the module to which the data type belongs.
TYPEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the data type of the parameter or result belongs. The null value if not a module data type.
TYPENAME	VARCHAR (128)	Y	Unqualified name of the data type.

Table 238. SYSCAT.ROUTINEPARMS Catalog View (continued)

Column Name	Data Type	Nullable	Description
LOCATOR	CHAR (1)	Y	• N = Paramater or result is not passed in the form of a locator • Y = Paramater or result is passed in the form of a locator
LENGTH[1]	INTEGER	Y	Length of the data type; 0 for a user-defined data type; -1 if length attribute of data type is specified as ANY.
SCALE[1]	SMALLINT	Y	Scale if the data type is DECIMAL; the number of digits of fractional seconds if the data type is TIMESTAMP; 0 otherwise.
TYPESTRINGUNITS	VARCHAR (11)	Y	In a Unicode database, the string units that apply to a character string or graphic string data type. Otherwise, the null value.
STRINGUNITSLENGTH	INTEGER	Y	In a Unicode database, the declared number of string units for a character string or graphic string data type. Otherwise, the null value.
CODEPAGE	SMALLINT	Y	Code page associated with the data type; 0 denotes either not applicable, or a character data type declared with the FOR BIT DATA attribute.
COLLATIONSCHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the parameter; the null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the parameter; the null value otherwise.
CAST_FUNCSCHEMA	VARCHAR (128)	Y	Schema name of the function used to cast an argument or a result. Applies to sourced and external functions; the null value otherwise.
CAST_FUNCSPECIFIC	VARCHAR (128)	Y	Unqualified name of the function used to cast an argument or a result. Applies to sourced and external functions; the null value otherwise.
TARGET_TYPESCHEMA	VARCHAR (128)	Y	Schema name of the target type if the type of the parameter or result is REFERENCE. Null value if the type of the parameter or result is not REFERENCE.
TARGET_TYPEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the target type belongs if the type of the parameter or result is REFERENCE. The null value if the type of the parameter or result is not REFERENCE or if the target type is not a module data type.

Table 238. SYSCAT.ROUTINEPARMS Catalog View (continued)

Column Name	Data Type	Nullable	Description
TARGET_TYPENAME	VARCHAR (128)	Y	Unqualified name of the module to which the target type belongs if the type of the parameter or result is REFERENCE. The null value if the type of the parameter or result is not REFERENCE or if the target type is not a module data type.
SCOPE_TABSCHEMA	VARCHAR (128)	Y	Schema name of the scope (target table) if the parameter type is REFERENCE; null value otherwise.
SCOPE_TABNAME	VARCHAR (128)	Y	Unqualified name of the scope (target table) if the parameter type is REFERENCE; null value otherwise.
TRANSFORMGRPNAME	VARCHAR (128)	Y	Name of the transform group for a structured type parameter or result.
DEFAULT	CLOB (64K)	Y	Expression used to calculate the default value of the parameter. The null value if DEFAULT clause was not specified for the parameter.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note: LENGTH and SCALE are set to 0 for sourced functions (functions defined with a reference to another function), because they inherit the length and scale of parameters from their source.

SYSCAT.ROUTINES

Each row represents a user-defined routine (scalar function, table function, sourced function, aggregate interface function, method, or procedure).

Table 239. SYSCAT.ROUTINES Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESCHEMA	VARCHAR (128)	Y	Schema name of the routine if ROUTINEMODULEID is null; otherwise schema name of the module to which the routine belongs.
ROUTINEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the routine belongs. The null value if not a module routine.
ROUTINENAME	VARCHAR (128)	Y	Unqualified name of the routine.
ROUTINETYPE	CHAR (1)	Y	Type of routine. • F = Function • M = Method • P = Procedure
OWNER	VARCHAR (128)	Y	Authorization ID of the owner of the routine.

Table 239. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
OWNERTYPE	CHAR (1)	Y	• S = The owner is the system • U = The owner is an individual user
SPECIFICNAME	VARCHAR (128)	Y	Name of the routine instance (might be system-generated).
ROUTINEID	INTEGER	Y	Identifier for the routine.
ROUTINEMODULEID	INTEGER	Y	Identifier for the module to which the routine belongs. The null value if not a module routine.
RETURN_TYPESCHEMA	VARCHAR (128)	Y	Schema name of the return type for a scalar function or method.
RETURN_TYPEMODULE	VARCHAR (128)	Y	The module name of the return type; the null value if the return type does not belong to any module.
RETURN_TYPENAME	VARCHAR (128)	Y	Unqualified name of the return type for a scalar function or method.
ORIGIN	CHAR (1)	Y	• A = User-defined aggregate interface function • B = Built-in • E = User-defined, external • M = Template function • F = Federated procedure • Q = SQL-bodied[1] • R = System-generated SQL-bodied routine • S = System-generated • T = System-generated transform function (not directly invokable) • U = User-defined, based on a source
FUNCTIONTYPE	CHAR (1)	Y	• C = Column or aggregate • R = Row • S = Scalar • T = Table • Blank = Procedure
PARM_COUNT	SMALLINT	Y	Number of routine parameters; -1 if parameters specified as VARARGS.
LANGUAGE	CHAR (8)	Y	Implementation language for the routine body (or for the source function body, if this function is sourced on another function). Possible values are "C", "CLR", "COBOL", "JAVA", "OLE", "OLEDB", "R", or "SQL". Blanks if ORIGIN is not "E", "Q", or "R".

Table 239. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
DIALECT	VARCHAR (10)	Y	The source dialect of the SQL routine body: • Db2 SQL PL • PL/SQL • Blank = Not an SQL routine
SOURCESCHEMA	VARCHAR (128)	Y	If ORIGIN = "U" and the source function is a user-defined function, contains the schema name of the specific name of the source function. If ORIGIN = "U" and the source function is a built-in function, contains the value "SYSIBM". The null value if ORIGIN is not "U".
SOURCEMODULENAME	VARCHAR (128)	Y	Contains the module name of the specific name of the source function if ORIGIN = "U" and the source function is a user-defined function defined in a module; the null value otherwise.
SOURCESPECIFIC	VARCHAR (128)	Y	If ORIGIN = "U" and the source function is a user-defined function, contains the unqualified specific name of the source function. If ORIGIN = "U" and the source function is a built-in function, contains the value "N/A for built-in". The null value if ORIGIN is not "U".
PUBLISHED	CHAR (1)	Y	Indicates whether the module routine can be invoked outside its module. • N = The module routine is not published • Y = The module routine is published • Blank = Not applicable
DETERMINISTIC	CHAR (1)	Y	• N = Results are not deterministic (same parameters might give different results in different routine calls) • Y = Results are deterministic • Blank = ORIGIN is not "A", "E", "F", "Q", or "R"
EXTERNAL_ACTION	CHAR (1)	Y	• E = Function has external side-effects (therefore, the number of invocations is important) • N = No side-effects • Blank = ORIGIN is not "A", "E", "F", "Q", or "R"

Table 239. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
NULLCALL	CHAR (1)	Y	• N = RETURNS NULL ON NULL INPUT (function result is implicitly the null value if one or more operands are null) • Y = CALLED ON NULL INPUT • Blank = ORIGIN is not "A", "E", "Q", or "R"
CAST_FUNCTION	CHAR (1)	Y	• N = Not a cast function • Y = Cast function • Blank = ROUTINETYPE is not "F"
ASSIGN_FUNCTION	CHAR (1)	Y	• N = Not an assignment function • Y = Implicit assignment function • Blank = ROUTINETYPE is not "F"
SCRATCHPAD	CHAR (1)	Y	• N = Routine has no scratchpad • Y = Routine has a scratchpad • Blank = ORIGIN is not "A" or "E", or ROUTINETYPE is "P"
SCRATCHPAD_LENGTH	SMALLINT	Y	Size (in bytes) of the scratchpad for the routine. • -1 = LANGUAGE is "OLEDB" and SCRATCHPAD is "Y" • 0 = SCRATCHPAD is not "Y"
FINALCALL	CHAR (1)	Y	• N = No final call is made • Y = Final call is made to this routine at the runtime end-of-statement • Blank = ORIGIN is not "A" or "E", or ROUTINETYPE is "P"
PARALLEL	CHAR (1)	Y	• N = Routine cannot be executed in parallel • Y = Routine can be executed in parallel • Blank = ORIGIN is not "A" or "E"

Table 239. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
PARAMETER_STYLE	CHAR (8)	Y	Parameter style that was declared when the routine was created. Possible values are: • DB2DARI • DB2GENRL • DB2SQL • GENERAL • GNRLNULL • JAVA • NPSGENRC • SQL • Blanks if ORIGIN is not "E"
FENCED	CHAR (1)	Y	• N = Not fenced • Y = Fenced • Blank = ORIGIN is not "E"
SQL_DATA_ACCESS	CHAR (1)	Y	Indicates what type of SQL statements, if any, the database manager should assume is contained in the routine. • C = Contains SQL (simple expressions with no subqueries only) • M = Contains SQL statements that modify data • N = Does not contain SQL statements • R = Contains read-only SQL statements • Blank = ORIGIN is not "E", "F", "Q", or "R"
DBINFO	CHAR (1)	Y	Indicates whether a DBINFO parameter is passed to an external routine. • N = DBINFO is not passed • Y = DBINFO is passed • Blank = ORIGIN is not "E"
PROGRAMTYPE	CHAR (1)	Y	Indicates how the external routine is invoked. • M = Main • S = Subroutine • Blank = ORIGIN is "A" or "F"

Table 239. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
COMMIT_ON_RETURN	CHAR (1)	Y	Indicates whether the transaction is committed on successful return from this procedure. • N = The unit of work is not committed • Y = The unit of work is committed • Blank = ROUTINETYPE is not "P"
AUTONOMOUS	CHAR (1)	Y	Indicates whether or not the routine executes autonomously. • N = Routine does not execute autonomously from invoking transaction • Y = Routine executes autonomously from invoking transaction • Blank = ROUTINETYPE is not "P"
RESULT_SETS	SMALLINT	Y	Estimated maximum number of result sets.
SPEC_REG	CHAR (1)	Y	Indicates the special registers values that are used when the routine is called. • I = Inherited special registers • Blank = PARAMETER_STYLE is "DB2DARI" or ORIGIN is not "E", "Q", or "R"
FEDERATED	CHAR (1)	Y	Indicates whether or not federated objects can be accessed from the routine. • Y = Federated objects can be accessed • Blank = ORIGIN is not "F"
THREADSAFE	CHAR (1)	Y	Indicates whether or not the routine can run in the same process as other routines. • N = Routine is not threadsafe • Y = Routine is threadsafe • Blank = ORIGIN is not "E"

Table 239. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
VALID	CHAR (1)	Y	Applies to LANGUAGE = "SQL" and routines having parameters with default or ORIGIN = "A"; blank otherwise. • N = Routine needs rebinding • X = Routine is inoperative and must be recreated • Y = Routine is valid
MODULEROUTINEIMPLEMENTED	CHAR (1)	Y	• N = Module routine body is not implemented • Y = Module routine body is implemented • Blank = ROUTINEMODULENAME is null value
METHODIMPLEMENTED	CHAR (1)	Y	• N = Method body is not implemented • Y = Method body is implemented • Blank = ROUTINETYPE is not "M" or ROUTINEMODULENAME is not the null value
METHODEFFECT	CHAR (2)	Y	• CN = Constructor method • MU = Mutator method • OB = Observer method • Blanks = Not a system method
TYPE_PRESERVING	CHAR (1)	Y	• N = Return type is the declared return type of the method • Y = Return type is governed by a "type-preserving" parameter; all system-generated mutator methods are type-preserving • Blank = ROUTINETYPE is not "M"
WITH_FUNC_ACCESS	CHAR (1)	Y	• N = This method cannot be invoked by using functional notation • Y = This method can be invoked by using functional notation; that is, the "WITH FUNCTION ACCESS" attribute is specified • Blank = ROUTINETYPE is not "M"
OVERRIDDEN_METHODID	INTEGER	Y	Identifier for the overridden method when the OVERRIDING option is specified for a user-defined method. The null value if ROUTINETYPE is not "M".

Table 239. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SUBJECT_TYPESCHEMA	VARCHAR (128)	Y	Schema name of the subject type for the user-defined method. The null value if ROUTINETYPE is not "M".
SUBJECT_TYPENAME	VARCHAR (128)	Y	Unqualified name of the subject type for the user-defined method. The null value if ROUTINETYPE is not "M".
CLASS	VARCHAR (384)	Y	For LANGUAGE JAVA, CLR, or OLE, this is the class that implements this routine; null value otherwise.
JAR_ID	VARCHAR (128)	Y	For LANGUAGE JAVA, this is the JAR_ID of the installed jar file that implements this routine if a jar file was specified at routine creation time; null value otherwise. For LANGUAGE CLR, this is the assembly file that implements this routine.
JARSCHEMA	VARCHAR (128)	Y	For LANGUAGE JAVA when a JAR_ID is present, this is the schema name of the jar file that implements this routine; null value otherwise.
JAR_SIGNATURE	VARCHAR (2048)	Y	For LANGUAGE JAVA, this is the method signature of this routine's specified Java method. For LANGUAGE CLR, this is a reference field for this CLR routine. Null value otherwise.
CREATE_TIME	TIMESTAMP	Y	Time at which the routine was created.
ALTER_TIME	TIMESTAMP	Y	Time at which the routine was last altered.
FUNC_PATH	CLOB (2K)	Y	SQL path in effect when the routine was defined. The null value if LANGUAGE is not "SQL", ORIGIN is not "A", and no parameters have defaults.
QUALIFIER	VARCHAR (128)	Y	Value of the default schema at the time of object definition. Used to complete any unqualified references.
IOS_PER_INVOC	DOUBLE	Y	Estimated number of inputs/outputs (I/Os) per invocation; 0 is the default; -1 if not known.
INSTS_PER_INVOC	DOUBLE	Y	Estimated number of instructions per invocation; 450 is the default; -1 if not known.
IOS_PER_ARGBYTE	DOUBLE	Y	Estimated number of I/Os per input argument byte; 0 is the default; -1 if not known.

Table 239. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
INSTS_PER_ARGBYTE	DOUBLE	Y	Estimated number of instructions per input argument byte; 0 is the default; -1 if not known.
PERCENT_ARGBYTES	SMALLINT	Y	Estimated average percent of input argument bytes that the routine will actually read; 100 is the default; -1 if not known.
INITIAL_IOS	DOUBLE	Y	Estimated number of I/Os performed the first time that the routine is invoked; 0 is the default; -1 if not known.
INITIAL_INSTS	DOUBLE	Y	Estimated number of instructions executed the first time the routine is invoked; 0 is the default; -1 if not known.
CARDINALITY	BIGINT	Y	Predicted cardinality of a table function; -1 if not known, or if the routine is not a table function.
SELECTIVITY[2]	DOUBLE	Y	For user-defined predicates; -1 if there are no user-defined predicates.
RESULT_COLS	SMALLINT	Y	For a table function (ROUTINETYPE = "F" and FUNCTIONTYPE = "T"), contains the number of columns in the result table; for a procedure (ROUTINETYPE = "P"), contains 0; contains 1 otherwise.
IMPLEMENTATION	VARCHAR (762)	Y	The value from the EXTERNAL NAME clause if ORIGIN="E" (an external routine); the name and signature of the source function if ORIGIN = "U" and the source function is built-in; the entry point in the library if LIB_ID is not null and LANGUAGE="SQL" (a compiled SQL routine); the null value otherwise.
LIB_ID	INTEGER	Y	Internal identifier for compiled SQL routines. Otherwise the null value.
TEXT_BODY_OFFSET	INTEGER	Y	If LANGUAGE = "SQL", the offset to the start of the compiled SQL routine body in the full text of the CREATE statement; -1 if LANGUAGE is not "SQL" or the SQL routine is not compiled.
TEXT	CLOB (2M)	Y	If LANGUAGE = "SQL" or ORIGIN = "A", the full text of the CREATE FUNCTION, CREATE METHOD, or CREATE PROCEDURE statement; null value otherwise.

Table 239. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
NEWSAVEPOINTLEVEL	CHAR (1)	Y	Indicates whether the routine initiates a new savepoint level when it is invoked. • N = A new savepoint level is not initiated when the routine is invoked; the routine uses the existing savepoint level • Y = A new savepoint level is initiated when the routine is invoked • Blank = Not applicable
DEBUG_MODE[3]	VARCHAR (8)	Y	Indicates whether the routine can be debugged using the debugger that is integrated with the database. • DISALLOW = Routine is not debuggable • ALLOW = Routine is debuggable, and can participate in a client debug session with the integrated debugger • DISABLE = Routine is not debuggable, and this setting cannot be altered without dropping and recreating the routine • Blank = Routine type is not currently supported by the integrated debugger
TRACE_LEVEL	VARCHAR (1)	Y	Reserved for future use.
DIAGNOSTIC_LEVEL	VARCHAR (1)	Y	Reserved for future use.
CHECKOUT_USERID	VARCHAR (128)	Y	ID of the user who performed a checkout of the object; the null value if the object is not checked out.
PRECOMPILE_OPTIONS	VARCHAR (1024)	Y	The precompile and bind options that were in effect when the compiled SQL routine was created. The null value if LANGUAGE is not "SQL" or if the SQL routine is not compiled.
COMPILE_OPTIONS	VARCHAR (1024)	Y	The value of the SQL_CCFLAGS special register that was in effect when the compiled SQL routine was created and inquiry directives were present. An empty string if no inquiry directives were present in the compiled SQL routine. The null value if LANGUAGE is not "SQL" or if the SQL routine is not compiled.

Table 239. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
EXECUTION_CONTROL	CHAR (1)	Y	Execution control mode of a common language runtime (CLR) routine. Possible values are: • N = Network • R = Fileread • S = Safe • U = Unsafe • W = Filewrite • Blank = LANGUAGE is not "CLR"
CODEPAGE	SMALLINT	Y	Routine code page, which specifies the default code page used for all character parameter types, result types, and local variables within the routine body.
COLLATIONSCHEMA	VARCHAR (128)	Y	Schema name of the collation for the routine.
COLLATIONNAME	VARCHAR (128)	Y	Unqualified name of the collation for the routine.
COLLATIONSCHEMA_ORDERBY	VARCHAR (128)	Y	Schema name of the collation for ORDER BY clauses in the routine.
COLLATIONNAME_ORDERBY	VARCHAR (128)	Y	Unqualified name of the collation for ORDER BY clauses in the routine.
ENCODING_SCHEME	CHAR (1)	Y	Encoding scheme of the routine, as specified in the PARAMETER CCSID clause. Possible values are: • A = ASCII • U = UNICODE • Blank = PARAMETER CCSID clause was not specified
LAST_REGEN_TIME	TIMESTAMP	Y	Time at which the SQL routine packed descriptor was last regenerated.
INHERITLOCKREQUEST	CHAR (1)	Y	• N = This function or method cannot be invoked in the context of an SQL statement that includes a lock-request-clause as part of a specified isolation-clause • Y = This function or method inherits the isolation level of the invoking statement; it also inherits the specified lock-request-clause • Blank = LANGUAGE is not "SQL" or ROUTINETYPE is "P"
DEFINER[4]	VARCHAR (128)	Y	Authorization ID of the owner of the routine.

Table 239. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SECURE	CHAR (1)	Y	Indicates whether the function is secure for row and column access control • N = Not secure • Y = Secure • Blank = ROUTINETYPE is not "F"
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.
STAYRESIDENT	CHAR (1)		The STAYRESIDENT option of the routine, which determines whether the routine library is to be deleted from memory when the routine ends • N = The routine library is to be deleted from memory after the routine terminates • Blank = STAY RESIDENT NO clause not specified
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. During database upgrade, the SELECTIVITY column will be set to -1 in the packed descriptor and system catalogs for all user-defined routines. For a user-defined predicate, the selectivity in the system catalog will be -1. In this case, the selectivity value used by the optimizer is 0.01.
2. For Java routines, the DEBUG_MODE setting does not indicate whether the Java routine was actually compiled in debug mode, or whether a debug Jar was installed at the server.
3. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.ROUTINESFEDERATED

Each row represents a federated procedure.

Table 240. SYSCAT.ROUTINESFEDERATED Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESCHEMA	VARCHAR (128)		Schema name of the routine if ROUTINEMODULEID is null; otherwise schema name of the module to which the routine belongs.
ROUTINENAME	VARCHAR (128)		Unqualified name of the routine.
ROUTINETYPE	CHAR (1)		Type of routine. • P = Procedure
OWNER	VARCHAR (128)		Authorization ID of the owner of the routine.

Table 240. SYSCAT.ROUTINESFEDERATED Catalog View (continued)

Column Name	Data Type	Nullable	Description
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
SPECIFICNAME	VARCHAR (128)		Name of the routine instance (might be system-generated).
ROUTINEID	INTEGER		Identifier for the routine.
PARM_COUNT	SMALLINT		Number of routine parameters; -1 if parameters specified as VARARGS.
DETERMINISTIC	CHAR (1)		• N = Results are not deterministic (same parameters might give different results in different routine calls) • Y = Results are deterministic
EXTERNAL_ACTION	CHAR (1)		• E = Routine has external side-effects (therefore, the number of invocations is important) • N = No side-effects
SQL_DATA_ACCESS	CHAR (1)		Indicates what type of SQL statements, if any, the database manager should assume is contained in the routine. • C = Contains SQL (simple expressions with no subqueries only) • M = Contains SQL statements that modify data • N = Does not contain SQL statements • R = Contains read-only SQL statements
COMMIT_ON_RETURN	CHAR (1)		Indicates whether the transaction is committed on successful return from this procedure. • N = The unit of work is not committed • Y = The unit of work is committed • Blank = ROUTINETYPE is not 'P'
RESULT_SETS	SMALLINT		Estimated maximum number of result sets.
CREATE_TIME	TIMESTAMP		Time at which the routine was created.
ALTER_TIME	TIMESTAMP		Time at which the routine was last altered.
QUALIFIER	VARCHAR (128)		Value of the default schema at the time of object definition. Used to complete any unqualified references.
RESULT_COLS	SMALLINT		For a procedure (ROUTINETYPE = 'P'), contains 0; contains 1 otherwise.

Table 240. SYSCAT.ROUTINESFEDERATED Catalog View (continued)

Column Name	Data Type	Nullable	Description
CODEPAGE	SMALLINT		Routine code page, which specifies the default code page used for all character parameter types, result types, and local variables within the routine body.
LAST_REGEN_TIME	TIMESTAMP		Time at which the SQL routine packed descriptor was last regenerated.
REMOTE_PROCEDURE	VARCHAR (128)	Y	Unqualified name of the source procedure for which the federated routine was created.
REMOTE_SCHEMA	VARCHAR (128)	Y	Schema name of the source procedure for which the federated routine was created.
SERVERNAME	VARCHAR (128)	Y	Name of the data source that contains the source procedure for which the federated routine was created.
REMOTE_PACKAGE	VARCHAR (128)	Y	Name of the package to which the source procedure belongs (applies only to wrappers for Oracle data sources).
REMOTE_PROCEDURE_ ALTER_TIME	VARCHAR (128)	Y	Reserved for future use.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.ROWFIELDS

Each row represents a field that is defined for a user-defined row data type.

Table 241. SYSCAT.ROWFIELDS Catalog View

Column Name	Data Type	Nullable	Description
TYPESCHEMA	VARCHAR (128)		Schema name of the row data type that includes the field.
TYPEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the row data type belongs. The null value if not a module row data type.
TYPENAME	VARCHAR (128)		Unqualified name of the row data type that includes the field.
FIELDNAME	VARCHAR (128)		Field name.
FIELDTYPESCHEMA	VARCHAR (128)		Schema name of the data type of the field.
FIELDTYPEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the data type of the field belongs. The null value if the field data type is not a module user-defined data type.
FIELDTYPENAME	VARCHAR (128)		Unqualified name of the data type of the field.
ORDINAL	SMALLINT		Position of the field in the definition of the row data type, starting with 0.

Table 241. SYSCAT.ROWFIELDS Catalog View (continued)

Column Name	Data Type	Nullable	Description
LENGTH	INTEGER		Length of the field data type. For decimal types, contains the precision.
SCALE	SMALLINT		For decimal types, contains the scale of the field data type; for timestamp types, contains the timestamp precision of the field data type; 0 otherwise.
TYPESTRINGUNITS	VARCHAR (11)	Y	In a Unicode database, the string units that apply to a character string or graphic string data type. Otherwise, the null value.
STRINGUNITSLENGTH	INTEGER	Y	In a Unicode database, the declared number of string units for a character string or graphic string data type. Otherwise, the null value.
CODEPAGE	SMALLINT		For string types, denotes the code page; 0 indicates FOR BIT DATA; 0 for non-string types.
COLLATIONSCHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the field; the null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the field; the null value otherwise.
NULLS	CHAR (1)		Reserved for future use.
QUALIFIER	VARCHAR (128)	Y	Reserved for future use.
FUNC_PATH	CLOB (2K)	Y	Reserved for future use.
DEFAULT	CLOB (64K)	Y	Reserved for future use.
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.

SYSCAT.SCHEMAAUTH

Each row represents a user, group, or role that has been granted one or more privileges on a schema.

Table 242. SYSCAT.SCHEMAAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of a privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of a privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user

Table 242. SYSCAT.SCHEMAAUTH Catalog View (continued)

Column Name	Data Type	Nullable	Description
SCHEMANAME	VARCHAR (128)		Name of the schema to which this privilege applies.
ALTERINAUTH	CHAR (1)		Privilege to alter or comment on objects in the named schema. • G = Held and grantable • N = Not held • Y = Held
CREATEINAUTH	CHAR (1)		Privilege to create objects in the named schema. • G = Held and grantable • N = Not held • Y = Held
DROPINAUTH	CHAR (1)		Privilege to drop objects from the named schema. • G = Held and grantable • N = Not held • Y = Held
SELECTINAUTH	CHAR (1)		Implicit SELECT privilege on all of the existing and future tables or views defined in the schema. • G = Held and grantable • N = Not held • Y = Held
INSERTINAUTH	CHAR (1)		Implicit INSERT privilege on all of the existing and future tables or updatable views defined in the schema. • G = Held and grantable • N = Not held • Y = Held
UPDATEINAUTH	CHAR (1)		Implicit UPDATE privilege on all of the existing and future tables or updatable views defined in the schema. • G = Held and grantable • N = Not held • Y = Held

Table 242. SYSCAT.SCHEMAAUTH Catalog View (continued)

Column Name	Data Type	Nullable	Description
DELETEINAUTH	CHAR (1)		Implicit DELETE privilege on all of the existing and future tables or updatable views defined in the schema. • G = Held and grantable • N = Not held • Y = Held
EXECUTEINAUTH	CHAR (1)		Implicit EXECUTE privilege on all of the existing and future routines, packages and module objects defined in the schema • G = Held and grantable • N = Not held • Y = Held
SCHEMAADMAUTH	CHAR (1)		SCHEMAADM authority. • N = Not held • Y = Held
ACCESSCTRLAUTH	CHAR (1)		Schema ACCESSCTRL authority to grant and revoke schema object privileges. • N = Not held • Y = Held
DATAACCESSAUTH	CHAR (1)		Schema DATAACCESS authority to access data. • N = Not held • Y = Held
LOADAUTH	CHAR (1)		Schema LOAD authority to use the load utility. • N = Not held • Y = Held

SYSCAT.SCHEMATA

Each row represents a schema.

Table 243. SYSCAT.SCHEMATA Catalog View

Column Name	Data Type	Nullable	Description
SCHEMANAME	VARCHAR (128)		Name of the schema.
OWNER	VARCHAR (128)		Authorization ID of the owner of the schema.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user

Table 243. SYSCAT.SCHEMATA Catalog View (continued)

Column Name	Data Type	Nullable	Description
DEFINER	VARCHAR (128)		Authorization ID of the definer of the schema or authorization ID of the owner of the schema if the ownership of the schema has been transferred.
DEFINERTYPE	CHAR (1)		• S = The definer is the system • U = The definer is an individual user
CREATE_TIME	TIMESTAMP		Time at which the schema was created.
AUDITPOLICYID	INTEGER	Y	Identifier for the audit policy.
AUDITPOLICYNAME	VARCHAR (128)		Name of the audit policy.
AUDITEXCEPTIONENABLED	CHAR (1)		Reserved for future use.
DATACAPTURE	CHAR (1)		Indicates the default data capture setting for new tables that are created within this schema. • N = New tables do not participate in data capture • Y = New tables participate in data capture, including replication of all columns
ROWMODIFICATIONTRACKING	VARCHAR (1)	N	Indicates tables in schema are enabled for logical backup • Y = Indicates the schema is enabled for row modification tracking • N = Indicates the schema is not enabled for row modification tracking
QUIESCED	VARCHAR (1)	N	Indicates the schema is locked by the in-progress LOGICAL_RESTORE operation • Y = Indicates the schema is locked by in-progress LOGICAL_RESTORE operation • N =Indicates the schema is not locked by in-progress LOGICAL_RESTORE operation
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.SCPREFTBSPACES

Each row represents a preferred system temporary table space for the service class.

Table 244. SYSCAT.SCPREFTBSPACES Catalog View

Column Name	Data Type	Nullable	Description
SERVICECLASSNAME	VARCHAR (128)		Name of the service class.
PARENTSERVICECLASSNAME	VARCHAR (128)		Service class name of the parent service superclass.

Table 244. SYSCAT.SCPREFTBSPACES Catalog View (continued)

Column Name	Data Type	Nullable	Description
TBSPACE	VARCHAR (128)		Name of the table space.
SERVICECLASSID	SMALLINT		Identifier for the service class.
PARENTSERVICECLASSID	SMALLINT		Identifier for the parent service class for the service class. 0 if the service class is a super service class.
TBSPACEID	INTEGER		Identifier for the table space.
DATATYPE	CHAR (1)		Type of data that can be stored in this table space. • A = All types of permanent data; regular table space • L = All types of permanent data; large table space • T = System temporary tables only • U = Created temporary tables or declared temporary tables only

SYSCAT.SECURITYLABELACCESS

Each row represents a security label that was granted to the database authorization ID.

Table 245. SYSCAT.SECURITYLABELACCESS Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the security label.
GRANTEE	VARCHAR (128)		Holder of the security label.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
SECLABELID	INTEGER		Identifier for the security label. For the name of the security label, select the SECLABELNAME column for the corresponding SECLABELID value in the SYSCAT.SECURITYLABELS catalog view.
SECPOLICYID	INTEGER		Identifier for the security policy that is associated with the security label. For the name of the security policy, select the SECPOLICYNAME column for the corresponding SECPOLICYID value in the SYSCAT.SECURITYPOLICIES catalog view.
ACCESSTYPE	CHAR (1)		• B = Both read and write access • R = Read access • W = Write access
GRANT_TIME	TIMESTAMP		Time at which the security label was granted.

SYSCAT.SECURITYLABELCOMPONENTELEMENTS

Each row represents an element value for a security label component.

Table 246. SYSCAT.SECURITYLABELCOMPONENTELEMENTS Catalog View

Column Name	Data Type	Nullable	Description
COMPID	INTEGER		Identifier for the security label component.
ELEMENTVALUE	VARCHAR (32)		Element value for the security label component.
ELEMENTVALUEENCODING	CHAR (8) FOR BIT DATA		Encoded form of the element value.
PARENTELEMENTVALUE	VARCHAR (32)	Y	Name of the parent of an element for tree components; the null value for set and array components, and for the ROOT node of a tree component.

SYSCAT.SECURITYLABELCOMPONENTS

Each row represents a security label component.

Table 247. SYSCAT.SECURITYLABELCOMPONENTS Catalog View

Column Name	Data Type	Nullable	Description
COMPNAME	VARCHAR (128)		Name of the security label component.
COMPID	INTEGER		Identifier for the security label component.
COMPTYPE	CHAR (1)		Security label component type. • A = Array • S = Set • T = Tree
NUMELEMENTS	INTEGER		Number of elements in the security label component.
CREATE_TIME	TIMESTAMP		Time at which the security label component was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.SECURITYLABELS

Each row represents a security label.

Table 248. SYSCAT.SECURITYLABELS Catalog View

Column Name	Data Type	Nullable	Description
SECLABELNAME	VARCHAR (128)		Name of the security label.
SECLABELID	INTEGER		Identifier for the security label.
SECPOLICYID	INTEGER		Identifier for the security policy to which the security label belongs.
SECLABEL	SYSPROC. DB2SECURITYLABEL		Internal representation of the security label.

Table 248. SYSCAT.SECURITYLABELS Catalog View (continued)

Column Name	Data Type	Nullable	Description
CREATE_TIME	TIMESTAMP		Time at which the security label was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.SECURITYPOLICIES

Each row represents a security policy.

Table 249. SYSCAT.SECURITYPOLICIES Catalog View

Column Name	Data Type	Nullable	Description
SECPOLICYNAME	VARCHAR (128)		Name of the security policy.
SECPOLICYID	INTEGER		Identifier for the security policy.
NUMSECLABELCOMP	INTEGER		Number of security label components in the security policy.
RWSECLABELREL	CHAR (1)		Relationship between the security labels for read and write access granted to the same authorization ID. • S = The security label for write access granted to a user is a subset of the security label for read access granted to that same user
NOTAUTHWRITESECLABEL	CHAR (1)		Action to take when a user is not authorized to write the security label that is specified in the INSERT or UPDATE statement. • O = Override • R = Restrict
CREATE_TIME	TIMESTAMP		Time at which the security policy was created.
GROUPAUTHS	CHAR (1)		Indicates if authorizations of security labels and exemptions granted to an authorization ID that represents a group will be used or ignored. • I = Ignored • U = Used
ROLEAUTHS	CHAR (1)		Indicates if authorizations of security labels and exemptions granted to an authorization ID that represents a role will be used or ignored. • I = Ignored • U = Used
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.SECURITYPOLICYCOMPONENTRULES

Each row represents the read and write access rules for a security label component of the security policy.

Table 250. SYSCAT.SECURITYPOLICYCOMPONENTRULES Catalog View

Column Name	Data Type	Nullable	Description
SECPOLICYID	INTEGER		Identifier for the security policy.
COMPID	INTEGER		Identifier for the security label component of the security policy.
ORDINAL	INTEGER		Position of the security label component as it appears in the security policy, starting with 1.
READACCESSRULENAME	VARCHAR (128)		Name of the read access rule that is associated with the security label component.
READACCESSRULETEXT	VARCHAR (512)		Text of the read access rule that is associated with the security label component.
WRITEACCESSRULENAME	VARCHAR (128)		Name of the write access rule that is associated with the security label component.
WRITEACCESSRULETEXT	VARCHAR (512)		Text of the write access rule that is associated with the security label component.

SYSCAT.SECURITYPOLICYEXEMPTIONS

Each row represents a security policy exemption that was granted to a database authorization ID.

Table 251. SYSCAT.SECURITYPOLICYEXEMPTIONS Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the exemption.
GRANTEE	VARCHAR (128)		Holder of the exemption.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
SECPOLICYID	INTEGER		Identifier for the security policy for which the exemption was granted. For the name of the security policy, select the SECPOLICYNAME column for the corresponding SECPOLICYID value in the SYSCAT.SECURITYPOLICIES catalog view.
ACCESSRULENAME	VARCHAR (128)		Name of the access rule for which the exemption was granted.
ACCESSTYPE	CHAR (1)		Type of access to which the rule applies. • R = Read access • W = Write access

Table 251. SYSCAT.SECURITYPOLICYEXEMPTIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
ORDINAL	INTEGER		Position of the security label component in the security policy to which the rule applies.
ACTIONALLOWED	CHAR (1)		If the rule is DB2LBACWRITEARRAY, then: • D = Write down • U = Write up Blank otherwise.
GRANT_TIME	TIMESTAMP		Time at which the exemption was granted.

SYSCAT.SEQUENCEAUTH

Each row represents a user, group, or role that has been granted one or more privileges on a sequence.

Table 252. SYSCAT.SEQUENCEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of a privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of a privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
SEQSCHEMA	VARCHAR (128)		Schema name of the sequence.
SEQNAME	VARCHAR (128)		Unqualified name of the sequence.
ALTERAUTH	CHAR (1)		Privilege to alter the sequence. • G = Held and grantable • N = Not held • Y = Held
USAGEAUTH	CHAR (1)		Privilege to reference the sequence. • G = Held and grantable • N = Not held • Y = Held

SYSCAT.SEQUENCES

Each row represents a sequence or alias.

Table 253. SYSCAT.SEQUENCES Catalog View

Column Name	Data Type	Nullable	Description
SEQSCHEMA	VARCHAR (128)		Schema name of the sequence.

Table 253. SYSCAT.SEQUENCES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SEQNAME	VARCHAR (128)		Unqualified name of the sequence.
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the sequence.
DEFINERTYPE	CHAR (1)		• S = The definer is the system • U = The definer is an individual user
OWNER	VARCHAR (128)		Authorization ID of the owner of the sequence.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
SEQID	INTEGER		Identifier for the sequence or alias.
SEQTYPE	CHAR (1)		Type of sequence. • A = Alias • I = Identity sequence • S = Sequence
BASE_SEQSCHEMA	VARCHAR (128)	Y	If SEQTYPE is 'A', contains the schema name of the sequence or alias that is referenced by this alias; the null value otherwise.
BASE_SEQNAME	VARCHAR (128)	Y	If SEQTYPE is 'A', contains the unqualified name of the sequence or alias that is referenced by this alias; the null value otherwise.
INCREMENT	DECIMAL (31,0)	Y	Increment value. The null value if the sequence is an alias.
START	DECIMAL (31,0)	Y	Start value of the sequence. The null value if the sequence is an alias.
MAXVALUE	DECIMAL (31,0)	Y	Maximum value of the sequence. The null value if the sequence is an alias.
MINVALUE	DECIMAL (31,0)	Y	Minimum value of the sequence. The null value if the sequence is an alias.
NEXTCACHEFIRSTVALUE	DECIMAL (31,0)	Y	The first value available to be assigned in the next cache block. If no caching, the next value available to be assigned.
CYCLE	CHAR (1)		Indicates whether or not the sequence can continue to generate values after reaching its maximum or minimum value. • N = Sequence cannot cycle • Y = Sequence can cycle • Blank = Sequence is an alias.

Table 253. SYSCAT.SEQUENCES Catalog View (continued)

Column Name	Data Type	Nullable	Description
CACHE	INTEGER		Number of sequence values to pre-allocate in memory for faster access. 0 indicates that values of the sequence are not to be preallocated. In a partitioned database, this value applies to each database partition. -1 if the sequence is an alias.
ORDER	CHAR (1)		Indicates whether or not the sequence numbers must be generated in order of request. • N = Sequence numbers are not required to be generated in order of request • Y = Sequence numbers must be generated in order of request • Blank = Sequence is an alias.
DATATYPEID	INTEGER		For built-in types, the internal identifier of the built-in type. For distinct types, the internal identifier of the distinct type. 0 if the sequence is an alias.
SOURCETYPEID	INTEGER		For a built-in type or if the sequence is an alias, this has a value of 0. For a distinct type, this is the internal identifier of the built-in type that is the source type for the distinct type.
CREATE_TIME	TIMESTAMP		Time at which the sequence was created.
ALTER_TIME	TIMESTAMP		Time at which the sequence was last altered.
PRECISION	SMALLINT		Precision of the data type of the sequence. Possible values are: • 5 = SMALLINT • 10 = INTEGER • 19 = BIGINT For DECIMAL, it is the precision of the specified DECIMAL data type. 0 if the sequence is an alias.
ORIGIN	CHAR (1)		Origin of the sequence. • S = System-generated sequence • U = User-generated sequence
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.SERVEROPTIONS

Each row represents a server-specific option value.

Table 254. SYSCAT.SERVEROPTIONS Catalog View

Column Name	Data Type	Nullable	Description
WRAPNAME	VARCHAR (128)	Y	Name of the wrapper.
SERVERNAME	VARCHAR (128)	Y	Uppercase name of the server.
SERVERTYPE	VARCHAR (30)	Y	Type of server.
SERVERVERSION	VARCHAR (18)	Y	Server version.
CREATE_TIME	TIMESTAMP		Time at which the entry was created.
OPTION	VARCHAR (128)		Name of the server option.
SETTING	VARCHAR (2048)		Value of the server option.
SERVEROPTIONKEY	VARCHAR (18)		Uniquely identifies a row.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.SERVERS

Each row represents a data source.

Table 255. SYSCAT.SERVERS Catalog View

Column Name	Data Type	Nullable	Description
WRAPNAME	VARCHAR (128)		Name of the wrapper.
SERVERNAME	VARCHAR (128)		Uppercase name of the server.
SERVERTYPE	VARCHAR (30)	Y	Type of server.
SERVERVERSION	VARCHAR (18)	Y	Server version.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.SERVICECLASSES

Each row represents a service class.

Table 256. SYSCAT.SERVICECLASSES Catalog View

Column Name	Data Type	Nullable	Description
SERVICECLASSNAME	VARCHAR (128)		Name of the service class.
PARENTSERVICECLASSNAME	VARCHAR (128)	Y	Service class name of the parent service superclass.
SERVICECLASSID	SMALLINT		Identifier for the service class.
PARENTID	SMALLINT		Identifier for the parent service class for this service class. 0 if this service class is a super service class.
CREATE_TIME	TIMESTAMP		Time when the service class was created.
ALTER_TIME	TIMESTAMP		Time when the service class was last altered.

Table 256. SYSCAT.SERVICECLASSES Catalog View (continued)

Column Name	Data Type	Nullable	Description
ENABLED	CHAR (1)		State of the service class. • N = Disabled • Y = Enabled
AGENTPRIORITY	SMALLINT		This column is no longer used and will be removed in a future release.
PREFETCHPRIORITY	CHAR (1)		Prefetch priority of the agents in the service class. • H = High • L = Low • M = Medium • Blank = not set
MAXDEGREE	SMALLINT	Y	Possible values: • 1 - 32767 = Maximum degree of parallelism for the service class • -1 = MAXIMUM DEGREE NONE was specified for the service class • -2 = MAXIMUM DEGREE DEFAULT was specified for the service class
BUFFERPOOLPRIORITY	CHAR (1)		Bufferpool priority of the agents in the service class: • H = High • L = Low • M = Medium • Blank = Not set
INBOUNDCORRELATOR	VARCHAR (128)	Y	For future use.
OUTBOUNDCORRELATOR	VARCHAR (128)	Y	String used to associate the service class with an operating system workload manager service class.
COLLECTAGGACTDATA	CHAR (1)		Specifies what aggregate activity data should be captured for the service class by the applicable event monitor. • B = Collect base aggregate activity data • E = Collect extended aggregate activity data • N = None
COLLECTAGGREQDATA	CHAR (1)		Specifies what aggregate request data should be captured for the service class by the applicable event monitor. • B = Collect base aggregate request data • N = None

Table 256. SYSCAT.SERVICECLASSES Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLLECTACTDATA	CHAR (1)		Specifies what activity data should be collected by the applicable event monitor. • D = Activity data with details • N = None • S = Activity data with details and section environment • V = Activity data with details and values • W = Activity data without details • X = Activity data with details, section environment, and values
COLLECTACTPARTITION	CHAR (1)		Specifies where activity data is collected. • C = Coordinator member of the activity • D = All members
COLLECTREQMETRICS	CHAR (1)		Specifies the monitoring level for requests submitted by a connection that is associated with the service superclass. • B = Collect base request metrics • E = Collect extended request metrics • N = None
CPUSHARES	INTEGER		The number of CPU shares allocated to this service class.
CPUSHARETYPE	CHAR (1)		Specifies the type of CPU shares. • S = Soft shares • H = Hard shares
CPULIMIT	SMALLINT		The maximum percentage of the CPU resource that can be allocated to the service class; -1 if there is no CPU limit.
SORTMEMORYPRIORITY	CHAR (1)		Reserved for future use.
SECTIONACTUALSOPTIONS	VARCHAR (32)		Specifies what section actuals are collected during the execution of a section. The first position in the string represents whether the collection of section actuals is enabled. • B = Enabled and collect basic operator cardinality counts and statistics for each object referenced by the section (DML statements only). • N = Not enabled. The second position is always 'N' and reserved for future use.

Table 256. SYSCAT.SERVICECLASSES Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLLECTAGGUOWDATA	CHAR (1)		Specifies what aggregate unit of work data should be captured for the service class by the applicable event monitor. • B = Collect base aggregate unit of work data • N = None
RESOURCESHARES	INTEGER		The number of shares of resources this service class is entitled to when workload manager (WLM) adaptive admission control is enabled. This value is relative to other service classes in the same scope.
RESOURCESHARETYPE	CHAR (1)		Specifies the type of admission shares. • S = Soft shares • H = Hard shares
MINRESOURCESHAREPCT	SMALLINT		Specifies the percentage (0 - 100) of entitled resources reserved by workload manager (WLM) adaptive admission control. WLM adaptive admission control holds these resources in reserve for the service class when other service classes exceed their admission resource entitlement.
ADMISSIONQUEUEORDER	CHAR (1)		Specifies the order of the service class admission queue. • F = First in first out; queued requests are ordered based on the time in which they arrived. • L = Latency; queued requests are ordered based on their estimated execution time (latency) relative to the amount of time they have been queued. • Blank = Not applicable; this service class is a service superclass.
DEGREESCALEBACK	CHAR (1)		Specifies if query degree for queries running with DEGREE ANY can be scaled back at runtime when the system is under high CPU load. • D = Degree scaleback value is inherited from the superclass. This is the default for a service subclass. • Y = Degree scaleback is enabled. This is the default for a service superclass. • N = Degree scaleback is disabled.

Table 256. SYSCAT.SERVICECLASSES Catalog View (continued)

Column Name	Data Type	Nullable	Description
WORKLOADTYPE	SMALLINT	Y	Specifies the workload type for the service class. • NULL = Service class is a subclass • 1 = Custom • 2 = Mixed • 3 = Interactive • 4 = Batch
COLLECTHISTORY	CHAR (1)	Y	Specifies whether activity data with details is collected at the coordinator member. If COLLECT ACTIVITY DATA is independently modified, this value is NULL. • Y = Activity data with details is collected at coordinator member • N = Activity data is not collected
ACTSORTMEMLIMIT	INTEGER		The activity sort memory limit for queries executing in the service class. The default of 100 is shown if you do not specify a sort limit.
REMARKS	VARCHAR (254)	Y	User-provided comments, or NULL.

SYSCAT.STATEMENTS

Each row represents an SQL statement in a package.

Table 257. SYSCAT.STATEMENTS Catalog View

Column Name	Data Type	Nullable	Description
PKGSCHEMA	VARCHAR (128)		Schema name of the package.
PKGNAME	VARCHAR (128)		Unqualified name of the package.
STMTNO	INTEGER		Line number of the SQL statement in the source module of the application program.
SECTNO	SMALLINT		Number of the package section containing the SQL statement.
SEQNO	INTEGER		Always 1.
TEXT	CLOB (2M)		Text of the SQL statement.
UNIQUE_ID	CHAR (8) FOR BIT DATA		Identifier for a specific package when multiple packages having the same name exist.
VERSION	VARCHAR (64)	Y	Version identifier for the package.

SYSCAT.STOGROUPS

Each row represents a storage group object.

Table 258. SYSCAT.STOGROUPS Catalog View

Column Name	Data Type	Nullable	Description
SGNAME	VARCHAR (128)		Name of the storage group.
SGID	INTEGER		Identifier for the storage group.
OWNER	VARCHAR (128)		Authorization ID of the owner of the storage group.
CREATE_TIME	TIMESTAMP		Time at which the storage group was created.
DEFAULTSG	CHAR (1)		Indicates whether the storage group is the default storage group. • N = NO • Y = YES
OVERHEAD	DOUBLE		Controller overhead and disk seek and latency time, in milliseconds (average for the storage paths in this storage group); 0 indicates value is not defined (assigned only by upgrade processing).
DEVICEREADRATE	DOUBLE		Read transfer rate of the device, in megabytes per second (average for the storage paths in this storage group); 0 indicates value is not defined (assigned only by upgrade processing).
WRITEOVERHEAD	DOUBLE	Y	Reserved for future use.
DEVICEWRITERATE	DOUBLE	Y	Reserved for future use.
DATATAG	SMALLINT		Tag to identify data stored in this storage group. Valid user-specified range is 1 through 9; 0 indicates no data tag is specified.
CACHINGTIER	SMALLINT		Reserved for future use.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.STATEMENTTEXTS

Each row represents a user-provided SQL statement for statement thresholds.

Table 259. SYSCAT.STATEMENTTEXTS Catalog View

Column Name	Data Type	Nullable	Description
TEXTID	INTEGER		Identifier for the SQL statement.
TEXT	CLOB (2M)		Text of the SQL statement.

SYSCAT.SURROGATEAUTHIDS

Each row represents a user or a group that has been granted SETSESSIONUSER privilege on a user or PUBLIC.

Table 260. SYSCAT.SURROGATEAUTHIDS Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Authorization ID that granted TRUSTEDID the ability to act as a surrogate. When the TRUSTEDID represents a trusted context object, this field represents the authorization ID that created or altered the trusted context object.
TRUSTEDID	VARCHAR (128)		Identifier for the entity that is trusted to act as a surrogate.
TRUSTEDIDTYPE	CHAR (1)		• C = Trusted context • G = Group • U = User
SURROGATEAUTHID	VARCHAR (128)		Surrogate authorization ID that can be assumed by TRUSTEDID. 'PUBLIC' indicates that TRUSTEDID can assume any authorization ID.
SURROGATEAUTHIDTYPE	CHAR (1)		• G = Group • U = User
AUTHENTICATE	CHAR (1)		• N = No authentication is required • Y = Authentication token is required with the authorization ID to authenticate the user before the authorization ID can be assumed • Blank = TRUSTEDIDTYPE is not 'C'
CONTEXTROLE	VARCHAR (128)	Y	A specific role to be assigned to the assumed authorization ID, which supercedes the default role, if any, that is defined for the trusted context. Null value when TRUSTEDIDTYPE is not 'C'.
GRANT_TIME	TIMESTAMP		Time at which the grant was made .

SYSCAT.TABAUTH

Each row represents a user, group, or role that has been granted one or more privileges on a table, view or nickname.

Table 261. SYSCAT.TABAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user

Table 261. SYSCAT.TABAUTH Catalog View (continued)

Column Name	Data Type	Nullable	Description
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
TABSCHEMA	VARCHAR (128)		Schema name of the table or view.
TABNAME	VARCHAR (128)		Unqualified name of the table or view.
CONTROLAUTH	CHAR (1)		CONTROL privilege. • N = Not held • Y = Held but not grantable
ALTERAUTH	CHAR (1)		Privilege to alter the table; allow a parent table to this table to drop its primary key or unique constraint; allow a table to become a materialized query table that references this table or view in the materialized query; or allow a table that references this table or view in its materialized query to no longer be a materialized query table. • G = Held and grantable • N = Not held • Y = Held
DELETEAUTH	CHAR (1)		Privilege to delete rows from a table or updatable view. • G = Held and grantable • N = Not held • Y = Held
INDEXAUTH	CHAR (1)		Privilege to create an index on a table. • G = Held and grantable • N = Not held • Y = Held
INSERTAUTH	CHAR (1)		Privilege to insert rows into a table or updatable view, or to run the import utility against a table or view. • G = Held and grantable • N = Not held • Y = Held

Table 261. SYSCAT.TABAUTH Catalog View (continued)

Column Name	Data Type	Nullable	Description
REFAUTH	CHAR (1)		Privilege to create and drop a foreign key referencing a table as the parent. • G = Held and grantable • N = Not held • Y = Held
SELECTAUTH	CHAR (1)		Privilege to retrieve rows from a table or view, create views on a table, or to run the export utility against a table or view. • G = Held and grantable • N = Not held • Y = Held
UPDATEAUTH	CHAR (1)		Privilege to run the UPDATE statement against a table or updatable view. • G = Held and grantable • N = Not held • Y = Held

SYSCAT.TABCONST

Each row represents a table constraint of type CHECK, UNIQUE, PRIMARY KEY, or FOREIGN KEY. For table hierarchies, each constraint is recorded only at the level of the hierarchy where the constraint was created.

Table 262. SYSCAT.TABCONST Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR (128)		Name of the constraint.
TABSCHEMA	VARCHAR (128)		Schema name of the table to which this constraint applies.
TABNAME	VARCHAR (128)		Unqualified name of the table to which this constraint applies.
OWNER	VARCHAR (128)		Authorization ID of the owner of the constraint.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
TYPE	CHAR (1)		Indicates the constraint type. • F = Foreign key • I = Functional dependency • K = Check • P = Primary key • U = Unique

Table 262. SYSCAT.TABCONST Catalog View (continued)

Column Name	Data Type	Nullable	Description
ENFORCED	CHAR (1)		• N = Do not enforce constraint • Y = Enforce constraint
TRUSTED	CHAR (1)		If ENFORCED = 'N', specifies whether the data can be trusted to conform to the constraint. • N = Not trusted • Y = Trusted • Blank = Not applicable
CHECKEXISTINGDATA	CHAR (1)		• D = Defer checking any existing data • I = Immediately check existing data • N = Never check existing data
ENABLEQUERYOPT	CHAR (1)		• N = Query optimization is disabled • Y = Query optimization is enabled
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the constraint.
PERIODNAME	VARCHAR (128)	Y	Name of the period used to define this constraint.
PERIODPOLICY	CHAR (1)		If a period name was specified, the constraint uses this period policy. • N = Not applicable • O = Period overlaps not allowed
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.TABDEP

Each row represents a dependency of a view or a materialized query table on some other object. The view or materialized query table depends on the object of type BTYPE of name BNAME, so a change to the object affects the view or materialized query table. Also encodes how privileges on views depend on privileges on underlying tables and views.

Table 263. SYSCAT.TABDEP Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the view or materialized query table.
TABNAME	VARCHAR (128)		Unqualified name of the view or materialized query table.

Table 263. SYSCAT.TABDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
DTYPE	CHAR (1)		Type of the depending object. • S = Materialized query table • T = Table (staging only) • V = View (untyped) • W = Typed view • 7 = Synopsis table
OWNER	VARCHAR (128)		Authorization ID of the creator of the view or materialized query table.
OWNERTYPE	CHAR (1)		• U = The owner is an individual user
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: • A = Table alias • F = Routine • I = Index, if recording dependency on a base table • G = Global temporary table • N = Nickname • O = Privilege dependency on all subtables or subviews in a table or view hierarchy • R = User-defined structured type • S = Materialized query table • T = Table (untyped) • U = Typed table • V = View (untyped) • W = Typed view • Z = XSR object • m = Module • u = Module alias • v = Global variable
BSCHEMA	VARCHAR (128)		Schema name of the object on which the view or materialized query table depends.
BMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object on which there is a dependency belongs. The null value if not a module object.
BNAME	VARCHAR (128)		Unqualified name of the object on which the view or materialized query table depends.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which the view or materialized query table depends.

Table 263. SYSCAT.TABDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
TABAUTH	SMALLINT	Y	If BTYPE is 'N', 'O', 'S', 'T', 'U', 'V', or 'W', encodes the privileges on the underlying table or view on which this view or materialized query table depends; the null value otherwise.
VARAUTH	SMALLINT	Y	If BTYPE is 'v', encodes the privileges on the underlying global variable on which this view or materialized query table depends; the null value otherwise.
DEFINER[1]	VARCHAR (128)		Authorization ID of the creator of the view or materialized query table.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.TABDETACHEDDEP

Each row represents a detached dependency between a detached dependent table and a detached table.

Table 264. SYSCAT.TABDETACHEDDEP Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the detached table.
TABNAME	VARCHAR (128)		Unqualified name of the detached table.
DEPTABSCHEMA	VARCHAR (128)		Schema name of the detached dependent table.
DEPTABNAME	VARCHAR (128)		Unqualified name of the detached dependent table.

SYSCAT.TABLES

Each row represents a table, view, alias, or nickname. Each table or view hierarchy has one additional row that represents the hierarchy table or hierarchy view that implements the hierarchy. Catalog tables and views are included.

Table 265. SYSCAT.TABLES Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the object.
TABNAME	VARCHAR (128)		Unqualified name of the object.
OWNER	VARCHAR (128)		Authorization ID of the owner of the table, view, alias, or nickname.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user

Table 265. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
TYPE	CHAR (1)		Type of object. • A = Alias • G = Created temporary table • H = Hierarchy table • L = Detached table • N = Nickname • S = Materialized query table • T = Table (untyped) • U = Typed table • V = View (untyped) • W = Typed view
STATUS	CHAR (1)		Status of the object. • C = Set integrity pending • N = Normal • X = Inoperative
BASE_TABSCHEMA	VARCHAR (128)	Y	If TYPE = 'A', contains the schema name of the table, view, alias, or nickname that is referenced by this alias; null value otherwise.
BASE_TABNAME	VARCHAR (128)	Y	If TYPE = 'A', contains the unqualified name of the table, view, alias, or nickname that is referenced by this alias; null value otherwise.
ROWTYPESCHEMA	VARCHAR (128)	Y	Schema name of the row type for this table, if applicable; null value otherwise.
ROWTYPENAME	VARCHAR (128)	Y	Unqualified name of the row type for this table, if applicable; null value otherwise.
CREATE_TIME	TIMESTAMP		Time at which the object was created.
ALTER_TIME	TIMESTAMP		Time at which the object was last altered.
INVALIDATE_TIME	TIMESTAMP		Time at which the object was last invalidated.
STATS_TIME	TIMESTAMP	Y	Time at which any change was last made to recorded statistics for this object. The null value if statistics are not collected.
COLCOUNT	SMALLINT		Number of columns, including inherited columns (if any).
TABLEID	SMALLINT		Internal logical object identifier.

Table 265. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
TBSPACEID	SMALLINT		Internal logical identifier for the primary table space for this object.
CARD	BIGINT		Total number of rows in the table; -1 if statistics are not collected.
NPAGES	BIGINT		Total number of pages on which the rows of the table exist; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.
MPAGES	BIGINT		Total number of pages for table metadata. Non-zero only for a table that is organized by column. -1 for a view, an alias, or if statistics are not collected; -2 for subtables or hierarchy tables.
FPAGES	BIGINT		Total number of pages; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.
NPARTITIONS	BIGINT		Reserved for future use.
NFILES	BIGINT		Reserved for future use.
TABLESIZE	BIGINT		Reserved for future use.
OVERFLOW	BIGINT		Total number of overflow records in the table; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.
TBSPACE	VARCHAR (128)	Y	Name of the primary table space for the table. If no other table space is specified, all parts of the table are stored in this table space. The null value for aliases, views, and partitioned tables.
INDEX_TBSPACE	VARCHAR (128)	Y	Name of the table space that holds all indexes created on this table. The null value for aliases, views, and partitioned tables, or if the INDEX IN clause was omitted or specified with the same value as the IN clause of the CREATE TABLE statement.
LONG_TBSPACE	VARCHAR (128)	Y	Name of the table space that holds all long data (LONG or LOB column types) for this table. The null value for aliases, views, and partitioned tables, or if the LONG IN clause was omitted or specified with the same value as the IN clause of the CREATE TABLE statement.
PARENTS	SMALLINT	Y	Number of parent tables for this object; that is, the number of referential constraints in which this object is a dependent.

Table 265. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
CHILDREN	SMALLINT	Y	Number of dependent tables for this object; that is, the number of referential constraints in which this object is a parent.
SELFREFS	SMALLINT	Y	Number of self-referencing referential constraints for this object; that is, the number of referential constraints in which this object is both a parent and a dependent.
KEYCOLUMNS	SMALLINT	Y	Number of columns in the primary key.
KEYINDEXID	SMALLINT	Y	Index identifier for the primary key index: 0 or the null value if there is no primary key.
KEYUNIQUE	SMALLINT		Number of unique key constraints (other than the primary key constraint) defined on this object.
CHECKCOUNT	SMALLINT		Number of check constraints that are defined on this object.
DATACAPTURE	CHAR (1)		• L = Table participates in data replication, including replication of LONG VARCHAR and LONG VARGRAPHIC columns • N = Table does not participate in data replication • Y = Table participates in data replication, excluding replication of LONG VARCHAR and LONG VARGRAPHIC columns

Table 265. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
CONST_CHECKED	CHAR (32)		• Byte 1 represents foreign key constraint. • Byte 2 represents check constraint. • Byte 5 represents materialized query table. • Byte 6 represents generated column. • Byte 7 represents staging table. • Byte 8 represents data partitioning constraint. • Other bytes are reserved for future use. Possible values are: • F = In byte 5, the materialized query table cannot be refreshed incrementally. In byte 7, the content of the staging table is incomplete and cannot be used for incremental refresh of the associated materialized query table. • N = Not checked • U = Checked by user • W = Was in 'U' state when the table was placed in set integrity pending state • Y = Checked by system
PMAP_ID	SMALLINT	Y	Identifier for the distribution map that is currently in use by this table (the null value for aliases or views).
PARTITION_MODE	CHAR (1)		Indicates how data is distributed among database partitions in a partitioned database system. • H = Hashing • R = Replicated across database partitions • Blank = No database partitioning
LOG_ATTRIBUTE	CHAR (1)		• Always 0. This column is no longer used.
PCTFREE	SMALLINT		Percentage of each page to be reserved for future inserts.

Table 265. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
APPEND_MODE	CHAR (1)		For row-organized tables, controls how rows are inserted into pages. • N = New rows are inserted into existing spaces, if available • Y = New rows are appended to the end of the data For column-organized tables, which are implicitly always in append mode, this field always has a value of 'N'.
REFRESH	CHAR (1)		Refresh mode. • D = Deferred • I = Immediate • O = Once • Blank = Not a materialized query table
REFRESH_TIME	TIMESTAMP	Y	For REFRESH = 'D' or 'O', time at which the data was last refreshed (REFRESH TABLE statement); null value otherwise.
LOCKSIZE	CHAR (1)		Indicates the preferred lock granularity for tables that are accessed by data manipulation language (DML) statements. Applies to tables only. Possible values are: • I = Block insert • R = Row • T = Table • Blank = Not applicable
VOLATILE	CHAR (1)		• C = Cardinality of the table is volatile • Blank = Not applicable
ROW_FORMAT	CHAR (1)		Not used.

Table 265. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
PROPERTY	VARCHAR (32)		Properties for a table. A single blank indicates that the table has no properties. The following is position within string, value, and meaning: • 1, Y = User maintained materialized query table • 2, Y = Staging table • 3, Y = Propagate immediate • 11, Y = Nickname that will not be cached • 13, Y = Statistical view • 19, Y = Statistical view for an index with an expression-based key • 20, Y = Column-organized table • 21, Y = Synopsis table • 23, Y = Shadow table (materialized query table maintained by replication) • 25, Y = Random distribution table • 27, Y = External table
STATISTICS_PROFILE	CLOB (10M)	Y	RUNSTATS command used to register a statistical profile for the object.
COMPRESSION	CHAR (1)		• B = Both value and row compression are enabled • N = No compression is enabled; a row format that does not support compression is used • R = Row compression is enabled; a row format that supports compression might be used • V = Value compression is enabled; a row format that supports compression is used • Blank = Not applicable
ROWCOMPMODE	CHAR (1)		Row compression mode for the table. • A = ADAPTIVE • S = STATIC • Blank = Row compression is not enabled

Table 265. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
ACCESS_MODE	CHAR (1)		Access restriction state of the object. These states apply to objects that are in set integrity pending state or to objects that were processed by a SET INTEGRITY statement only. Possible values are: • D = No data movement • F = Full access • N = No access • R = Read-only access
CLUSTERED	CHAR (1)	Y	• T = Table is clustered by insert time • Y = Table is clustered by dimensions (even if only by one dimension) • Null value = Table is not clustered by dimensions or insert time
ACTIVE_BLOCKS	BIGINT		Total number of active blocks in the table, or -1. Applies to multidimensional clustering (MDC) tables or insert time clustering (ITC) tables only.
DROPRULE	CHAR (1)		• N = No rule • R = Restrict rule applies on drop
MAXFREESPACESEARCH	SMALLINT		Reserved for future use.
AVGCOMPRESSEDROWSIZE	SMALLINT		Average length (in bytes) of compressed rows in this table; -1 if statistics are not collected.
AVGROWCOMPRESSIONRATIO	REAL		For compressed rows in the table, this is the average compression ratio by row, that is, the average uncompressed row length divided by the average compressed row length; -1 if statistics are not collected.
AVGROWSIZE	SMALLINT		Average length (in bytes) of both compressed and uncompressed rows in this table; -1 if statistics are not collected.
PCTROWSCOMPRESSED	REAL		Compressed rows as a percentage of the total number of rows in the table; -1 if statistics are not collected.

Table 265. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
LOGINDEXBUILD	VARCHAR (3)	Y	Level of logging that is to be performed during create, re-create, or reorganize index operations on the table. • OFF = Index build operations on the table is logged minimally • ON = Index build operations on the table is logged completely • Null value = Value of the *logindexbuild* database configuration parameter is used to determine whether index build operations are to be logged completely
CODEPAGE	SMALLINT		Code page of the object. This is the default code page that is used for all character columns, triggers, check constraints, and expression-generated columns.
COLLATIONSCHEMA	VARCHAR (128)		Schema name of the collation for the table.
COLLATIONNAME	VARCHAR (128)		Unqualified name of the collation for the table.
COLLATIONSCHEMA_ORDERBY	VARCHAR (128)		Schema name of the collation for ORDER BY clauses in the table.
COLLATIONNAME_ORDERBY	VARCHAR (128)		Unqualified name of the collation for ORDER BY clauses in the table.
ENCODING_SCHEME	CHAR (1)		• A = CCSID ASCII was specified • U = CCSID UNICODE was specified • Blank = CCSID clause was not specified
PCTPAGESSAVED	SMALLINT	N	The approximate percentage of pages that are saved in a row-organized table as a result of row compression. For a column-organized table, the estimate is based on the number of data pages that are needed to store the table in uncompressed row organization.-1 if statistics are not collected.
LAST_REGEN_TIME	TIMESTAMP	Y	Time at which any views or check constraints on the table were last regenerated.
SECPOLICYID	INTEGER		Identifier for the security policy that protects the table; 0 for non-protected tables.

Table 265. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
PROTECTIONGRANULARITY	CHAR (1)		• B = Both column- and row-level granularity • C = Column-level granularity • R = Row-level granularity • Blank = Non-protected table
AUDITPOLICYID	INTEGER	Y	Identifier for the audit policy.
AUDITPOLICYNAME	VARCHAR (128)	Y	Name of the audit policy.
AUDITEXCEPTIONENABLED	CHAR (1)		Reserved for future use.
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the table, view, alias, or nickname.
ONCOMMIT	CHAR (1)		Specifies the action that is taken on the created temporary table when a COMMIT operation is performed. • D = Delete rows • P = Preserve rows • Blank = Table is not a created temporary table
LOGGED	CHAR (1)		Specifies whether the created temporary table is logged. • N = Not logged • Y = Logged • Blank = Table is not a created temporary table
ONROLLBACK	CHAR (1)		Specifies the action that is taken on the created temporary table when a ROLLBACK operation is performed. • D = Delete rows • P = Preserve rows • Blank = Table is not a created temporary table
LASTUSED	DATE		Date when the table was last used by any DML statement or the LOAD command. This column is not updated for an alias, created temporary table, nickname, or view. This column is not updated when the table is used on an HADR standby database. The default value is '0001-01-01'. This value is updated asynchronously not more than once within a 24 hour period and might not reflect usage within the last 15 minutes.

Table 265. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
CONTROL	CHAR (1)		Access control that is enforced for the table • B = Both row and column • C = Column • R = Row • Blank = No access control
TEMPORALTYPE	CHAR (1)		Type of temporal table. • A = Application-period temporal table • B = Bitemporal table • N = Not a temporal table • S = System-period temporal table
TABLEORG	CHAR(1)		• C = Column-organized table • R = Row-organized table • N = Not a table
EXTENDED_ROW_SIZE	CHAR(1)		Indicates whether the row size of a table that is organized by row exceeds the maximum record length for the page size of the table space in which it is defined. • N = Row size does not exceed the maximum record length for the page size • Y = Row size exceeds the maximum record length for the page size • blank = Not applicable
PCTEXTENDEDROWS	REAL		Extended rows as a percentage of the total number of rows in the table; -1 if statistics are not collected.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.TABLESPACES

Each row represents a table space.

Table 266. SYSCAT.TABLESPACES Catalog View

Column Name	Data Type	Nullable	Description
TBSPACE	VARCHAR (128)		Name of the table space.
OWNER	VARCHAR (128)		Authorization ID of the owner of the table space.

Table 266. SYSCAT.TABLESPACES Catalog View (continued)

Column Name	Data Type	Nullable	Description
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
CREATE_TIME	TIMESTAMP		Time at which the table space was created.
TBSPACEID	INTEGER		Identifier for the table space.
TBSPACETYPE	CHAR (1)		Type of table space. • D = Database-managed space • S = System-managed space
DATATYPE	CHAR (1)		Type of data that can be stored in this table space. • A = All types of permanent data; regular table space • L = All types of permanent data; large table space • T = System temporary tables only • U = Created temporary tables or declared temporary tables only
EXTENTSIZE	INTEGER		Size of each extent, in pages of size PAGESIZE. This many pages are written to one container in the table space before switching to the next container.
PREFETCHSIZE	INTEGER		Number of pages of size PAGESIZE to be read when prefetching is performed; -1 when AUTOMATIC.
OVERHEAD	DOUBLE		Controller overhead and disk seek and latency time, in milliseconds (average for the containers in this table space); -1 when value is inherited from the storage group that the table space uses.
TRANSFERRATE	DOUBLE		Time to read one page of size PAGESIZE into the buffer (average for the containers in this table space); -1 when value is inherited from the storage group that the table space uses.
WRITEOVERHEAD	DOUBLE	Y	Reserved for future use.
WRITETRANSFERRATE	DOUBLE	Y	Reserved for future use.
PAGESIZE	INTEGER		Size (in bytes) of pages in this table space.
DBPGNAME	VARCHAR (128)		Name of the database partition group that is associated with this table space.
BUFFERPOOLID	INTEGER		Identifier for the buffer pool that is used by this table space (1 indicates the default buffer pool).

Table 266. SYSCAT.TABLESPACES Catalog View (continued)

Column Name	Data Type	Nullable	Description
DROP_RECOVERY	CHAR (1)		Indicates whether or not tables in this table space can be recovered after a drop table operation. • N = Tables are not recoverable • Y = Tables are recoverable
NGNAME[1]	VARCHAR (128)		Name of the database partition group that is associated with this table space.
DEFINER[2]	VARCHAR (128)		Authorization ID of the owner of the table space.
DATATAG	SMALLINT		A tag to identify data stored in this table space. Valid user-specified range is 1 through 9; 0 indicates no data tag specified; -1 indicates value is inherited from the storage group that the table space uses.
SGNAME	VARCHAR (128)	Y	Name of the storage group the table space is using; null value when the table space is not using automatic storage.
SGID	INTEGER		Identifier of the storage group the table space is using; -1 when the table space is not using automatic storage.
EFFECTIVEPREFETCHSIZE	INTEGER		Value of the effective prefetch size when PREFETCHSIZE is set to -1 (AUTOMATIC); otherwise same as PREFETCHSIZE.
CACHINGTIER	SMALLINT		Reserved for future use.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The NGNAME column is included for backwards compatibility. See DBPGNAME.
2. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.TABOPTIONS

Each row represents an option that is associated with a remote table.

Table 267. SYSCAT.TABOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of a table, view, alias, or nickname.
TABNAME	VARCHAR (128)		Unqualified name of a table, view, alias, or nickname.
OPTION	VARCHAR (128)		Name of the table option.
SETTING	CLOB (32K)		Value of the table option.

SYSCAT.TBSPACEAUTH

Each row represents a user, group, or role that has been granted the USE privilege on a particular table space in the database.

Table 268. SYSCAT.TBSPACEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
TBSPACE	VARCHAR (128)		Name of the table space.
USEAUTH	CHAR (1)		Privilege to create tables within the table space. • G = Held and grantable • N = Not held • Y = Held

SYSCAT.THRESHOLDS

Each row represents a threshold.

Table 269. SYSCAT.THRESHOLDS Catalog View

Column Name	Data Type	Nullable	Description
THRESHOLDNAME	VARCHAR (128)		Name of the threshold.
THRESHOLDID	INTEGER		Identifier for the threshold.
ORIGIN	CHAR (1)		Origin of the threshold. • U = Threshold was created by a user • W = Threshold was created through a work action set
THRESHOLDCLASS	CHAR (1)		Classification of the threshold. • A = Aggregate threshold • C = Activity threshold

Table 269. SYSCAT.THRESHOLDS Catalog View (continued)

Column Name	Data Type	Nullable	Description
THRESHOLDPREDICATE	VARCHAR (15)		Type of the threshold. Possible values are: • AGGTEMPSPACE • CONCDBC • CONCWCN • CONCWOC • CONNIDLETIME • CPUTIME • CPUTIMEINSC • DATATAGINSC • DATATAGNOTINSC • DBCONN • ESTSQLCOST • ROWSREAD • ROWSREADINSC • ROWSRET • RUNTIME • RUNTIMEINALLSC • SCCONN • SORTSHRHEAPUTIL  **Attention:** This feature is available in Db2 Version 11.5 Mod Pack 2 and later versions. • TEMPSPACE • TOTALTIME • UOWTOTALTIME
THRESHOLDPREDICATEID	SMALLINT		Identifier for the threshold predicate.
DOMAIN	CHAR (2)		Domain of the threshold. • DB = Database • SB = Service subclass • SP = Service superclass • WA = Work action set • WD = Workload definition • SQ = SQL statement
DOMAINID	INTEGER		Identifier for the object with which the threshold is associated. This can be a service class, work action, workload unique ID, or SQL statement. If this is a database threshold, this value is 0.

Table 269. SYSCAT.THRESHOLDS Catalog View (continued)

Column Name	Data Type	Nullable	Description
ENFORCEMENT	CHAR (1)		Scope of enforcement for the threshold. • D = Database • P = Member • W = Workload occurrence
QUEUING	CHAR (1)		• N = The threshold is not queueing • Y = The threshold is queueing
MAXVALUE	BIGINT		Upper bound specified by the threshold. If THRESHOLDPREDICATE is 'DATATAGINSC' or 'DATATAGNOTINSC', this value encodes one or more data tags.
DATATAGLIST	VARCHAR (256)	Y	If THRESHOLDPREDICATE is 'DATATAGINSC' or 'DATATAGNOTINSC', this value represents one or more data tags as a comma separated list. Otherwise, the null value.
QUEUESIZE	INTEGER		If QUEUEING is 'Y', the size of the queue. -1 otherwise.
OVERFLOWPERCENT	SMALLINT		Reserved for future use.
COLLECTACTDATA	CHAR (1)		Specifies what activity data should be collected by the applicable event monitor. • D = Activity data with details • N = None • S = Activity data with details and section environment • V = Activity data with details and values • W = Activity data without details • X = Activity data with details, section environment, and values
COLLECTACTPARTITION	CHAR (1)		Specifies where activity data is collected. • C = Coordinator member of the activity • D = All members
EXECUTION	CHAR (1)		Indicates the execution action taken after a threshold has been exceeded. • C = Execution continues • F = Application is forced off the system • R = Execution is remapped to a different service subclass • S = Execution stops
REMAPSCID	SMALLINT		Target service subclass ID of the REMAP ACTIVITY action.

Table 269. SYSCAT.THRESHOLDS Catalog View (continued)

Column Name	Data Type	Nullable	Description
VIOLATIONRECORDLOGGED	CHAR (1)		Indicates whether a record is written to the event monitor upon threshold violation. • N = No • Y = Yes
CHECKINTERVAL	INTEGER		The interval, in seconds, in which the threshold condition is checked if THRESHOLDPREDICATE is: • 'CPUTIME' • 'CPUTIMEINSC' • 'ROWSREAD' • 'ROWSREADINSC' Otherwise, -1.
ENABLED	CHAR (1)		• N = This threshold is disabled. • Y = This threshold is enabled.
CREATE_TIME	TIMESTAMP		Time at which the threshold was created.
ALTER_TIME	TIMESTAMP		Time at which the threshold was last altered.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.TRANSFORMS

Each row represents the functions that handle transformations between a user-defined type and a base SQL type, or the reverse.

Table 270. SYSCAT.TRANSFORMS Catalog View

Column Name	Data Type	Nullable	Description
TYPEID	SMALLINT		Identifier for the data type.
TYPESCHEMA	VARCHAR (128)		Schema name of the data type if TYPEMODULEID is null; otherwise schema name of the module to which the data type belongs.
TYPENAME	VARCHAR (128)		Unqualified name of the data type.
GROUPNAME	VARCHAR (128)		Name of the transform group.
FUNCID	INTEGER		Identifier for the routine.
FUNCSCHEMA	VARCHAR (128)		Schema name of the routine if ROUTINEMODULEID is null; otherwise schema name of the module to which the routine belongs.
FUNCNAME	VARCHAR (128)		Unqualified name of the routine.
SPECIFICNAME	VARCHAR (128)		Name of the routine instance (might be system-generated).

Table 270. SYSCAT.TRANSFORMS Catalog View (continued)

Column Name	Data Type	Nullable	Description
TRANSFORMTYPE	VARCHAR (8)		• 'FROM SQL' = Transform function transforms a structured type from SQL • 'TO SQL' = Transform function transforms a structured type to SQL
FORMAT	CHAR (1)		Format produced by the FROM SQL transform. • S = Structured data type • U = User-defined
MAXLENGTH	INTEGER	Y	Maximum length (in bytes) of output from the FROM SQL transform; the null value for TO SQL transforms.
ORIGIN	CHAR (1)		Source of this group of transforms. • O = Original transform group • R = Redefined transform group
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.TRIGDEP

Each row represents a dependency of a trigger on some other object. The trigger depends on the object of type BTYPE of name BNAME, so a change to the object affects the trigger.

Table 271. SYSCAT.TRIGDEP Catalog View

Column Name	Data Type	Nullable	Description
TRIGSCHEMA	VARCHAR (128)		Schema name of the trigger.
TRIGNAME	VARCHAR (128)		Unqualified name of the trigger.

Table 271. SYSCAT.TRIGDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: • A = Table alias • B = Trigger • C = Column • F = Routine • G = Global temporary table • H = Hierachy table • I = Index • K = Package • L = Detached table • N = Nickname • O = Privilege dependency on all subtables or subviews in a table or view hierarchy • Q = Sequence • R = User-defined data type • S = Materialized query table • T = Table (not typed) • U = Typed table • V = View (not typed) • W = Typed view • X = Index extension • Z = XSR object • m = Module • q = Sequence alias • u = Module alias • v = Global variable • * = Anchored to the row of a base table
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which there is a dependency.
TABAUTH	SMALLINT	Y	If BTYPE = 'O', 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by a dependent trigger; null value otherwise.

SYSCAT.TRIGGERS

Each row represents a trigger. For table hierarchies, each trigger is recorded only at the level of the hierarchy where the trigger was created.

Table 272. SYSCAT.TRIGGERS Catalog View

Column Name	Data Type	Nullable	Description
TRIGSCHEMA	VARCHAR (128)		Schema name of the trigger.
TRIGNAME	VARCHAR (128)		Unqualified name of the trigger.
OWNER	VARCHAR (128)		Authorization ID of the owner of the trigger.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
TABSCHEMA	VARCHAR (128)		Schema name of the table or view to which this trigger applies.
TABNAME	VARCHAR (128)		Unqualified name of the table or view to which this trigger applies.
TRIGTIME	CHAR (1)		Time at which triggered actions are applied to the base table, relative to the event that fired the trigger. • A = Trigger is applied after the event • B = Trigger is applied before the event • I = Trigger is applied instead of the event
TRIGEVENT	CHAR (1)		Event that fires the trigger. • D = Delete event • I = Insert event • M = Multiple events • U = Update event
EVENTUPDATE	CHAR (1)		Indicates whether an update event fires the trigger. • N = No • Y = Yes
EVENTDELETE	CHAR (1)		Indicates whether a delete event fires the trigger. • N = No • Y = Yes
EVENTINSERT	CHAR (1)		Indicates whether an insert event fires the trigger. • N = No • Y = Yes

Table 272. SYSCAT.TRIGGERS Catalog View (continued)

Column Name	Data Type	Nullable	Description
GRANULARITY	CHAR (1)		Trigger is executed once per: • R = Row • S = Statement
VALID	CHAR (1)		• N = Trigger is invalid • X = Trigger is inoperative and must be re-created • Y = Trigger is valid
CREATE_TIME	TIMESTAMP		Time at which the trigger was defined. Used in resolving functions and types.
QUALIFIER	VARCHAR (128)		Value of the default schema at the time of object definition. Used to complete any unqualified references.
FUNC_PATH	CLOB (2K)		SQL path in effect when the trigger was defined.
TEXT	CLOB (2M)		Full text of the CREATE TRIGGER statement, exactly as typed.
LAST_REGEN_TIME	TIMESTAMP		Time at which the packed descriptor for the trigger was last regenerated.
COLLATIONSCHEMA	VARCHAR (128)		Schema name of the collation for the trigger.
COLLATIONNAME	VARCHAR (128)		Unqualified name of the collation for the trigger.
COLLATIONSCHEMA_ORDERBY	VARCHAR (128)		Schema name of the collation for ORDER BY clauses in the trigger.
COLLATIONNAME_ORDERBY	VARCHAR (128)		Unqualified name of the collation for ORDER BY clauses in the trigger.
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the trigger.
SECURE	CHAR (1)		Indicates whether the trigger is secure for row and column access control • N = Not secure • Y = Secure
ALTER_TIME	TIMESTAMP		Time at which the trigger was last altered.

Table 272. SYSCAT.TRIGGERS Catalog View (continued)

Column Name	Data Type	Nullable	Description
DEBUG_MODE	VARCHAR (8)		Indicates whether the trigger can be debugged using the Db2 debugger. • DISALLOW = Trigger is not debuggable • ALLOW = Trigger is debuggable, and can participate in a client debug session with the Db2 debugger • DISABLE = Trigger is not debuggable, and this setting cannot be altered without dropping and re-creating the trigger • Blank = Trigger type is not currently supported by the Db2 debugger
ENABLED	CHAR (1)		Reserved for future use.
LIB_ID	INTEGER	Y	Internal identifier for compiled SQL triggers. Otherwise, the null value.
PRECOMPILE_OPTIONS	VARCHAR(1024)	Y	The precompile and bind options that were in effect when the compiled trigger was created. The null value if the trigger is not compiled.
COMPILE_OPTIONS	VARCHAR(1024)	Y	The value of the SQL_CCFLAGS special register that was in effect when the compiled trigger was created and inquiry directives were present. An empty string if no inquiry directives were present in the compiled trigger. The null value if the trigger is not compiled.
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for compatibility with earlier versions. See OWNER.

SYSCAT.TYPEMAPPINGS

Each row represents a data type mapping between a locally-defined data type and a data source data type. There are two mapping types (mapping directions):*forward type mapping*s map a data source data type to a locally-defined data type; *reverse type mapping*s map a locally-defined data type to a data source data type.

Table 273. SYSCAT.TYPEMAPPINGS Catalog View

Column Name	Data Type	Nullable	Description
TYPE_MAPPING	VARCHAR (18)		Name of the type mapping (might be system-generated).

Table 273. SYSCAT.TYPEMAPPINGS Catalog View (continued)

Column Name	Data Type	Nullable	Description
MAPPINGDIRECTION	CHAR (1)		Indicates whether this type mapping is a forward or a reverse type mapping. • F = Forward type mapping • R = Reverse type mapping
TYPESCHEMA	VARCHAR (128)	Y	Schema name of the local type in a data type mapping; the null value for built-in types.
TYPENAME	VARCHAR (128)		Unqualified name of the local type in a data type mapping.
TYPEID	SMALLINT		Identifier for the data type.
SOURCETYPEID	SMALLINT		Identifier for the source type.
OWNER	VARCHAR (128)		Authorization ID of the owner of the type mapping. 'SYSIBM' indicates a built-in type mapping.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
LENGTH	INTEGER	Y	Maximum length or precision of the local data type in this mapping. If the null value, the system determines the maximum length or precision. For character types, represents the maximum number of bytes.
SCALE	SMALLINT	Y	Maximum number of digits in the fractional part of a local decimal value or the maximum number of digits of fractional seconds of a local TIMESTAMP value in this mapping. If the null value, the system determines the maximum number.
LOWER_LEN	INTEGER	Y	Minimum length or precision of the local data type in this mapping. If the null value, the system determines the minimum length or precision. For character types, represents the minimum number of bytes.
UPPER_LEN	INTEGER	Y	Maximum length or precision of the local data type in this mapping. If the null value, the system determines the maximum length or precision. For character types, represents the maximum number of bytes.
LOWER_SCALE	SMALLINT	Y	Minimum number of digits in the fractional part of a local decimal value or the minimum number of digits of fractional seconds of a local TIMESTAMP value in this mapping. If the null value, the system determines the minimum number.

Table 273. SYSCAT.TYPEMAPPINGS Catalog View (continued)

Column Name	Data Type	Nullable	Description
UPPER_SCALE	SMALLINT	Y	Maximum number of digits in the fractional part of a local decimal value or the maximum number of digits of fractional seconds of a local TIMESTAMP value in this mapping. If the null value, the system determines the maximum number.
S_OPR_P	CHAR (2)	Y	Relationship between the scale and precision of a local decimal value in this mapping. Basic comparison operators (=, <, >, <=, >=, <>) can be used. A null value indicates that no specific relationship is required.
BIT_DATA	CHAR (1)	Y	Indicates whether or not this character type is for bit data. Possible values are: • N = This type is not for bit data • Y = This type is for bit data • Null value = This is not a character data type, or the system determines the bit data attribute
WRAPNAME	VARCHAR (128)	Y	Data access protocol (wrapper) to which this mapping applies.
SERVERNAME	VARCHAR (128)	Y	Uppercase name of the server.
SERVERTYPE	VARCHAR (30)	Y	Type of server.
SERVERVERSION	VARCHAR (18)	Y	Server version.
REMOTE_TYPESCHEMA	VARCHAR (128)	Y	Schema name of the data source data type.
REMOTE_TYPENAME	VARCHAR (128)		Unqualified name of the data source data type.
REMOTE_META_TYPE	CHAR (1)	Y	Indicates whether this remote type is a system built-in type or a distinct type. • S = System built-in type • T = Distinct type
REMOTE_LOWER_LEN	INTEGER	Y	Minimum length or precision of the remote data type in this mapping, or the null value. For character types, represents the minimum number of characters (not bytes). For binary types, represents the minimum number of bytes. A value of -1 indicates that the default length or precision is used, or that the remote type does not have a length or precision.

Table 273. SYSCAT.TYPEMAPPINGS Catalog View (continued)

Column Name	Data Type	Nullable	Description
REMOTE_UPPER_LEN	INTEGER	Y	Maximum length or precision of the remote data type in this mapping, or the null value. For character types, represents the maximum number of characters (not bytes). For binary types, represents the maximum number of bytes. A value of -1 indicates that the default length or precision is used, or that the remote type does not have a length or precision.
REMOTE_LOWER_SCALE	SMALLINT	Y	Minimum number of digits in the fractional part of a remote decimal value or the minimum number of digits of fractional seconds of a remote TIMESTAMP value in this mapping, or the null value.
REMOTE_UPPER_SCALE	SMALLINT	Y	Maximum number of digits in the fractional part of a remote decimal value or the maximum number of digits of fractional seconds of a remote TIMESTAMP value in this mapping, or the null value.
REMOTE_S_OPR_P	CHAR (2)	Y	Relationship between the scale and precision of a remote decimal value in this mapping. Basic comparison operators (=, <, >, <=, >=, <>) can be used. A null value indicates that no specific relationship is required.
REMOTE_BIT_DATA	CHAR (1)	Y	Indicates whether or not this remote character type is for bit data. Possible values are: • N = This type is not for bit data • Y = This type is for bit data • Null value = This is not a character data type, or the system determines the bit data attribute
USER_DEFINED	CHAR (1)		Indicates whether or not the mapping is user-defined. The value is always 'Y'; that is, the mapping is always user-defined.
CREATE_TIME	TIMESTAMP		Time at which this mapping was created.
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the type mapping. 'SYSIBM' indicates a built-in type mapping.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.USAGELISTS

Each row represents a usage list for a table or index object.

Table 274. SYSCAT.USAGELISTS Catalog View

Column Name	Data Type	Nullable	Description
USAGELISTSCHEMA	VARCHAR (128)		Schema of the usage list.
USAGELISTNAME	VARCHAR (128)		Name of the usage list.
USAGELISTID	INTEGER		Identifier for the usage list.
OBJECTSCHEMA	VARCHAR (128)		Schema name of the object for which the usage list is defined.
OBJECTNAME	VARCHAR (128)		Unqualified name of the object for which the usage list is defined.
OBJECTTYPE	CHAR (1)		The type of the object for which this usage list is defined. • I = Index • T = Table
STATUS	CHAR (1)		The status of the usage list. • I = Invalid • V = Valid
MAXLISTSIZE	INTEGER		The maximum number of entries in the usage list.
WHENFULL	CHAR (1)		Action to be performed when the usage list is full. • D = Deactivate collection • W = Wrap
AUTOSTART	CHAR (1)		Indicates whether this usage list is to be activated automatically when the database starts. • N = Manual start • Y = Autostart
ACTIVEDURATION	INTEGER		Reserved for future use.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.USEROPTIONS

Each row represents a server-specific user option value.

Table 275. SYSCAT.USEROPTIONS Catalog View

Column Name	Data Type	Nullable	Description
AUTHID	VARCHAR (128)		Local authorization ID, in uppercase characters.
AUTHIDTYPE	CHAR (1)		• U = Grantee is an individual user

Table 275. SYSCAT.USEROPTIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
SERVERNAME	VARCHAR (128)		Name of the server on which the user is defined.
OPTION	VARCHAR (128)		Name of the user option.
SETTING	VARCHAR (2048)		Value of the user option.

SYSCAT.VARIABLEAUTH

Each row represents a user, group, or role that has been granted one or more privileges by a specific grantor on a global variable in the database that is not defined in a module.

Table 276. SYSCAT.VARIABLEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
VARSCHEMA	VARCHAR (128)		Schema name of the global variable if VARMODULEID is null; otherwise schema name of the module to which the global variable belongs.
VARNAME	VARCHAR (128)		Unqualified name of the global variable.
VARID	INTEGER		Identifier for the global variable.
READAUTH	CHAR (1)		Privilege to read the global variable. • G = Held and grantable • N = Not held • Y = Held
WRITEAUTH	CHAR (1)		Privilege to write the global variable. • G = Held and grantable • N = Not held • Y = Held

SYSCAT.VARIABLEDEP

Each row represents a dependency of a global variable on some other object. The global variable depends on the object of type BTYPE of name BNAME, so a change to the object affects the global variable.

Table 277. SYSCAT.VARIABLEDEP Catalog View

Column Name	Data Type	Nullable	Description
VARSCHEMA	VARCHAR (128)		Schema name of the global variable that has dependencies on another object.
VARMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the global variable belongs. The null value if not a module variable.
VARNAME	VARCHAR (128)		Unqualified name of the global variable that has dependencies on another object.
VARMODULEID	INTEGER	Y	Identifier for the module of the object that has dependencies on another object.
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: • A = Table alias • F = Routine • G = Global temporary table • H = Hierarchy table • I = Index • N = Nickname • O = Privilege dependency on all subtables or subviews in a table or view hierarchy • R = User-defined data type • S = Materialized query table • T = Table (not typed) • U = Typed table • V = View (not typed) • W = Typed view • m = Module • q = Sequence alias • u = Module alias • v = Global variable • * = Anchored to the row of a base table
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.

Table 277. SYSCAT.VARIABLEDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
BMODULEID	INTEGER	Y	Identifier for the module of the object on which there is a dependency.
TABAUTH	SMALLINT	Y	If BTYPE = 'O', 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by the dependent global variable; the null value otherwise.

SYSCAT.VARIABLES

Each row represents a global variable.

Table 278. SYSCAT.VARIABLES Catalog View

Column Name	Data Type	Nullable	Description
VARSCHEMA	VARCHAR (128)		Schema name of the global variable if VARMODULEID is null; otherwise schema name of the module to which the global variable belongs.
VARMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the global variable belongs. The null value if not a module variable.
VARNAME	VARCHAR (128)		Unqualified name of the global variable.
VARMODULEID	INTEGER	Y	Identifier for the module to which the global variable belongs. The null value if not a module variable.
VARID	INTEGER		Identifier for the global variable.
OWNER	VARCHAR (128)		Authorization ID of the owner of the global variable.
OWNERTYPE	CHAR (1)		• U = The owner is an individual user
CREATE_TIME	TIMESTAMP		Time at which the global variable was created.
LAST_REGEN_TIME	TIMESTAMP		Time at which the default expression was last regenerated.
VALID	CHAR (1)		• N = The global variable is invalid • Y = The global variable is valid
PUBLISHED	CHAR (1)		Indicates whether the module variable can be referenced outside its module. • N = The module variable is not published • Y = The module variable is published • Blank = Not applicable

Table 278. SYSCAT.VARIABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
TYPESCHEMA	VARCHAR (128)		Schema name of the data type if TYPEMODULEID is null; otherwise schema name of the module to which the data type belongs.
TYPEMODULENAME	VARCHAR (128)		Unqualified name of the module to which the variable data type belongs. The null value if the variable data type does not belong to a module.
TYPENAME	VARCHAR (128)		Unqualified name of the data type.
TYPEMODULEID	INTEGER	Y	Identifier for the module to which the variable data type belongs. The null value if the variable data type does not belong to a module.
LENGTH	INTEGER		Maximum length of the global variable.
SCALE	SMALLINT		Scale if the global variable data type is DECIMAL or distinct type based on DECIMAL; the number of digits of fractional seconds if the global variable data type is TIMESTAMP or distinct type based on TIMESTAMP; 0 otherwise.
TYPESTRINGUNITS	VARCHAR (11)	Y	In a Unicode database, the string units that apply to a character string or graphic string data type. Otherwise, the null value.
STRINGUNITSLENGTH	INTEGER	Y	In a Unicode database, the declared number of string units for a character string or graphic string data type. Otherwise, the null value.
CODEPAGE	SMALLINT		Code page of the global variable.
COLLATIONSCHEMA	VARCHAR (128)		Schema name of the collation for the variable.
COLLATIONNAME	VARCHAR (128)		Unqualified name of the collation for the variable.
COLLATIONSCHEMA_ORDERBY	VARCHAR (128)		Schema name of the collation for ORDER BY clauses in the variable.
COLLATIONNAME_ORDERBY	VARCHAR (128)		Unqualified name of the collation for ORDER BY clauses in the variable.
SCOPE	CHAR (1)		Scope of the global variable. • D = Database • S = Session
DEFAULT	CLOB (64K)	Y	Expression used to calculate the initial value of the global variable when first referenced.

Table 278. SYSCAT.VARIABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
QUALIFIER	VARCHAR (128)	Y	Value of the default schema at the time the variable was defined.
FUNC_PATH	CLOB (2K)	Y	SQL path in effect when the variable was defined.
NULLS	CHAR (1)		Reserved for future use.
READONLY	CHAR (1)		• C = Read-only because the global variable is defined with a CONSTANT clause • N = Not read-only • S = Read-only because the global variable value is maintained by the database manager
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.VIEWS

Each row represents a view or materialized query table.

Table 279. SYSCAT.VIEWS Catalog View

Column Name	Data Type	Nullable	Description
VIEWSCHEMA	VARCHAR (128)		Schema name of the view or materialized query table.
VIEWNAME	VARCHAR (128)		Unqualified name of the view or materialized query table.
OWNER	VARCHAR (128)		Authorization ID of the owner of the view or materialized query table.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
SEQNO	SMALLINT		Always 1.
VIEWCHECK	CHAR (1)		Type of view checking. • C = Cascaded check option • L = Local check option • N = No check option or is a materialized query table

Table 279. SYSCAT.VIEWS Catalog View (continued)

Column Name	Data Type	Nullable	Description
READONLY	CHAR (1)		• N = View can be updated by users with appropriate authorization or is a materialized query table • Y = View is read-only because of its definition
VALID	CHAR (1)		• N = View or materialized query table definition is invalid • X = View or materialized query table definition is inoperative and must be recreated • Y = View or materialized query table definition is valid
QUALIFIER	VARCHAR (128)		Value of the default schema at the time of object definition. Used to complete any unqualified references.
FUNC_PATH	CLOB (2K)		SQL path in effect when the view or materialized query table was defined.
TEXT	CLOB (2M)		Full text of the view or materialized query table CREATE statement, exactly as typed.
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the view or materialized query table.
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.WORKACTIONS

Each row represents a work action that is defined for a work action set.

Table 280. SYSCAT.WORKACTIONS Catalog View

Column Name	Data Type	Nullable	Description
ACTIONNAME	VARCHAR (128)		Name of the work action.
ACTIONID	INTEGER		Identifier for the work action.
ACTIONSETNAME	VARCHAR (128)	Y	Name of the work action set.
ACTIONSETID	INTEGER		Identifier of the work action set to which this work action belongs. This column refers to the ACTIONSETID column in the SYSCAT.WORKACTIONSETS view.
WORKCLASSNAME	VARCHAR (128)	Y	Name of the work class.

Table 280. SYSCAT.WORKACTIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
WORKCLASSID	INTEGER		Identifier of the work class. This column refers to the WORKCLASSID column in the SYSCAT.WORKCLASSES view.
CREATE_TIME	TIMESTAMP		Time at which the work action was created.
ALTER_TIME	TIMESTAMP		Time at which the work action was last altered.
ENABLED	CHAR (1)		• N = This work action is disabled. • Y = This work action is enabled.

Table 280. SYSCAT.WORKACTIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
ACTIONTYPE	CHAR (1)		The type of action performed on each Db2 activity that matches the attributes in the work class within scope. • B = Collect basic aggregate activity data, specifiable only for work action sets that apply to service classes or workloads. • C = Allow any Db2 activity under the associated work class to execute and increment the work class counter. • D = Collect activity data with details at the coordinating member of the activity. • E = Collect extended aggregate activity data, specifiable only for work action sets that apply to service classes or workloads. • F = Collect activity data with details, section, and values at the coordinating member of the activity. • G = Collect activity details and section at the coordinating member of the activity and collect activity data at all members. • H = Collect activity details, section, and values at the coordinating member of the activity and collect activity data at all members. • M = Map to a service subclass, specifiable only for work action sets that apply to service classes. • P = Prevent the execution of any Db2 activity under the work class with which this work action is associated. • S = Collect activity data with details and section at the coordinating member of the activity. • T = The action represents a threshold, specifiable only for work action sets that are associated with a database or a workload. • U = Map all activities with a nesting level of zero and all activities nested under these activities to a service subclass, specifiable only for work action sets that apply to service classes. • V = Collect activity data with details and values at the coordinating member. • W = Collect activity data without details at the coordinating member. • X = Collect activity data with details at the coordinating member and collect activity data at all members. • Y = Collect activity data with details and values at the coordinating member and collect activity data at all members. • Z = Collect activity data without details at all members.

Table 280. SYSCAT.WORKACTIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
REFOBJECTID	INTEGER	Y	If ACTIONTYPE is 'M' (map) or 'N' (map nested), this value is set to the ID of the service subclass to which the Db2 activity is mapped. If ACTIONTYPE is 'T' (threshold), this value is set to the ID of the threshold to be used. For all other actions, this value is NULL.
REFOBJECTTYPE	VARCHAR (30)		If the ACTIONTYPE is 'M' or 'N', this value is set to 'SERVICE CLASS'; if the ACTIONTYPE is 'T', this value is 'THRESHOLD'; the null value otherwise.
SECTIONACTUALSOPTIONS	VARCHAR (32)		Specifies what section actuals are collected during the execution of a section. The first position in the string represents whether the collection of section actuals is enabled. • B = Enabled and collect basic operator cardinality counts and statistics for each object referenced by the section (DML statements only). • N = Not enabled. The second position is always 'N' and reserved for for future use.

SYSCAT.WORKACTIONSETS

Each row represents a work action set.

Table 281. SYSCAT.WORKACTIONSETS Catalog View

Column Name	Data Type	Nullable	Description
ACTIONSETNAME	VARCHAR (128)		Name of the work action set.
ACTIONSETID	INTEGER		Identifier for the work action set.
WORKCLASSSETNAME	VARCHAR (128)	Y	Name of the work class set.
WORKCLASSSETID	INTEGER		The identifier of the work class set that is to be mapped to the object specified by the OBJECTID. This column refers to WORKCLASSSETID in the SYSCAT.WORKCLASSSETS view.
CREATE_TIME	TIMESTAMP		Time at which the work action set was created.
ALTER_TIME	TIMESTAMP		Time at which the work action set was last altered.
ENABLED	CHAR (1)		• N = This work action set is disabled. • Y = This work action set is enabled.

Table 281. SYSCAT.WORKACTIONSETS Catalog View (continued)

Column Name	Data Type	Nullable	Description
OBJECTTYPE	CHAR (1)		• b = Service superclass • w = Workload • Blank = Database
OBJECTNAME	VARCHAR (128)	Y	Name of the service class or workload.
OBJECTID	INTEGER		The identifier of the object to which the work class set (specified by the WORKCLASSSETID) is mapped. If the OBJECTTYPE is 'b', the OBJECTID is the ID of the service superclass. If the OBJECTTYPE is 'w', the OBJECTID is the ID of the workload. If the OBJECTTYPE is blank, the OBJECTID is -1.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.WORKCLASSATTRIBUTES

Each row represents an attribute in the definition of a work class.

Table 282. SYSCAT.WORKCLASSATTRIBUTES Catalog View

Column Name	Data Type	Nullable	Description
WORKCLASSNAME	VARCHAR (128)		Name of the work class.
WORKCLASSSETNAME	VARCHAR (128)		Name of the work class set.
WORKCLASSID	INTEGER		Identifier for the work class.
WORKCLASSSETID	INTEGER		Identifier for the work class set.
TYPE	VARCHAR (30)		The type of work class attribute. Possible values are: • WORK TYPE • TIMERONCOST • CARDINALITY • DATA TAG • ROUTINE SCHEMA • RUNTIME

Table 282. SYSCAT.WORKCLASSATTRIBUTES Catalog View (continued)

Column Name	Data Type	Nullable	Description
VALUE1	DOUBLE	Y	The meaning of this value depends on the value in the TYPE column: **TYPE='WORK TYPE'** A number that indicates the type of database activity: • 1 = ALL • 2 = READ • 3 = WRITE • 4 = CALL • 5 = DML • 6 = DDL • 7 = LOAD **TYPE='TIMERONCOST'** The low value in the range. **TYPE='CARDINALITY'** The low value in the range. **TYPE='DATA TAG'** The tag that the estimated data tag list must contain. **TYPE='ROUTINE SCHEMA'** The null value.
VALUE2	DOUBLE	Y	The meaning of this value depends on the value in the TYPE column: **TYPE='WORK TYPE'** The null value. **TYPE='TIMERONCOST'** The high value in the range. The value -1 indicates that there is no upper bound. **TYPE='CARDINALITY'** The high value in the range. The value -1 indicates that there is no upper bound. **TYPE='DATA TAG'** The null value. **TYPE='ROUTINE SCHEMA'** The null value.
VALUE3	VARCHAR (128)	Y	The meaning of this value depends on the value in the TYPE column. If TYPE='ROUTINE SCHEMA', it specifies the schema name of the procedures that are called by the CALL statement. Otherwise, it is the null value.

SYSCAT.WORKCLASSES

Each row represents a work class defined for a work class set.

Table 283. SYSCAT.WORKCLASSES Catalog View

Column Name	Data Type	Nullable	Description
WORKCLASSNAME	VARCHAR (128)		Name of the work class.
WORKCLASSSETNAME	VARCHAR (128)	Y	Name of the work class set.
WORKCLASSID	INTEGER		Identifier for the work class.
WORKCLASSSETID	INTEGER		Identifier for the work class set to which this work class belongs. This column refers to the WORKCLASSSETID column in the SYSCAT.WORKCLASSSETS view.
CREATE_TIME	TIMESTAMP		Time at which the work class was created.
ALTER_TIME	TIMESTAMP		Time at which the work class was last altered.
EVALUATIONORDER	SMALLINT		Uniquely identifies the evaluation order used for choosing a work class within a work class set.

SYSCAT.WORKCLASSSETS

Each row represents a work class set.

Table 284. SYSCAT.WORKCLASSSETS Catalog View

Column Name	Data Type	Nullable	Description
WORKCLASSSETNAME	VARCHAR (128)		Name of the work class set.
WORKCLASSSETID	INTEGER		Identifier for the work class set.
CREATE_TIME	TIMESTAMP		Time at which the work class set was created.
ALTER_TIME	TIMESTAMP		Time at which the work class set was last altered.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.WORKLOADAUTH

Each row represents a user, group, or role that has been granted USAGE privilege on a workload.

Table 285. SYSCAT.WORKLOADAUTH Catalog View

Column Name	Data Type	Nullable	Description
WORKLOADID	INTEGER		Identifier for the workload.
WORKLOADNAME	VARCHAR (128)		Name of the workload.
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantee is an individual user
GRANTEE	VARCHAR (128)		Holder of the privilege.

Table 285. SYSCAT.WORKLOADAUTH Catalog View (continued)

Column Name	Data Type	Nullable	Description
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
USAGEAUTH	CHAR (1)		Indicates whether grantee holds USAGE privilege on the workload. • N = Not held • Y = Held

SYSCAT.WORKLOADCONNATTR

Each row represents a connection attribute in the definition of a workload.

Table 286. SYSCAT.WORKLOADCONNATTR Catalog View

Column Name	Data Type	Nullable	Description
WORKLOADID	INTEGER		Identifier for the workload.
WORKLOADNAME	VARCHAR (128)		Name of the workload.
CONNATTRTYPE	VARCHAR (30)		Type of the connection attribute. • 1 = APPLNAME • 2 = SYSTEM_USER • 3 = SESSION_USER • 4 = SESSION_USER GROUP • 5 = SESSION_USER ROLE • 6 = CURRENT CLIENT_USERID • 7 = CURRENT CLIENT_APPLNAME • 8 = CURRENT CLIENT_WRKSTNNAME • 9 = CURRENT CLIENT_ACCTNG • 10 = ADDRESS
CONNATTRVALUE	VARCHAR (1000)		Value of the connection attribute.

SYSCAT.WORKLOADS

Each row represents a workload.

Table 287. SYSCAT.WORKLOADS Catalog View

Column Name	Data Type	Nullable	Description
WORKLOADID	INTEGER		Identifier for the workload.
WORKLOADNAME	VARCHAR (128)		Name of the workload.
EVALUATIONORDER	SMALLINT		Evaluation order used for choosing a workload.
CREATE_TIME	TIMESTAMP		Time at which the workload was created.

Table 287. SYSCAT.WORKLOADS Catalog View (continued)

Column Name	Data Type	Nullable	Description
ALTER_TIME	TIMESTAMP		Time at which the workload was last altered.
ENABLED	CHAR (1)		• N = This workload is disabled. • Y = This workload is enabled.
ALLOWACCESS	CHAR (1)		• N = A UOW associated with this workload will be rejected. • Y = A unit of work (UOW) associated with this workload can access the database.
MAXDEGREE	SMALLINT		Maximum degree of parallelism for the workload. The valid values are: 1 to 32767, and -1. If MAXIMUM DEGREE is DEFAULT, the value is -1.
SERVICECLASSNAME	VARCHAR (128)		Name of the service subclass to which a unit of work (associated with this workload) is assigned.
PARENTSERVICECLASSNAME	VARCHAR (128)	Y	Name of the service superclass to which a unit of work (associated with this workload) is assigned.
COLLECTAGGACTDATA	CHAR (1)		Specifies what aggregate activity data should be captured for the workload by the applicable event monitor. • B = Collect base aggregate activity data • E = Collect extended aggregate activity data • N = None
COLLECTACTDATA	CHAR (1)		Specifies what activity data should be collected by the applicable event monitor. • D = Activity data with details • N = None • S = Activity data with details and section environment • V = Activity data with details and values. Applies when the COLLECT column is set to 'C' • W = Activity data without details • X = Activity data with details, section environment, and values
COLLECTACTPARTITION	CHAR (1)		Specifies where activity data is collected. • C = Coordinator member of the activity • D = All members

Table 287. SYSCAT.WORKLOADS Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLLECTDEADLOCK	CHAR (1)		Specifies that deadlock events should be collect by the applicable event monitor. • H = Collect deadlock data with past activities only • V = Collect deadlock data with past activities and values • W = Collect deadlock data without past activities and values
COLLECTLOCKTIMEOUT	CHAR (1)		Specifies that lock timeout events should be collect by the applicable event monitor. • H = Collect lock timeout data with past activities only • N = Do not not collect lock timeout data • V = Collect lock timeout data with past activities and values • W = Collect lock timeout data without past activities and values
COLLECTLOCKWAIT	CHAR (1)		Specifies that lock wait events should be collect by the applicable event monitor. • H = Collect lock wait data with past activities only • N = Do not not collect lock wait data • V = Collect lock wait data with past activities and values • W = Collect lock wait data without past activities and values
LOCKWAITVALUE	INTEGER		Specifies the time in milliseconds a lock should wait before a lock event is collected by the applicable event monitor; 0 when COLLECTLOCKWAIT = 'N'
COLLECTACTMETRICS	CHAR (1)		Specifies the monitoring level for activities submitted by an occurrence of the workload. • B = Collect base activity metrics • E = Collect extended activity metrics • N = None

Table 287. SYSCAT.WORKLOADS Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLLECTUOWDATAOPTIONS	VARCHAR (32)		Specifies what unit of work data is collected by the applicable event monitor. The first position in the string represents whether the collection of unit of work data is enabled. • B = Enabled and collect base unit of work data • N = Not enabled Starting from second position, each position in the string represents a specific extended option: • 2 = Package Reference List • 3 = Executable ID list Each position representing an extended option is then set to one of the following values: • Y = Extended option is included • N = Extended option is not included
COLLECTUOWDATA	CHAR (1)		Specifies what unit of work data should be collected by the applicable event monitor. • B = Collect base unit of work data • N = None • P = Collect base unit of work data and the package list This column is deprecated. Information for the column is available from COLLECTUOWDATAOPTIONS.
EXTERNALNAME	VARCHAR (128)	Y	Reserved for future use.
SECTIONACTUALSOPTIONS	VARCHAR (32)		Specifies what section actuals are collected during the execution of a section. The first position in the string represents the whether the collection of section actuals is enabled. • B = Enabled and collect basic operator cardinality counts and statistics for each object referenced by the section (DML statements only). • N = Not enabled. The second position is always 'N' and reserved for for future use.

Table 287. SYSCAT.WORKLOADS Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLLECTAGGUOWDATA	CHAR (1)		Specifies what aggregate unit of work data should be captured for the workload by the applicable event monitor. • B = Collect base aggregate unit of work data • N = None
PRIORITY **Attention:** This property is available to users of Db2 v11.5.5 and later versions.	CHAR(8)	N	Specifies the WLM priority for the workload. The priority can be set to one of the following values: • CRITICAL • HIGH • MEDIUM • LOW
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.WRAPOPTIONS

Each row represents a wrapper-specific option.

Table 288. SYSCAT.WRAPOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
WRAPNAME	VARCHAR (128)		Name of the wrapper.
OPTION	VARCHAR (128)		Name of the wrapper option.
SETTING	VARCHAR (2048)		Value of the wrapper option.

SYSCAT.WRAPPERS

Each row represents a registered wrapper.

Table 289. SYSCAT.WRAPPERS Catalog View

Column Name	Data Type	Nullable	Description
WRAPNAME	VARCHAR (128)		Name of the wrapper.
WRAPTYPE	CHAR (1)		Type of wrapper. • N = Non-relational • R = Relational
WRAPVERSION	INTEGER		Version of the wrapper.
LIBRARY	VARCHAR (255)		Name of the file that contains the code used to communicate with the data sources that are associated with this wrapper.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.XDBMAPGRAPHS

Each row represents a schema graph for an XDB map (XSR object).

Table 290. SYSCAT.XDBMAPGRAPHS Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Unique generated identifier for an XSR object.
OBJECTSCHEMA	VARCHAR (128)		Schema name of the XSR object.
OBJECTNAME	VARCHAR (128)		Unqualified name of the XSR object.
SCHEMAGRAPHID	INTEGER		Schema graph identifier, which is unique within an XDB map identifier.
NAMESPACE	VARCHAR (1001)	Y	String identifier for the namespace URI of the root element.
ROOTELEMENT	VARCHAR (1001)	Y	String identifier for the element name of the root element.

SYSCAT.XDBMAPSHREDTREES

Each row represents one shred tree for a given schema graph identifier.

Table 291. SYSCAT.XDBMAPSHREDTREES Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Unique generated identifier for an XSR object.
OBJECTSCHEMA	VARCHAR (128)		Schema name of the XSR object.
OBJECTNAME	VARCHAR (128)		Unqualified name of the XSR object.
SCHEMAGRAPHID	INTEGER		Schema graph identifier, which is unique within an XDB map identifier.
SHREDTREEID	INTEGER		Shred tree identifier, which is unique within an XDB map identifier.
MAPPINGDESCRIPTION	CLOB (1M)	Y	Diagnostic mapping information.

SYSCAT.XMLSTRINGS

Each row represents a single string and its unique string ID, used to condense structural XML data. The string is provided in both UTF-8 encoding and database code page encoding.

Table 292. SYSCAT.XMLSTRINGS Catalog View

Column Name	Data Type	Nullable	Description
STRINGID	INTEGER		Unique string ID.
STRING	VARCHAR (1001)		The string represented in the database code page.
STRING_UTF8	VARCHAR (1001)		The string in UTF-8 encoding (as stored in the catalog table).

SYSCAT.XSROBJECTAUTH

Each row represents a user, group, or role that has been granted the USAGE privilege on a particular XSR object.

Table 293. SYSCAT.XSROBJECTAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
OBJECTID	BIGINT		Identifier for the XSR object.
USAGEAUTH	CHAR (1)		Privilege to use the XSR object and its components. • N = Not held • Y = Held

SYSCAT.XSROBJECTCOMPONENTS

Each row represents an XSR object component.

Table 294. SYSCAT.XSROBJECTCOMPONENTS Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Unique generated identifier for an XSR object.
OBJECTSCHEMA	VARCHAR (128)		Schema name of the XSR object.
OBJECTNAME	VARCHAR (128)		Unqualified name of the XSR object.
COMPONENTID	BIGINT		Unique generated identifier for an XSR object component.
TARGETNAMESPACE	VARCHAR (1001)	Y	String identifier for the target namespace.
SCHEMALOCATION	VARCHAR (1001)	Y	String identifier for the schema location.
COMPONENT	BLOB (30M)		External representation of the component.
CREATE_TIME	TIMESTAMP		Time at which the XSR object component was registered.
STATUS	CHAR (1)		Registration status. • C = Complete • I = Incomplete

SYSCAT.XSROBJECTDEP

Each row represents a dependency of an XSR object on some other object. The XSR object depends on the object of type BTYPE of name BNAME, so a change to the object affects the XSR object.

Table 295. SYSCAT.XSROBJECTDEP Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Unique generated identifier for an XSR object.
OBJECTSCHEMA	VARCHAR (128)		Schema name of the XSR object.
OBJECTNAME	VARCHAR (128)		Unqualified name of the XSR object.
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: • A = Table alias • B = Trigger • C = Column • F = Routine • G = Global temporary table • H = Hierachy table • I = Index • K = Package • L = Detached table • N = Nickname • O = Privilege dependency on all subtables or subviews in a table or view hierarchy • Q = Sequence • R = User-defined data type • S = Materialized query table • T = Table (not typed) • U = Typed table • V = View (not typed) • W = Typed view • X = Index extension • Z = XSR object • m = Module • q = Sequence alias • u = Module alias • v = Global variable • * = Anchored to the row of a base table
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.

Table 295. SYSCAT.XSROBJECTDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which there is a dependency.
TABAUTH	SMALLINT	Y	If BTYPE = 'O', 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by a dependent trigger; null value otherwise.

SYSCAT.XSROBJECTDETAILS

Each row represents an XML schema repository object.

Table 296. SYSCAT.XSROBJECTDETAILS Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Unique generated identifier for an XML schema object.
OBJECTSCHEMA	VARCHAR (128)		Schema name of the XML schema object.
OBJECTNAME	VARCHAR (128)		Unqualified name of the XML schema object.
GRAMMAR	BLOB (127M)	Y	Binary representation of the grammar for the XML schema object.
PROPERTIES	BLOB (4190000)	Y	Properties document for the XML schema object.

SYSCAT.XSROBJECTHIERARCHIES

Each row represents the hierarchical relationship between an XSR object and its components.

Table 297. SYSCAT.XSROBJECTHIERARCHIES Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Identifier for an XSR object.
COMPONENTID	BIGINT		Identifier for an XSR component.
HTYPE	CHAR (1)		Hierarchy type. • D = Document • N = Top-level namespace • P = Primary document
TARGETNAMESPACE	VARCHAR (1001)	Y	String identifier for the component's target namespace.
SCHEMALOCATION	VARCHAR (1001)	Y	String identifier for the component's schema location.

SYSCAT.XSROBJECTS

Each row represents an XML schema repository object.

Table 298. SYSCAT.XSROBJECTS Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Unique generated identifier for an XSR object.
OBJECTSCHEMA	VARCHAR (128)		Schema name of the XSR object.
OBJECTNAME	VARCHAR (128)		Unqualified name of the XSR object.
TARGETNAMESPACE	VARCHAR (1001)	Y	String identifier for the target namespace, or public identifier.
SCHEMALOCATION	VARCHAR (1001)	Y	String identifier for the schema location, or system identifier.
OBJECTINFO	XML	Y	Metadata document.
OBJECTTYPE	CHAR (1)		XSR object type. • D = DTD • E = External entity • S = XML schema
OWNER	VARCHAR (128)		Authorization ID of the owner of the XSR object.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
CREATE_TIME	TIMESTAMP		Time at which the object was registered.
ALTER_TIME	TIMESTAMP		Time at which the object was last updated (replaced).
STATUS	CHAR (1)		Registration status. • C = Complete • I = Incomplete • R = Replace • T = Temporary
DECOMPOSITION	CHAR (1)		Indicates whether or not decomposition (shredding) is enabled on this XSR object. • N = Not enabled • X = Inoperative • Y = Enabled
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSIBM.SYSDUMMY1

Contains one row. This view is available for applications that require compatibility with Db2 for z/OS.

Table 299. SYSIBM.SYSDUMMY1 Catalog View

Column Name	Data Type	Nullable	Description
IBMREQD	CHAR (1)		'Y'

SYSSTAT.COLDIST

Each row represents the *n*th most frequent value of some column, or the *n*th quantile (cumulative distribution) value of the column. Applies to columns of real tables only (not views). No statistics are recorded for inherited columns of typed tables.

Table 300. SYSSTAT.COLDIST Catalog View

Column Name	Data Type	Nullable	Updat-able	Description
TABSCHEMA	VARCHAR (128)			Schema name of the table to which the statistics apply.
TABNAME	VARCHAR (128)			Unqualified name of the table to which the statistics apply.
COLNAME	VARCHAR (128)			Name of the column to which the statistics apply.
TYPE	CHAR (1)			• F = Frequency value • Q = Quantile value
SEQNO	SMALLINT			If TYPE = "F", *n* in this column identifies the *n*th most frequent value. If TYPE = "Q", *n* in this column identifies the *n*th quantile value.
COLVALUE[1]	VARCHAR (254)	Y	Y	Data value as a character literal or a null value.
VALCOUNT	BIGINT		Y	If TYPE = "F", VALCOUNT is the number of occurrences of COLVALUE in the column. If TYPE = "Q", VALCOUNT is the number of rows whose value is less than or equal to COLVALUE.
DISTCOUNT[2]	BIGINT	Y	Y	If TYPE = "Q", this column records the number of distinct values that are less than or equal to COLVALUE (the null value if unavailable).

Note:

1. In the catalog view, the value of COLVALUE is always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column's table, and will therefore use actual column values when applied during query optimization.
2. DISTCOUNT is collected only for columns that are the first key column in an index.

SYSSTAT.COLGROUPDIST

Each row represents the value of the column in a column group that makes up the *n*th most frequent value of the column group or the *n*th quantile value of the column group.

Table 301. SYSSTAT.COLGROUPDIST Catalog View

Column Name	Data Type	Nullable	Updatable	Description
COLGROUPID	INTEGER			Identifier for the column group.
TYPE	CHAR (1)			• F = Frequency value • Q = Quantile value
ORDINAL	SMALLINT			Ordinal number of the column in the column group.
SEQNO	SMALLINT			If TYPE = 'F', *n* in this column identifies the *n*th most frequent value. If TYPE = 'Q', *n* in this column identifies the *n*th quantile value.
COLVALUE	VARCHAR (254)		Y	Data value as a character literal or a null value.

SYSSTAT.COLGROUPDISTCOUNTS

Each row represents the distribution statistics that apply to the *n*th most frequent value of a column group or the *n*th quantile of a column group.

Table 302. SYSSTAT.COLGROUPDISTCOUNTS Catalog View

Column Name	Data Type	Nullable	Updatable	Description
COLGROUPID	INTEGER			Identifier for the column group.
TYPE	CHAR (1)			• F = Frequency value • Q = Quantile value
SEQNO	SMALLINT			Sequence number *n* representing the *n*th TYPE value.
VALCOUNT	BIGINT		Y	If TYPE = 'F', VALCOUNT is the number of occurrences of COLVALUE for the column group with this SEQNO. If TYPE = 'Q', VALCOUNT is the number of rows whose value is less than or equal to COLVALUE for the column group with this SEQNO.
DISTCOUNT	BIGINT		Y	If TYPE = 'Q', this column records the number of distinct values that are less than or equal to COLVALUE for the column group with this SEQNO (the null value if unavailable).

SYSSTAT.COLGROUPS

Each row represents a column group and statistics that apply to the entire column group.

Table 303. SYSSTAT.COLGROUPS Catalog View

Column Name	Data Type	Nullable	Updat-able	Description
COLGROUPSCHEMA	VARCHAR (128)			Schema name of the column group.
COLGROUPNAME	VARCHAR (128)			Unqualified name of the column group.
COLGROUPID	INTEGER			Identifier for the column group.
COLGROUPCARD	BIGINT		Y	Cardinality of the column group.
NUMFREQ_VALUES	SMALLINT			Number of frequent values collected for the column group.
NUMQUANTILES	SMALLINT			Number of quantiles collected for the column group.

SYSSTAT.COLUMNS

Each row represents a column defined for a table, view, or nickname.

Table 304. SYSSTAT.COLUMNS Catalog View

Column Name	Data Type	Nullable	Updat-able	Description
TABSCHEMA	VARCHAR (128)			Schema name of the table, view, or nickname that contains the column.
TABNAME	VARCHAR (128)			Unqualified name of the table, view, or nickname that contains the column.
COLNAME	VARCHAR (128)			Name of the column.
COLCARD	BIGINT		Y	Number of distinct values in the column; -1 if statistics are not collected; -2 for inherited columns and columns of hierarchy tables.
HIGH2KEY[1]	VARCHAR (254)	Y	Y	Second-highest data value. Representation of numeric data changed to character literals. Empty if statistics are not collected. Empty for inherited columns and columns of hierarchy tables.
LOW2KEY[1]	VARCHAR (254)	Y	Y	Second-lowest data value. Representation of numeric data changed to character literals. Empty if statistics are not collected. Empty for inherited columns and columns of hierarchy tables.

Table 304. SYSSTAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Updat-able	Description
AVGCOLLEN	INTEGER		Y	Average space in bytes when the column is stored in database memory or a temporary table. For LOB data types that are not inlined, LONG data types, and XML documents, the value used to calculate the average column length is the length of the data descriptor. An extra byte is required if the column is nullable; -1 if statistics have not been collected; -2 for inherited columns and columns of hierarchy tables. Note: The average space required to store the column on disk may be different than the value represented by this statistic.
NUMNULLS	BIGINT		Y	Number of null values in the column; -1 if statistics are not collected.
PCTINLINED	SMALLINT			Percentage of inlined data for columns with VARCHAR, VARGRAPHIC, LOB, or XML data types. -1 if statistics have not been collected or the column data type does not support storing data outside the row. Also -1 for VARCHAR and VARGRAPHIC column if the table is organized by column or the table is organized by row and the row size of the table does not exceed the maximum record length for the page size of the table space.
SUB_COUNT	SMALLINT		Y	Average number of sub-elements in the column. Applicable to character string columns only.
SUB_DELIM_LENGTH	SMALLINT		Y	Average length of the delimiters that separate each sub-element in the column. Applicable to character string columns only.
AVGCOLLENCHAR	INTEGER		Y	Average number of characters (based on the collation in effect for the column) required for the column; -1 if the data type of the column is long, LOB, or XML or if statistics have not been collected; -2 for inherited columns and columns of hierarchy tables.

Table 304. SYSSTAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Updat-able	Description
PCTENCODED	SMALLINT		Y	Percentage of values that are encoded as a result of compression for a column in a column-organized table; -1 if the table is not organized by column or if statistics are not collected; -2 for inherited columns and columns of hierarchy tables.
PAGEVARIANCERATIO	DOUBLE		Y	Reserved for future use.
AVGENCODEDCOLLEN	DOUBLE		Y	Average space in bytes when the column is stored in database memory, taking into account that some of the column values might be compressed; -1 if the table is not organized by column or if statistics are not collected; -2 for inherited columns and columns of hierarchy tables.

Note:

1. In the catalog view, the values of HIGH2KEY and LOW2KEY are always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column's table, and will therefore use actual column values when applied during query optimization.

SYSSTAT.INDEXES

Each row represents an index. Indexes on typed tables are represented by two rows: one for the "logical index" on the typed table, and one for the "H-index" on the hierarchy table.

Table 305. SYSSTAT.INDEXES Catalog View

Column Name	Data Type	Nullable	Updat-able	Description
INDSCHEMA	VARCHAR (128)			Schema name of the index.
INDNAME	VARCHAR (128)			Unqualified name of the index.
TABSCHEMA	VARCHAR (128)			Schema name of the table or nickname on which the index is defined.
TABNAME	VARCHAR (128)			Unqualified name of the table or nickname on which the index is defined.
COLNAMES	VARCHAR (640)			This column is no longer used and will be removed in the next release. Use SYSCAT.INDEXCOLUSE for this information.
NLEAF	BIGINT		Y	Number of leaf pages; -1 if statistics are not collected.
NLEVELS	SMALLINT		Y	Number of index levels; -1 if statistics are not collected.

Table 305. SYSSTAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Updat-able	Description
FIRSTKEYCARD	BIGINT		Y	Number of distinct first-key values; -1 if statistics are not collected.
FIRST2KEYCARD	BIGINT		Y	Number of distinct keys using the first two columns of the index; -1 if statistics are not collected, or if not applicable.
FIRST3KEYCARD	BIGINT		Y	Number of distinct keys using the first three columns of the index; -1 if statistics are not collected, or if not applicable.
FIRST4KEYCARD	BIGINT		Y	Number of distinct keys using the first four columns of the index; -1 if statistics are not collected, or if not applicable.
FULLKEYCARD	BIGINT		Y	Number of distinct full-key values; -1 if statistics are not collected.
CLUSTERRATIO[4]	SMALLINT		Y	Degree of data clustering with the index; -1 if statistics are not collected or if detailed index statistics are collected (in which case, CLUSTERFACTOR will be used instead).
CLUSTERFACTOR[4]	DOUBLE		Y	Finer measurement of the degree of clustering; -1 if statistics are not collected or if the index is defined on a nickname.
SEQUENTIAL_PAGES	BIGINT		Y	Number of leaf pages located on disk in index key order with few or no large gaps between them; -1 if statistics are not collected.
DENSITY	INTEGER		Y	Ratio of SEQUENTIAL_PAGES to number of prefetched pages. Expressed as a percentage; -1 if statistics are not collected.
PAGE_FETCH_PAIRS[4]	VARCHAR (520)		Y	A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the table with this index using that hypothetical buffer. Zero-length string if no data is available.
NUMRIDS[4]	BIGINT		Y	Total number of row identifiers (RIDs) or block identifiers (BIDs) in the index; -1 if not known.

Table 305. SYSSTAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Updat-able	Description
NUMRIDS_DELETED[4]	BIGINT		Y	Total number of row identifiers (or block identifiers) in the index that are marked deleted, excluding those identifiers on leaf pages on which all the identifiers are marked deleted.
NUM_EMPTY_LEAFS	BIGINT		Y	Total number of index leaf pages that have all of their row identifiers (or block identifiers) marked deleted.
AVERAGE_RANDOM_ FETCH_PAGES[1,2,4]	DOUBLE		Y	Average number of random table pages between sequential page accesses when fetching using the index; -1 if not known.
AVERAGE_RANDOM_ PAGES[2]	DOUBLE		Y	Average number of random table pages between sequential page accesses; -1 if not known.
AVERAGE_SEQUENCE_ GAP[2]	DOUBLE		Y	Gap between index page sequences. Detected through a scan of index leaf pages, each gap represents the average number of index pages that must be randomly fetched between sequences of index pages; -1 if not known.
AVERAGE_SEQUENCE_ FETCH_GAP[1,2,4]	DOUBLE		Y	Gap between table page sequences when fetching using the index. Detected through a scan of index leaf pages, each gap represents the average number of table pages that must be randomly fetched between sequences of table pages; -1 if not known.
AVERAGE_SEQUENCE_ PAGES[2]	DOUBLE		Y	Average number of index pages that are accessible in sequence (that is, the number of index pages that the prefetchers would detect as being in sequence); -1 if not known.
AVERAGE_SEQUENCE_ FETCH_PAGES[1,2,4]	DOUBLE		Y	Average number of table pages that are accessible in sequence (that is, the number of table pages that the prefetchers would detect as being in sequence) when fetching using the index; -1 if not known.
AVGPARTITION_ CLUSTERRATIO[3,4]	SMALLINT		Y	Degree of data clustering within a single data partition. -1 if the table is not partitioned, if statistics are not collected, or if detailed statistics are collected (in which case AVGPARTITION_ CLUSTERFACTOR will be used instead).

Table 305. SYSSTAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Updat-able	Description
AVGPARTITION_ CLUSTERFACTOR[3,4]	DOUBLE		Y	Finer measurement of the degree of clustering within a single data partition. -1 if the table is not partitioned, if statistics are not collected, or if the index is defined on a nickname.
AVGPARTITION_PAGE_ FETCH_PAIRS[3,4]	VARCHAR (520)		Y	A list of paired integers in character form. Each pair represents a potential buffer pool size and the corresponding page fetches required to access a single data partition from the table. Zero-length string if no data is available, or if the table is not partitioned.
DATAPARTITION_ CLUSTERFACTOR	DOUBLE		Y	A statistic measuring the "clustering" of the index keys with regard to data partitions. It is a number between 0 and 1, with 1 representing perfect clustering and 0 representing no clustering.
INDCARD	BIGINT		Y	Cardinality of the index. This might be different from the cardinality of the table for indexes that do not have a one-to-one relationship between the table rows and the index entries.
PCTPAGESSAVED	SMALLINT			Approximate percentage of pages saved in the index as a result of index compression. -1 if statistics are not collected.
AVGLEAFKEYSIZE	INTEGER		Y	Average index key size for keys on leaf pages in the index.
AVGNLEAFKEYSIZE	INTEGER		Y	Average index key size for keys on non-leaf pages in the index.

Table 305. SYSSTAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Updat-able	Description

Note:

1. When using DMS table spaces, this statistic cannot be computed.
2. Prefetch statistics are not gathered during a LOAD...STATISTICS USE PROFILE, or a CREATE INDEX...COLLECT STATISTICS operation, or when the database configuration parameter *seqdetect* is turned off.
3. AVGPARTITION_CLUSTERRATIO, AVGPARTITION_CLUSTERFACTOR, and AVGPARTITION_PAGE_FETCH_PAIRS measure the degree of clustering within a single data partition (local clustering). CLUSTERRATIO, CLUSTERFACTOR, and PAGE_FETCH_PAIRS measure the degree of clustering in the entire table (global clustering). Global clustering and local clustering values can diverge significantly if the table partitioning key is not a prefix of the index key, or when the table partitioning key and the index key are logically independent of each other.
4. This statistic cannot be updated if the index type is 'XPTH' (an XML path index).
5. Because logical indexes on an XML column do not have statistics, the SYSSTAT.INDEXES catalog view excludes rows whose index type is 'XVIL'.
6. There is a limitation for small and medium indexes. The density column will have values, which are not counted as described above. This behavior will not impact optimizer costing.

SYSSTAT.ROUTINES

Each row represents a user-defined routine (scalar function, table function, sourced function, method, or procedure). Does not include built-in functions.

Table 306. SYSSTAT.ROUTINES Catalog View

Column Name	Data Type	Nullable	Updat-able	Description
ROUTINESCHEMA	VARCHAR (128)			Schema name of the routine if ROUTINEMODULENAME is null; otherwise schema name of the module to which the routine belongs.
ROUTINEMODULENAME	VARCHAR (128)			Unqualified name of the module to which the routine belongs. The null value if not a module routine.
ROUTINENAME	VARCHAR (128)			Unqualified name of the routine.
ROUTINETYPE	CHAR (1)			Type of routine. • F = Function • M = Method • P = Procedure
SPECIFICNAME	VARCHAR (128)			Name of the routine instance (might be system-generated).
IOS_PER_INVOC	DOUBLE		Y	Estimated number of inputs/outputs (I/Os) per invocation; 0 is the default; -1 if not known.

Table 306. SYSSTAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Updat-able	Description
INSTS_PER_INVOC	DOUBLE		Y	Estimated number of instructions per invocation; 450 is the default; -1 if not known.
IOS_PER_ARGBYTE	DOUBLE		Y	Estimated number of I/Os per input argument byte; 0 is the default; -1 if not known.
INSTS_PER_ARGBYTE	DOUBLE		Y	Estimated number of instructions per input argument byte; 0 is the default; -1 if not known.
PERCENT_ARGBYTES	SMALLINT		Y	Estimated average percent of input argument bytes that the routine will actually read; 100 is the default; -1 if not known.
INITIAL_IOS	DOUBLE		Y	Estimated number of I/Os performed the first time that the routine is invoked; 0 is the default; -1 if not known.
INITIAL_INSTS	DOUBLE		Y	Estimated number of instructions executed the first time the routine is invoked; 0 is the default; -1 if not known.
CARDINALITY	BIGINT		Y	Predicted cardinality of a table function; -1 if not known, or if the routine is not a table function.
SELECTIVITY	DOUBLE		Y	For user-defined predicates; -1 if there are no user-defined predicates.

SYSSTAT.TABLES

Each row represents a table, view, alias, or nickname. Each table or view hierarchy has one additional row representing the hierarchy table or hierarchy view that implements the hierarchy. Catalog tables and views are included.

Table 307. SYSSTAT.TABLES Catalog View

Column Name	Data Type	Nullabl e	Updat-able	Description
TABSCHEMA	VARCHAR (128)			Schema name of the object.
TABNAME	VARCHAR (128)			Unqualified name of the object.
CARD	BIGINT		Y	Total number of rows in the table; -1 if statistics are not collected.
NPAGES	BIGINT		Y	Total number of pages on which the rows of the table exist; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.

Table 307. SYSSTAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Updatable	Description
MPAGES	BIGINT		Y	Total number of pages for table metadata. Non-zero only for a table that is organized by column; -1 for a view, an alias, or if statistics are not collected; -2 for subtables or hierarchy tables.
FPAGES	BIGINT		Y	Total number of pages; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.
NPARTITIONS	BIGINT			Reserved for future use.
NFILES	BIGINT			Reserved for future use.
TABLESIZE	BIGINT			Reserved for future use.
OVERFLOW	BIGINT		Y	Total number of overflow records in the table; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.
CLUSTERED	CHAR (1)	Y		• T = Table is clustered by insert time • Y = Table is clustered by dimensions (even if only by one dimension) • Null value = Table is not clustered by dimensions or insert time
ACTIVE_BLOCKS	BIGINT		Y	Total number of active blocks in the table, or -1. Applies to multidimensional clustering (MDC) tables or insert time clustering (ITC) tables only.
AVGCOMPRESSEDROWSIZE	SMALLINT		Y	Average length (in bytes) of compressed rows in this table; -1 if statistics are not collected.
AVGROWCOMPRESSIONRATIO	REAL		Y	For compressed rows in the table, this is the average compression ratio by row; that is, the average uncompressed row length divided by the average compressed row length; -1 if statistics are not collected.
AVGROWSIZE	SMALLINT			Average length (in bytes) of both compressed and uncompressed rows in this table; -1 if statistics are not collected.

Table 307. SYSSTAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Updatable	Description
PCTROWSCOMPRESSED	REAL		Y	Compressed rows as a percentage of the total number of rows in the table; -1 if statistics are not collected.
PCTPAGESSAVED	SMALLINT		N	Approximate percentage of pages saved in a row-organized table as a result of row compression. For a column-organized table, the estimate is based on the number of data pages needed to store the table in uncompressed row organization.-1 if statistics are not collected.
PCTEXTENDEDROWS	REAL			Extended rows as a percentage of the total number of rows in the table; -1 if statistics are not collected.

SQL and XML limits

The tables in this topic describe SQL and XML limits. Adhering to the most restrictive case can help you to design application programs that are easily portable.

Table 308 on page 2125 lists limits in bytes. These limits are enforced after conversion from the application code page to the database code page when creating identifiers. The limits are also enforced after conversion from the database code page to the application code page when retrieving identifiers from the database. If during either of these processes the identifier length limit is exceeded, truncation occurs, or an error is returned.

Character limits vary depending on the code page of the database and the code page of the application. For example, because the width of a UTF-8 character can range 1 - 4 bytes, the character limit for an identifier in a Unicode table whose limit is 128 bytes must range 32 - 128 characters, depending on which characters are used. If an attempt is made to create an identifier whose name is longer than the limit for this table after conversion to the database code page, an error is returned.

Applications that store identifier names must be able to handle the potentially increased size of identifiers after code page conversion occurs. When identifiers are retrieved from the catalog, they are converted to the application code page. Conversion from the database code page to the application code page can result in an identifier becoming longer than the byte limit for the table. If a host variable that is declared by the application cannot store the entire identifier after code page conversion, it is truncated. If that is unacceptable, the host variable can be increased in size to be able to accept the entire identifier name.

The same rules apply to Db2 utilities retrieving data and converting it to a user-specified code page. If a Db2 utility, such as export, is retrieving the data and forcing conversion to a user-specified code page (by using the export CODEPAGE modifier or the **DB2CODEPAGE** registry variable), and the identifier expands beyond the limit that is documented in this table because of code page conversion, an error might be returned or the identifier might be truncated.

Table 308. Identifier Length Limits

Description	Maximum in Bytes
Alias name	128
Attribute name	128

Table 308. Identifier Length Limits (continued)

Description	Maximum in Bytes
Audit policy name	128
Authorization name (can be single-byte characters only)	128
Buffer pool name	18
Column name[2]	128
Constraint name	128
Correlation name	128
Cursor name	128
Data partition name	128
Data source column name	255
Data source index name	128
Data source name	128
Data source table name (*remote-table-name*)	128
Database partition group name	128
Database partition name	128
Event monitor name	128
External program name	128
Function mapping name	128
Group name	128
Host identifier[1]	255
Identifier for a data source user (*remote-authorization-name*)	128
Identifier in an SQL procedure (condition name, for loop identifier, label, result set locator, statement name, variable name)	128
Index name	128
Index extension name	18
Index specification name	128
Label name	128
Namespace uniform resource identifier (URI)	1000
Nickname	128
Package name	128
Package version ID	64
Parameter name	128
Password to access a data source	32
Procedure name	128
Role name	128
Savepoint name	128

Table 308. Identifier Length Limits (continued)

Description	Maximum in Bytes
Schema name[2,3]	128
Security label component name	128
Security label name	128
Security policy name	128
Sequence name	128
Server (database alias) name	8
Specific name	128
SQL condition name	128
SQL variable name	128
Statement name	128
Storage Group	128
Table name	128
Table space name	18
Transform group name	18
Trigger name	128
Trusted context name	128
Type mapping name	18
User-defined function name	128
User-defined method name	128
User-defined type name[2]	128
View name	128
Wrapper name	128
XML element name, attribute name, or prefix name	1000
XML schema location uniform resource identifier (URI)	1000

Note:

1. Individual host language compilers might have a more restrictive limit on variable names.
2. The SQLDA structure is limited to storing 30-byte column names, 18-byte user-defined type names, and 8-byte schema names for user-defined types. Because the SQLDA is used in the DESCRIBE statement, embedded SQL applications that use the DESCRIBE statement to retrieve column or user-defined type name information must conform to these limits.
3. Schema names that are shorter than 8-bytes are padded with blanks and stored in the catalog as 8-byte names.

Table 309. Numeric Limits

Description	Limit
Smallest SMALLINT value	-32,768

Table 309. Numeric Limits (continued)

Description	Limit
Largest SMALLINT value	+32,767
Smallest INTEGER value	-2,147,483,648
Largest INTEGER value	+2,147,483,647
Smallest BIGINT value	-9,223,372,036,854,775,808
Largest BIGINT value	+9,223,372,036,854,775,807
Largest decimal precision	31
Maximum exponent (E_{max}) for REAL values	38
Smallest REAL value	-3.402E+38
Largest REAL value	+3.402E+38
Minimum exponent (E_{min}) for REAL values	-37
Smallest positive REAL value	+1.175E-37
Largest negative REAL value	-1.175E-37
Maximum exponent (E_{max}) for DOUBLE values	308
Smallest DOUBLE value	-1.79769E+308
Largest DOUBLE value	+1.79769E+308
Minimum exponent (E_{min}) for DOUBLE values	-307
Smallest positive DOUBLE value	+2.225E-307
Largest negative DOUBLE value	-2.225E-307
Maximum exponent (E_{max}) for DECFLOAT(16) values	384
Smallest DECFLOAT(16) value[1]	-9.999999999999999E+384
Largest DECFLOAT(16) value	9.999999999999999E+384
Minimum exponent (E_{min}) for DECFLOAT(16) values	-383
Smallest positive DECFLOAT(16) value	1.000000000000000E-383
Largest negative DECFLOAT(16) value	-1.000000000000000E-383
Maximum exponent (E_{max}) for DECFLOAT(34) values	6144
Smallest DECFLOAT(34) value[1]	-9.999999999999999999999999999999999E+6144
Largest DECFLOAT(34) value	9.999999999999999999999999999999999E+6144
Minimum exponent (E_{min}) for DECFLOAT(34) values	-6143

Table 309. Numeric Limits (continued)

Description	Limit
Smallest positive DECFLOAT(34) value	1.000000000000000000000000000000000E-6143
Largest negative DECFLOAT(34) value	-1.000000000000000000000000000000000E-6143

Note:

1. These are the limits of normal decimal floating-point numbers. Valid decimal floating-point values include the special values NAN, -NAN, SNAN, -SNAN, INFINITY, and -INFINITY. In addition, valid values include subnormal numbers.

   Subnormal numbers are nonzero numbers whose adjusted exponents are less than E_{min}. For a subnormal number, the minimum value of the exponent is E_{min} - (*precision*-1), called E_{tiny}, where *precision* is the working precision (16 or 34). That is, subnormal numbers extend the range of numbers close to zero by 15 or 33 orders of magnitude for DECFLOAT(16) or DECFLOAT(34), respectively. Subnormal numbers are different from normal numbers because the maximum number of digits for a subnormal number is less than the working precision (16 or 34). Decimal floating-point cannot represent the subnormal numbers with the same accuracy as it can represent normal numbers. The smallest positive subnormal number for DECFLOAT(34) is $1x10^{-6176}$, which contains only one digit, whereas the smallest positive normal number for DECFLOAT(34) is $1.000000000000000000000000000000000x10^{-6143}$, which contains 34 digits. The smallest positive subnormal number for DECFLOAT(16) is $1x10^{-398}$.

Table 310. String Limits

Description	Limit
Maximum length of CHAR (in bytes or OCTETS)	255
Maximum length of CHAR (in CODEUNITS32)	63
Maximum length of VARCHAR (in bytes or OCTETS)[2]	32,672
Maximum length of VARCHAR (in CODEUNITS32)[2]	8168
Maximum length of LONG VARCHAR (in bytes)[1]	32,700
Maximum length of CLOB (in bytes or OCTETS)	2,147,483,647
Maximum length of CLOB (in CODEUNITS32)	536,870,911
Maximum length of serialized XML (in bytes)	2,147,483,647
Maximum length of GRAPHIC (in double-byte characters or CODEUNITS16)	127
Maximum length of GRAPHIC (in CODEUNITS32)	63
Maximum length of VARGRAPHIC (in double-byte characters or CODEUNITS16)[2]	16,336
Maximum length of VARGRAPHIC (in CODEUNITS32)	8168
Maximum length of LONG VARGRAPHIC (in double-byte characters)[1]	16,350
Maximum length of DBCLOB (in double-byte characters or CODEUNITS16)	1,073,741,823
Maximum length of DBCLOB (in CODEUNITS32)	536,870,911
Maximum length of BINARY (in bytes)	255

Table 310. String Limits (continued)

Description	Limit
Maximum length of VARBINARY (in bytes)[2]	32,672
Maximum length of BLOB (in bytes)	2,147,483,647
Maximum length of character constant	32,672
Maximum length of graphic constant	16,336
Maximum length of concatenated character string	2,147,483,647
Maximum length of concatenated graphic string	1,073,741,823
Maximum length of concatenated binary string	2,147,483,647
Maximum number of hexadecimal constant digits	32,672
Largest instance of a structured type column object at run time (in gigabytes)	1
Maximum size of a catalog comment (in bytes)	254

Note:

1. The LONG VARCHAR and LONG VARGRAPHIC data types are deprecated and might be removed in a future release.
2. For page size-specific string limits for column-organized tables, see Table 311 on page 2130.

Table 311. Page Size-specific String Limits for Column-organized Tables

Description	4K page size limit	8K page size limit	16K page size limit	32K page size limit
Maximum length of VARCHAR (in bytes)	3920	8016	16,208	32,592
Maximum length of VARCHAR (in CODEUNITS32)	980	2004	4052	8148
Maximum length of VARGRAPHIC (in bytes)	1960	4008	8104	16,296
Maximum length of VARBINARY (in bytes)	3920	8016	16,208	32,592

Note: A column-organized table has an overhead of 176 bytes per page.

Table 312. Page Size-specific String Limits for Column-organized Tables. These limits are only applicable when you increase the column length by using ALTER TABLE

Description	4K page size limit	8K page size limit	16K page size limit	32K page size limit
Maximum length of VARCHAR (in bytes)	3888	7984	16,176	32,560
Maximum length of VARGRAPHIC (in bytes)	1944	3992	8088	16,280

Table 313. XML Limits

Description	Limit
Maximum depth of an XML document (in levels)	125
Maximum size of an XML schema document (in bytes)	31,457,280

Table 314. Datetime Limits

Description	Limit
Smallest DATE value	0001-01-01
Largest DATE value	9999-12-31
Smallest TIME value	00:00:00
Largest TIME value	24:00:00
Smallest TIMESTAMP value	0001-01-01-00.00.00.000000000000
Largest TIMESTAMP value	9999-12-31-24.00.00.000000000000
Smallest timestamp precision	0
Largest timestamp precision	12

Table 315. Database Manager Limits

Category	Description	Limit
Applications	Maximum number of host variable declarations in a precompiled program[3]	Storage. This number can go down with future versions. It is recommended to stay under 10000.
	Maximum length of a host variable value (in bytes)	2,147,483,647
	Maximum number of declared cursors in a program	storage
	Maximum number of rows that are changed in a unit of work	storage
	Maximum number of cursors that are opened at one time	storage
	Maximum number of connections per process within a database client	512
	Maximum number of simultaneously opened LOB locators in a transaction	4,194,304
	Maximum size of an SQLDA (in bytes)	storage
	Maximum number of prepared statements	storage

Table 315. Database Manager Limits (continued)

Category	Description	Limit
Buffer Pools	Maximum NPAGES in a buffer pool for 32-bit releases	1,048,576
	Maximum NPAGES in a buffer pool for 64-bit releases	2,147,483,647
	Maximum total size of all buffer pool slots (4K)	2,147,483,646
Concurrency	Maximum number of concurrent users of a server[4]	64,000
	Maximum number of concurrent users per instance	64,000
	Maximum number of concurrent applications per database	60,000
	Maximum number of databases per instance concurrently in use	256
Constraints	Maximum number of constraints on a table	storage
	Maximum number of columns in a UNIQUE constraint (supported through a UNIQUE index)	64
	Maximum combined length of columns in a UNIQUE constraint (supported through a UNIQUE index, in bytes)[8]	8192
	Maximum number of referencing columns in a foreign key	64
	Maximum combined length of referencing columns in a foreign key (in bytes)[8]	8192
	Maximum length of a check constraint specification (in bytes)	65,535
Databases	Maximum database partition number	999
	Maximum members in a Db2 pureScale environment	128

Table 315. Database Manager Limits (continued)

Category	Description	Limit
Indexes	Maximum number of indexes on a table	32,767 or storage
	Maximum number of columns in an index key	64
	Maximum length of an index key, including all overhead[6 8]	*indexpagesize*/4
	Maximum length of a variable index key part (in bytes)[7]	1022 or storage
	Maximum size of an index per database partition in an SMS table space (in terabytes)[6]	64
	Maximum size of an index per database partition in a regular DMS table space (in gigabytes)[6]	512
	Maximum size of an index per database partition in a large DMS table space (in terabytes)[6]	64
	Maximum length of a variable index key part for an index over XML data (in bytes)[7]	*pagesize*/4 - 207
Log records	Maximum Log Sequence Number	0xFFFF FFFF FFFF FFFF
Monitoring	Maximum number of simultaneously active event monitors	128
	In a partitioned database environment, maximum number of simultaneously active GLOBAL event monitors	32
Routines	Maximum number of parameters in a procedure with LANGUAGE SQL	32,767
	Maximum number of parameters in an external procedure with PROGRAM TYPE MAIN	32,767
	Maximum number of parameters in an external procedure with PROGRAM TYPE SUB	90
	Maximum number of parameters in a cursor value constructor	32,767
	Maximum number of parameters in a user-defined function	90
	Maximum number of nested levels for routines	64
	Maximum number of schemas in the SQL path	64
	Maximum length of the SQL path (in bytes)	2048
Security	Maximum number of elements in a security label component of type set or tree	64
	Maximum number of elements in a security label component of type array	65,535
	Maximum number of security label components in a security policy	16

Table 315. Database Manager Limits (continued)		
Category	**Description**	**Limit**
SQL	Maximum total length of an SQL statement (in bytes)	2,097,152
	Maximum number of tables that are referenced in an SQL statement or a view	storage
	Maximum number of host variable references in a dynamic SQL statement	32,767
	Maximum number of constants in a statement	storage
	Maximum number of elements in a select list[6]	2048
	Maximum number of predicates in a WHERE or HAVING clause	storage
	Maximum number of columns in a GROUP BY clause[6]	2048
	Maximum total length of columns in a GROUP BY clause (in bytes)[6]	32,677
	Maximum number of columns in an ORDER BY clause[6]	2048
	Maximum total length of columns in an ORDER BY clause (in bytes)[6]	32,677
	Maximum level of subquery nesting	storage
	Maximum number of subqueries in a single statement	storage
	Maximum number of values in an insert operation[6]	2048
Storage Groups	Maximum number of storage groups in a database	256
	Maximum number of storage paths in a storage group	128
	Maximum length of a storage path (in bytes)	175

Table 315. Database Manager Limits (continued)

Category	Description	Limit
Tables and Views	Maximum number of columns in a table [6] [10]	2048
	Maximum number of columns in a view[1]	5000
	Maximum number of columns in a data source table or view that is referenced by a nickname	5000
	Maximum number of columns in a distribution key[5]	500
	Maximum length of a row, including all overhead[2] [6] [9]	32,677
	Maximum number of rows in a non-partitioned table, per database partition	128×10^{10}
	Maximum number of rows in a data partition, per database partition	128×10^{10}
	Maximum size of a table per database partition in a regular table space (in gigabytes)[3] [6]	512
	Maximum size of a table per database partition in a large DMS table space (in terabytes)[6]	64
	Maximum number of data partitions for a single table	32,767
	Maximum number of table partitioning columns	16
	Maximum number of fields in a user-defined row data type	1012
Table Spaces	Maximum size of a LOB object per table or per table partition (in terabytes)	4
	Maximum size of an LF object per table or per table partition (in terabytes)	2
	Maximum number of table spaces in a database	32,768
	Maximum number of tables in an SMS table space	65,532
	Maximum size of a regular DMS table space (in gigabytes) [3] [6]	512
	Maximum size of a large DMS table space (in terabytes) [3] [6]	64
	Maximum size of a temporary DMS table space (in terabytes) [3] [6]	64
	Maximum number of table objects in a DMS table space	See Table 316 on page 2136
Triggers	Maximum runtime depth of cascading triggers	16
User-defined Types	Maximum number of attributes in a structured type	4082

Table 315. Database Manager Limits (continued)

Category	Description	Limit
Workload Manager	Maximum number of user-defined service superclasses per database	64
	Maximum number of user-defined service subclasses per service superclass	61
Open Files	Maximum number of files opened by a single Db2 instance. This includes, but not limited to, table space container files and transaction log files for all databases in the instance. In addition, the operating system might impose a lower limit.	65534

Note:

1. This maximum can be achieved by using a join in the CREATE VIEW statement. Selecting from such a view is subject to the limit of most elements in a select list.
2. The actual data for BLOB, CLOB, LONG VARCHAR, DBCLOB, and LONG VARGRAPHIC columns are not included in this count. However, information about the location of that data does take up some space in the row.
3. The numbers that are shown are architectural limits and approximations. The practical limits might be less.
4. The actual value is controlled by the **max_connections** and **max_coordagents** database manager configuration parameters.
5. This is an architectural limit. The limit on the most columns in an index key should be used as the practical limit.
6. For page size-specific values, see Table 316 on page 2136.
7. This is limited only by the longest index key, including all overhead (in bytes). As the number of index key parts increases, the maximum length of each key part decreases.
8. The maximum can be less, depending on index options.
9. If the **extended_row_sz** database configuration parameter is set to ENABLE and there are VARCHAR, VARBINARY, or VARGRAPHIC columns in the table, the maximum row size is 1,048,319 bytes, which includes all overhead.
10. Must account for columns that are generated internally by the database manager. The RANDOM_DISTRIBUTION_KEY is an example: it is created for random distribution tables that use the random by generation method.

Table 316. Database Manager Page Size-specific Limits

Description	4K page size limit	8K page size limit	16K page size limit	32K page size limit
Maximum number of table objects in a DMS table space[1]	51,971[2] 53,212[3]	53,299	53,747	54,264
Maximum number of columns in a row-organized table	500	1012	1012	2048
Maximum number of columns in a column-organized table	2048	2048	2048	2048
Maximum length of a row in a row-organized table, including all overhead	4005	8101	16,293	32,677

Table 316. Database Manager Page Size-specific Limits (continued)

Description	4K page size limit	8K page size limit	16K page size limit	32K page size limit
Maximum length of a row in a column-organized table, including all overhead	1,048,319	1,048,319	1,048,319	1,048,319
Maximum size of a table per database partition in a regular table space (in gigabytes)	64	128	256	512
Maximum size of a table per database partition in a large DMS table space (in terabytes)	8	16	32	64
Maximum length of an index key, including all overhead (in bytes)	1024	2048	4096	8192
Maximum size of an index per database partition in an SMS table space (in terabytes)	8	16	32	64
Maximum size of an index per database partition in a regular DMS table space (in gigabytes)	64	128	256	512
Maximum size of an index per database partition in a large DMS table space (in terabytes)	8	16	32	64
Maximum size of a regular DMS table space per database partition (in gigabytes)	64	128	256	512
Maximum size of a large DMS table space (in terabytes)	8	16	32	64
Maximum size of a temporary DMS table space (in terabytes)	8	16	32	64
Maximum number of elements in a select list	500[4]	1012[5]	1012[5]	2048
Maximum number of columns in a GROUP BY clause	500	1012[5]	1012[5]	2048
Maximum total length of columns in a GROUP BY clause (in bytes)	4005	8101	16,293	32,677
Maximum number of columns in an ORDER BY clause	500	1012[5]	1012[5]	2048
Maximum total length of columns in an ORDER BY clause (in bytes)	4005	8101	16,293	32,677
Maximum number of values in an insert operation	500	1012[5]	1012[5]	2048
Maximum number of SET clauses in a single update operation	500	1012[5]	1012[5]	2048
Maximum records per page for a regular table space	251	253	254	253

Table 316. Database Manager Page Size-specific Limits (continued)

Description	4K page size limit	8K page size limit	16K page size limit	32K page size limit
Maximum records per page for a large table space	287	580	1165	2335

Note:

1. Table objects include table data, indexes, LONG VARCHAR columns, LONG VARGRAPHIC columns, and LOB columns. Table objects that are in the same table space as the table data do not count extra toward the limit. However, each table object that is in a different table space than the table data does contribute one toward the limit for each table object type per table in the table space in which the table object resides.
2. When extent size is two pages.
3. When extent size is any size other than two pages.
4. In cases where the only system temporary table space is 4KB and the data overflows to the sort buffer, an error is generated. If the result set can fit into memory, there is no error.
5. May be up to 2048 for columnar tables.

Reserved schema names and reserved words

There are restrictions on the use of certain names that are required by the database manager.

In some cases, names are reserved, and cannot be used by application programs. In other cases, certain names are not recommended for use by application programs, although their use is not prevented by the database manager.

The reserved schema names are:

- SYSCAT
- SYSFUN
- SYSIBM
- SYSIBMADM
- SYSPROC
- SYSPUBLIC
- SYSSTAT

It is strongly recommended that schema names never begin with the 'SYS' prefix, because 'SYS', by convention, is used to indicate an area that is reserved by the system. No aliases, global variables, triggers, user-defined functions, or user-defined types can be placed into a schema whose name starts with 'SYS' (SQLSTATE 42939).

The DB2QP schema and the SYSTOOLS schema are set aside for utilities used by the database. It is recommended that users not explicitly define objects in these schemas, although their use is not prevented by the database manager.

It is recommended that schema names never begin with the 'Q' prefix, because on other Db2 database managers 'Q', by convention, is used to indicate an area reserved by the system.

It is also recommended that SESSION not be used as a schema name. Because declared temporary tables must be qualified by SESSION, it is possible to have an application declare a temporary table with a name that is identical to that of a persistent table, complicating the application logic. To avoid this possibility, do not use the schema SESSION except when dealing with declared temporary tables.

Keywords can be used as ordinary identifiers, except in a context where they could also be interpreted as SQL keywords. In such cases, the word must be specified as a delimited identifier. For example, COUNT cannot be used as a column name in a SELECT statement, unless it is delimited.

ISO/ANSI SQL2003 and other IBM database products include reserved words that are not enforced by Db2; however, it is recommended that these words not be used as ordinary identifiers, because it reduces portability.

For portability across the IBM database products, the following words should be considered reserved words:

```
ACTIVATE                DOUBLE                LOCALE
RESULT                            WLM
ADD                     DROP                  LOCALTIME
RESULT_SET_LOCATOR                WRITE
AFTER                   DSSIZE                LOCALTIMESTAMP
RETURN                            XMLELEMENT
ALIAS                   DYNAMIC               LOCATOR
RETURNS                           XMLEXISTS
ALL                     EACH                  LOCATORS
REVOKE                            XMLNAMESPACES
ALLOCATE                EDITPROC              LOCK
RIGHT                             YEAR
ALLOW                   ELSE                  LOCKMAX
ROLE                              YEARS
ALTER                   ELSEIF                LOCKSIZE              ROLLBACK
AND                     ENABLE                LONG                  ROUND_CEILING
ANY                     ENCODING              LOOP                  ROUND_DOWN
AS                      ENCRYPTION            MAINTAINED            ROUND_FLOOR
ASENSITIVE              END                   MATERIALIZED          ROUND_HALF_DOWN
ASSOCIATE               END-EXEC              MAXVALUE              ROUND_HALF_EVEN
ASUTIME                 ENDING                MICROSECOND           ROUND_HALF_UP
AT                      ERASE                 MICROSECONDS          ROUND_UP
ATTRIBUTES              ESCAPE                MINUTE                ROUTINE
AUDIT                   EVERY                 MINUTES               ROW
AUTHORIZATION           EXCEPT                MINVALUE              ROWNUMBER
AUX                     EXCEPTION             MODE                  ROWS
AUXILIARY               EXCLUDING             MODIFIES              ROWSET
BEFORE                  EXCLUSIVE             MONTH                 ROW_NUMBER
BEGIN                   EXECUTE               MONTHS                RRN
BETWEEN                 EXISTS                NAN                   RUN
BINARY                  EXIT                  NEW                   SAVEPOINT
BUFFERPOOL              EXPLAIN               NEW_TABLE             SCHEMA
BY                      EXTENDED              NEXTVAL               SCRATCHPAD
CACHE                   EXTERNAL              NO                    SCROLL
CALL                    EXTRACT               NOCACHE               SEARCH
CALLED                  FENCED                NOCYCLE               SECOND
CAPTURE                 FETCH                 NODENAME              SECONDS
CARDINALITY             FIELDPROC             NODENUMBER            SECQTY
CASCADED                FILE                  NOMAXVALUE            SECURITY
CASE                    FINAL                 NOMINVALUE            SELECT
CAST                    FIRST[1]              NONE                  SENSITIVE
CCSID                   FOR                   NOORDER               SEQUENCE
CHAR                    FOREIGN               NORMALIZED            SESSION
CHARACTER               FREE                  NOT[2]                SESSION_USER
CHECK                   FROM                  NOTNULL               SET
CLONE                   FULL                  NULL                  SIGNAL
CLOSE                   FUNCTION              NULLS                 SIMPLE
CLUSTER                 GENERAL               NUMPARTS              SNAN
COLLECTION              GENERATED             OBID                  SOME
COLLID                  GET                   OF                    SOURCE
COLUMN                  GLOBAL                OFF                   SPECIFIC
COMMENT                 GO                    OFFSET                SQL
COMMIT                  GOTO                  OLD                   SQLID
CONCAT                  GRANT                 OLD_TABLE             STACKED
CONDITION               GRAPHIC               ON                    STANDARD
CONNECT                 GROUP                 OPEN                  START
CONNECTION              HANDLER               OPTIMIZATION          STARTING
CONSTRAINT              HASH                  OPTIMIZE              STATEMENT
CONTAINS                HASHED_VALUE          OPTION                STATIC
```

[1] As of Db2 Version 11.1.0.0, **FIRST** is a SQL keyword in some expression contexts, for example, within an OLAP specification. This means **FIRST**, when used as an identifier, must be delimited when used within these contexts.

[2] As of Db2 Version 11.1.1.1, **NOT** is a valid operator in certain contexts, for example, within a select-list. This means **NOT**, when used as an identifier, must be delimited when used within these contexts.

```
CONTINUE                HAVING              OR                  STATMENT
COUNT                   HINT                ORDER               STAY
COUNT_BIG               HOLD                OUT                 STOGROUP
CREATE                  HOUR                OUTER               STORES
CROSS                   HOURS               OVER                STYLE
CURRENT                 IDENTITY            OVERRIDING          SUBSTRING
CURRENT_DATE            IF                  PACKAGE             SUMMARY
CURRENT_LC_CTYPE        IMMEDIATE           PADDED              SYNONYM
CURRENT_PATH            IMPORT              PAGESIZE            SYSFUN
CURRENT_SCHEMA          IN                  PARAMETER           SYSIBM
CURRENT_SERVER          INCLUDING           PART                SYSPROC
CURRENT_TIME            INCLUSIVE           PARTITION           SYSTEM
CURRENT_TIMESTAMP       INCREMENT           PARTITIONED         SYSTEM_USER
CURRENT_TIMEZONE        INDEX               PARTITIONING        TABLE
CURRENT_USER            INDICATOR           PARTITIONS          TABLESPACE
CURSOR                  INDICATORS          PASSWORD            THEN
CYCLE                   INF                 PATH                TIME
DATA                    INFINITY            PERCENT             TIMESTAMP
DATABASE                INHERIT             PIECESIZE           TO
DATAPARTITIONNAME       INNER               PLAN                TRANSACTION
DATAPARTITIONNUM        INOUT               POSITION            TRIGGER
DATE                    INSENSITIVE         PRECISION           TRIM
DAY                     INSERT              PREPARE             TRUNCATE
DAYS                    INTEGRITY           PREVVAL             TYPE
DB2GENERAL              INTERSECT           PRIMARY             UNDO
DB2GENRL                INTO                PRIQTY              UNION
DB2SQL                  IS                  PRIVILEGES          UNIQUE
DBINFO                  ISNULL              PROCEDURE           UNTIL
DBPARTITIONNAME         ISOBID              PROGRAM             UPDATE
DBPARTITIONNUM          ISOLATION           PSID                USAGE
DEALLOCATE              ITERATE             PUBLIC              USER
DECLARE                 JAR                 QUERY               USING
DEFAULT                 JAVA                QUERYNO             VALIDPROC
DEFAULTS                JOIN                RANGE               VALUE
DEFINITION              KEEP                RANK                VALUES
DELETE                  KEY                 READ                VARIABLE
DENSERANK               LABEL               READS               VARIANT
DENSE_RANK              LANGUAGE            RECOVERY            VCAT
DESCRIBE                LAST[3]             REFERENCES          VERSION
DESCRIPTOR              LATERAL             REFERENCING         VIEW
DETERMINISTIC           LC_CTYPE            REFRESH             VOLATILE
DIAGNOSTICS             LEAVE               RELEASE             VOLUMES
DISABLE                 LEFT                RENAME              WHEN
DISALLOW                LIKE                REPEAT              WHENEVER
DISCONNECT              LIMIT               RESET               WHERE
DISTINCT                LINKTYPE            RESIGNAL            WHILE
DO                      LOCAL               RESTART             WITH
DOCUMENT                LOCALDATE           RESTRICT            WITHOUT
```

ISO/ANSI SQL2003 reserved words that are not in the previous list:

```
ABS                             GROUPING        REGR_INTERCEPT
ARE                             INT             REGR_R2
ARRAY                           INTEGER         REGR_SLOPE
ASYMMETRIC                      INTERSECTION    REGR_SXX
ATOMIC                          INTERVAL        REGR_SXY
AVG                             LARGE           REGR_SYY
BIGINT                          LEADING         ROLLUP
BLOB                            LN              SCOPE
BOOLEAN                         LOWER           SIMILAR
BOTH                            MATCH           SMALLINT
CEIL                            MAX             SPECIFICTYPE
CEILING                         MEMBER          SQLEXCEPTION
CHAR_LENGTH                     MERGE           SQLSTATE
CHARACTER_LENGTH                METHOD          SQLWARNING
CLOB                            MIN             SQRT
COALESCE                        MOD             STDDEV_POP
COLLATE                         MODULE          STDDEV_SAMP
COLLECT                         MULTISET        SUBMULTISET
CONVERT                         NATIONAL        SUM
CORR                            NATURAL         SYMMETRIC
CORRESPONDING                   NCHAR           TABLESAMPLE
COVAR_POP                       NCLOB           TIMEZONE_HOUR
```

[3] As of Db2 Version 11.1.0.0, **LAST** is a SQL keywords in some expression contexts, for example, within an OLAP specification. This means **LAST**, when used as an identifier, must be delimited when used within these contexts.

```
COVAR_SAMP                          NORMALIZE          TIMEZONE_MINUTE
CUBE                                NULLIF             TRAILING
CUME_DIST                           NUMERIC            TRANSLATE
CURRENT_DEFAULT_TRANSFORM_GROUP     OCTET_LENGTH       TRANSLATION
CURRENT_ROLE                        ONLY               TREAT
CURRENT_TRANSFORM_GROUP_FOR_TYPE    OVERLAPS           TRUE
DEC                                 OVERLAY            UESCAPE
DECIMAL                             PERCENT_RANK       UNKNOWN
DEREF                               PERCENTILE_CONT    UNNEST
ELEMENT                             PERCENTILE_DISC    UPPER
EXEC                                POWER              VAR_POP
EXP                                 REAL               VAR_SAMP
FALSE                               RECURSIVE          VARBINARY
FILTER                              REF                VARCHAR
FLOAT                               REGR_AVGX          VARYING
FLOOR                               REGR_AVGY          WIDTH_BUCKET
FUSION                              REGR_COUNT         WINDOW
                                                       WITHIN
```

Communications areas, descriptor areas, and exception tables

SQLCA (SQL communications area)

An SQLCA is a collection of variables that is updated at the end of the execution of every SQL statement.

A program that contains executable SQL statements and is precompiled with option LANGLEVEL SAA1 (the default) or MIA must provide exactly one SQLCA, though more than one SQLCA is possible by having one SQLCA per thread in a multi-threaded application.

When a program is precompiled with option LANGLEVEL SQL92E, an SQLCODE or SQLSTATE variable may be declared in the SQL declare section or an SQLCODE variable can be declared somewhere in the program.

An SQLCA should not be provided when using LANGLEVEL SQL92E. The SQL INCLUDE statement can be used to provide the declaration of the SQLCA in all languages but REXX. The SQLCA is automatically provided in REXX.

To display the SQLCA after each command executed through the command line processor, issue the command db2 -a. The SQLCA is then provided as part of the output for subsequent commands. The SQLCA is also dumped in the **db2diag** log file.

SQLCA field descriptions

Table 317. Fields of the SQLCA. The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement.

Name	Data Type	Field Values
sqlcaid	CHAR(8)	An "eye catcher" for storage dumps containing "SQLCA". The sixth byte is "L" if line number information is returned from parsing an SQL routine, SQL trigger, or dynamic compound SQL statement. The sixth byte is "M" if the line number and object ID information is returned from executing a compiled SQL routine, compiled SQL trigger, or dynamic compound SQL (compiled) statement.
sqlcabc	INTEGER	Contains the length of the SQLCA, 136.

Table 317. Fields of the SQLCA. The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement. *(continued)*

Name	Data Type	Field Values
sqlcode	INTEGER	Contains the SQL return code: **0** Successful execution (although one or more SQLWARN indicators may be set). **positive** Successful execution, but with a warning condition. **negative** Error condition.
sqlerrml	SMALLINT	Length indicator for *sqlerrmc*, in the range 0 through 70. 0 means that the value of *sqlerrmc* is not relevant.
sqlerrmc	VARCHAR (70)	Contains one or more tokens, separated by X'FF', which are substituted for variables in the descriptions of error conditions. This field is also used when a successful connection is completed. When a NOT ATOMIC compound SQL statement is issued, it can contain information about seven or fewer errors. The last token might be followed by X'FF'. The *sqlerrml* value will include any trailing X'FF'.
sqlerrp	CHAR(8)	Starting with V11.1, the format of the SQLERRP field is changed to **SQL*vvrrmm***, where: • ***vv*** represents the version number • ***rr*** represents the release number • ***m*** represents the modification value The following examples illustrate the relationship between the product signature and the new token in the SQLERRP field. All subsequent fix packs for a given Mod Pack return the same SQLERRP value. **Product signature** / **SQLERRP token** Db2 10.5.0.7 — SQL10057 Db2 11.0.0.0 — SQL11010 Db2 11.1.1.1 — SQL11011 Db2 11.1.2.0 — SQL11012
sqlerrd	ARRAY	Six INTEGER variables that provide diagnostic information. These values are generally empty if there are no errors, except for sqlerrd(6) from a partitioned database.
sqlerrd(1)	INTEGER	If connection is invoked and successful, contains the maximum expected difference in length of mixed character data (CHAR data types) when converted to the database code page from the application code page. A value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction. On successful return from an SQL procedure, contains the return status value from the SQL procedure.

Table 317. Fields of the SQLCA. The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement. *(continued)*

Name	Data Type	Field Values
sqlerrd(2)	INTEGER	If connection is invoked and successful, contains the maximum expected difference in length of mixed character data (CHAR data types) when converted to the application code page from the database code page. A value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction. If the SQLCA results from a NOT ATOMIC compound SQL statement that encountered one or more errors, the value is set to the number of statements that failed.
sqlerrd(3)	INTEGER	If PREPARE is invoked and successful, contains an estimate of the number of rows that will be returned. After INSERT, UPDATE, DELETE, or MERGE, contains the actual number of rows that qualified for the operation. For a TRUNCATE statement, the value will be -1. If compound SQL is invoked, contains an accumulation of all sub-statement rows. If CONNECT is invoked, contains 1 if the database can be updated, or 2 if the database is read only. If the OPEN statement is invoked, and the cursor contains SQL data change statements, this field contains the sum of the number of rows that qualified for the embedded insert, update, delete, or merge operations. If an error is encountered during the compilation of an SQL routine, trigger, or dynamic compound SQL (inlined or compiled) statement, sqlerrd(3) contains the line number where the error was encountered. The sixth byte of sqlcaid must be "L" for this entry to be a valid line number. If an error is encountered during the execution of a compiled SQL routine, trigger, or dynamic SQL (compiled) statement, sqlerrd(3) contains the line number where the error was raised. The sixth byte of sqlcaid must be "M" for this to be a valid line number.
sqlerrd(4)	INTEGER	If PREPARE is invoked and successful , contains a relative cost estimate of the resources required to process the statement. If compound SQL is invoked, contains a count of the number of successful sub-statements. If CONNECT is invoked, contains 0 for a one-phase commit from a client which is not at the latest level; 1 for a one-phase commit; 2 for a one-phase, read-only commit; and 3 for a two-phase commit. If an error is encountered during the execution of a compiled SQL routine or trigger, sqlerrd(4) contains an integer number that uniquely identifies the routine or trigger within which the error was raised. The sixth byte of sqlcaid must be "M" for this entry to be a valid line number

Table 317. Fields of the SQLCA. The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement. *(continued)*

Name	Data Type	Field Values
sqlerrd(5)	INTEGER	Contains the total number of rows deleted, inserted, or updated as a result of both: • The enforcement of constraints after a successful delete operation • The processing of triggered SQL statements from activated inlined triggers If compound SQL is invoked, contains an accumulation of the number of such rows for all sub-statements. In some cases, when an error is encountered, this field contains a negative value that is an internal error pointer. If CONNECT is invoked, contains an authentication type value: • 0 for server authentication • 1 for client authentication • 2 for authentication using Db2 Connect • 4 for SERVER_ENCRYPT authentication • 5 for authentication using Db2 Connect with encryption; • 7 for KERBEROS authentication • 9 for GSSPLUGIN authentication • 11 for DATA_ENCRYPT authentication • 255 for unspecified authentication. **Important:** The DATA_ENCRYPT authentication type is deprecated and might be removed in a future release. To encrypt data in-transit between clients and Db2 databases, we recommend that you use the Db2 database system support of Transport Layer Security (TLS). For more information, see Encryption of data in transit
sqlerrd(6)	INTEGER	For a partitioned database, contains the partition number of the database partition that encountered the error or warning. If no errors or warnings were encountered, this field contains the partition number of the coordinator partition. The number in this field is the same as that specified for the database partition in the db2nodes.cfg file.
sqlwarn	Array	A set of warning indicators, each containing a blank or W. If compound SQL is invoked, contains an accumulation of the warning indicators set for all sub-statements.
sqlwarn0	CHAR(1)	Blank if all other indicators are blank; contains "W" if at least one other indicator is not blank.
sqlwarn1	CHAR(1)	Contains "W" if the value of a string column was truncated when assigned to a host variable. Contains "N" if the null terminator was truncated. Contains "A" if the CONNECT or ATTACH is successful, and the authorization name for the connection is longer than 8 bytes. Contains "P" if the PREPARE statement relative cost estimate stored in sqlerrd(4) exceeded the value that could be stored in an INTEGER or was less than 1, and either the CURRENT EXPLAIN MODE or the CURRENT EXPLAIN SNAPSHOT special register is set to a value other than NO.

Table 317. Fields of the SQLCA. The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement. *(continued)*

Name	Data Type	Field Values
sqlwarn2	CHAR(1)	Contains "W" if null values were eliminated from the argument of an aggregate function. [a] If CONNECT is invoked and successful, contains "D" if the database is in quiesce state, or "I" if the instance is in quiesce state.
sqlwarn3	CHAR(1)	Contains "W" if the number of columns is not equal to the number of host variables. Contains "Z" if the number of result set locators specified on the ASSOCIATE LOCATORS statement is less than the number of result sets returned by a procedure.
sqlwarn4	CHAR(1)	Contains "W" if a prepared UPDATE or DELETE statement does not include a WHERE clause.
sqlwarn5	CHAR(1)	Contains "E" if an error was tolerated during SQL statement execution.
sqlwarn6	CHAR(1)	Contains "W" if the result of a date calculation was adjusted to avoid an impossible date.
sqlwarn7	CHAR(1)	If CONNECT is invoked and successful, contains a "B" if the server is **BigSQL**, or "D" if the server is **Db2 Warehouse on Cloud**.
sqlwarn8	CHAR(1)	Contains "W" if a character that could not be converted was replaced with a substitution character. Contains "Y" if there was an unsuccessful attempt to establish a trusted connection.
sqlwarn9	CHAR(1)	Contains "W" if arithmetic expressions with errors were ignored during aggregate function processing.
sqlwarn10	CHAR(1)	Contains "W" if there was a conversion error when converting a character data value in one of the fields in the SQLCA.
sqlstate	CHAR(5)	A return code that indicates the outcome of the most recently executed SQL statement.

[a] Some functions may not set SQLWARN2 to W, even though null values were eliminated, because the result was not dependent on the elimination of null values.

Error reporting

The order of error reporting is:

1. Severe error conditions are always reported. When a severe error is reported, there are no additions to the SQLCA.
2. If no severe error occurs, a deadlock error takes precedence over other errors.
3. For all other errors, the SQLCA for the first negative SQL code is returned.
4. If no negative SQL codes are detected, the SQLCA for the first warning (that is, positive SQL code) is returned.

   In a partitioned database system, the exception to this rule occurs if a data manipulation operation is invoked against a table that is empty on one database partition, but has data on other database partitions. SQLCODE +100 is only returned to the application if agents from all database partitions return SQL0100W, either because the table is empty on all database partitions, or there are no more rows that satisfy the WHERE clause in an UPDATE statement.

SQLCA usage in partitioned database systems

In partitioned database systems, one SQL statement may be executed by a number of agents on different database partitions, and each agent may return a different SQLCA for different errors or warnings. The coordinator agent also has its own SQLCA.

To provide a consistent view for applications, all SQLCA values are merged into one structure, and SQLCA fields indicate global counts, such that:

- For all errors and warnings, the *sqlwarn* field contains the warning flags received from all agents.
- Values in the *sqlerrd* fields indicating row counts are accumulations from all agents.

Note: SQLSTATE 09000 might not be returned every time an error occurs during the processing of a triggered SQL statement.

SQLDA (SQL descriptor area)

An SQLDA is a collection of variables that is required for execution of the SQL DESCRIBE statement.

The SQLDA variables are options that can be used by the PREPARE, OPEN, FETCH, and EXECUTE statements. An SQLDA communicates with dynamic SQL; it can be used in a DESCRIBE statement, modified with the addresses of host variables, and then reused in a FETCH or EXECUTE statement.

SQLDAs are supported for all languages, but predefined declarations are provided only for C, REXX, FORTRAN, and COBOL.

The meaning of the information in an SQLDA depends on its use. In PREPARE and DESCRIBE, an SQLDA provides information to an application program about a prepared statement. In OPEN, EXECUTE, and FETCH, an SQLDA describes host variables.

In DESCRIBE and PREPARE, if any one of the columns being described is either a LOB type (LOB locators and file reference variables do not require doubled SQLDAs), reference type, or a user-defined type, the number of SQLVAR entries for the entire SQLDA will be doubled. For example:

- When describing a table with 3 VARCHAR columns and 1 INTEGER column, there will be 4 SQLVAR entries
- When describing a table with 2 VARCHAR columns, 1 CLOB column, and 1 integer column, there will be 8 SQLVAR entries

In EXECUTE, FETCH, and OPEN, if any one of the variables being described is a LOB type (LOB locators and file reference variables do not require doubled SQLDAs) or a structured type, the number of SQLVAR entries for the entire SQLDA must be doubled. (Distinct types and reference types are not relevant in these cases, because the additional information in the double entries is not required by the database. Array, cursor and row types are not supported as SQLDA variables in EXECUTE, FETCH and OPEN statements.)

SQLDA field descriptions

An SQLDA consists of four variables followed by an arbitrary number of occurrences of a sequence of variables collectively named SQLVAR. In OPEN, FETCH, and EXECUTE, each occurrence of SQLVAR describes a host variable. In DESCRIBE and PREPARE, each occurrence of SQLVAR describes a column of a result table or a parameter marker. There are two types of SQLVAR entries:

- **Base SQLVARs:** These entries are always present. They contain the base information about the column, parameter marker, or host variable such as data type code, length attribute, column name, host variable address, and indicator variable address.
- **Secondary SQLVARs:** These entries are only present if the number of SQLVAR entries is doubled as per the rules outlined previously. For user-defined types (excluding reference types), they contain the user-defined type name. For reference types, they contain the target type of the reference. For LOBs, they contain the length attribute of the host variable and a pointer to the buffer that contains the actual length. (The distinct type and LOB information does not overlap, so distinct types can be based on LOBs

without forcing the number of SQLVAR entries on a DESCRIBE to be tripled.) If locators or file reference variables are used to represent LOBs, these entries are not necessary.

In SQLDAs that contain both types of entries, the base SQLVARs are in a block before the block of secondary SQLVARs. In each, the number of entries is equal to the value in SQLD (even though many of the secondary SQLVAR entries may be unused).

The circumstances under which the SQLVAR entries are set by DESCRIBE is detailed in "Effect of DESCRIBE on the SQLDA" on page 2151.

Fields in the SQLDA header

Table 318. Fields in the SQLDA Header

C Name	SQL Data Type	Usage in DESCRIBE and PREPARE (set by the database manager except for SQLN)	Usage in FETCH, OPEN, and EXECUTE (set by the application before executing the statement)
sqldaid	CHAR(8)	The seventh byte of this field is a flag byte named SQLDOUBLED. The database manager sets SQLDOUBLED to the character "2" if two SQLVAR entries have been created for each column; otherwise it is set to a blank (X'20' in ASCII, X'40' in EBCDIC). See "Effect of DESCRIBE on the SQLDA" on page 2151 for details on when SQLDOUBLED is set.	The seventh byte of this field is used when the number of SQLVARs is doubled. It is named SQLDOUBLED. If any of the host variables being described is a structured type, BLOB, CLOB, or DBCLOB, the seventh byte must be set to the character "2"; otherwise it can be set to any character but the use of a blank is recommended.
sqldabc	INTEGER	For 32 bit, the length of the SQLDA, equal to SQLN*44+16. For 64 bit, the length of the SQLDA, equal to SQLN*56+16	For 32 bit, the length of the SQLDA, >= to SQLN*44+16. For 64 bit, the length of the SQLDA, >= to SQLN*56+16.
sqln	SMALLINT	Unchanged by the database manager. Must be set to a value greater than or equal to zero before the DESCRIBE statement is executed. Indicates the total number of occurrences of SQLVAR.	Total number of occurrences of SQLVAR provided in the SQLDA. SQLN must be set to a value greater than or equal to zero.
sqld	SMALLINT	Set by the database manager to the number of columns in the result table or to the number of parameter markers.	The number of host variables described by occurrences of SQLVAR.

Fields in an occurrence of a base SQLVAR

Table 319. Fields in a Base SQLVAR

Name	Data Type	Usage in DESCRIBE and PREPARE	Usage in FETCH, OPEN, and EXECUTE
sqltype	SMALLINT	Indicates the data type of the column or parameter marker, and whether it can contain nulls. (Parameter markers are always considered nullable.) Table 321 on page 2152 lists the allowable values and their meanings. Note that for a distinct, array, cursor, row, or reference type, the data type of the base type is placed into this field. For a structured type, the data type of the result of the FROM SQL transform function of the transform group (based on the CURRENT DEFAULT TRANSFORM GROUP special register) for the type is placed into this field. There is no indication in the base SQLVAR that it is part of the description of a user-defined type or reference type.	Same for host variable. Host variables for datetime values must be character string variables. For FETCH, a datetime type code means a fixed-length character string. If sqltype is an even number value, the sqlind field is ignored.
sqllen	SMALLINT	The length attribute of the column or parameter marker. For datetime columns and parameter markers, the length of the string representation of the values. See Table 321 on page 2152. Note that the value is set to 0 for large object strings (even for those whose length attribute is small enough to fit into a two byte integer).	The length attribute of the host variable. See Table 321 on page 2152. Note that the value is ignored by the database manager for CLOB, DBCLOB, and BLOB columns. The len.sqllonglen field in the Secondary SQLVAR is used instead.
sqldata	pointer	For string SQLVARS, sqldata contains the code page. For character-string SQLVARs where the column is defined with the FOR BIT DATA attribute, sqldata contains 0. For other character-string SQLVARS, sqldata contains either the SBCS code page for SBCS data, or the SBCS code page associated with the composite MBCS code page for MBCS data. For Japanese EUC, Traditional Chinese EUC, and Unicode UTF-8 character-string SQLVARS, sqldata contains 954, 964, and 1208 respectively. For all other column types, sqldata is undefined.	Contains the address of the host variable (where the fetched data will be stored).

Table 319. Fields in a Base SQLVAR (continued)

Name	Data Type	Usage in DESCRIBE and PREPARE	Usage in FETCH, OPEN, and EXECUTE
sqlind	pointer	For character-string SQLVARS, sqlind contains 0, except for MBCS data, when sqlind contains the DBCS code page associated with the composite MBCS code page. For all other types, sqlind is undefined.	Contains the address of an associated indicator variable, if there is one; otherwise, not used. If sqltype is an even number value, the sqlind field is ignored.
sqlname	VARCHAR (30)	Contains the unqualified name of the column or parameter marker. For columns and parameter markers that have a system-generated name, the thirtieth byte is set to X'FF'. For column names specified by the AS clause, this byte is X'00'.	When connecting to a host database, sqlname can be set to indicate a FOR BIT DATA string as follows: • The sixth byte of the SQLDAID in the SQLDA header is set to "+" • The length of sqlname is 8 • The first two bytes of sqlname are X'0000' • The third and fourth bytes of sqlname are X'0000' • The remaining four bytes of sqlname are reserved and should be set to X'00000000' When working with XML data, sqlname can be set to indicate an XML subtype as follows: • The length of sqlname is 8 • The first two bytes of sqlname are X'0000' • The third and fourth bytes of sqlname are X'0000' • The fifth byte of sqlname is X'01' • The remaining three bytes of sqlname are reserved and should be set to X'000000'

Fields in an occurrence of a secondary SQLVAR

Table 320. Fields in a Secondary SQLVAR

Name	Data Type	Usage in DESCRIBE and PREPARE	Usage in FETCH, OPEN, and EXECUTE
len.sqllonglen	INTEGER	The length attribute of a BLOB, CLOB, or DBCLOB column or parameter marker.	The length attribute of a BLOB, CLOB, or DBCLOB host variable. The database manager ignores the SQLLEN field in the Base SQLVAR for the data types. The length attribute stores the number of bytes for a BLOB or CLOB, and the number of double-byte characters for a DBCLOB.

Table 320. Fields in a Secondary SQLVAR (continued)

Name	Data Type	Usage in DESCRIBE and PREPARE	Usage in FETCH, OPEN, and EXECUTE
reserve2	CHAR(3) for 32 bit, and CHAR(11) for 64 bit.	Not used.	Not used.
sqlflag4	CHAR(1)	The value is X'01' if the SQLVAR represents a reference type with a target type named in sqldatatype_name. The value is X'12' if the SQLVAR represents a structured type, with the user-defined type name in sqldatatype_name. Otherwise, the value is X'00'.	Set to X'01' if the SQLVAR represents a reference type with a target type named in sqldatatype_name. Set to X'12' if the SQLVAR represents a structured type, with the user-defined type name in sqldatatype_name. Otherwise, the value is X'00'.
sqldatalen	pointer	Not used.	Used for BLOB, CLOB, and DBCLOB host variables only. If this field is the null value, then the actual length (in double-byte characters) should be stored in the 4 bytes immediately before the start of the data and SQLDATA should point to the first byte of the field length. If this field is not the null value, it contains a pointer to a 4 byte long buffer that contains the actual length *in bytes* (even for DBCLOB) of the data in the buffer pointed to from the SQLDATA field in the matching base SQLVAR. Note that, whether or not this field is used, the len.sqllonglen field must be set.
sqldatatype_name	VARCHAR(27)	For a user-defined type, the database manager sets this to the fully qualified user-defined type name.[1] For a reference type, the database manager sets this to the fully qualified type name of the target type of the reference.	For structured types, set to the fully qualified user-defined type name in the format indicated in the table note.[1]
reserved	CHAR(3)	Not used.	Not used.

Table 320. Fields in a Secondary SQLVAR (continued)

Name	Data Type	Usage in DESCRIBE and PREPARE	Usage in FETCH, OPEN, and EXECUTE

[1] The first 8 bytes contain the schema name of the type (extended to the right with spaces, if necessary). Byte 9 contains a dot (.). Bytes 10 to 27 contain the low order portion of the type name, which is *not* extended to the right with spaces.

Note that, although the prime purpose of this field is for the name of user-defined types, the field is also set for IBM predefined data types. In this case, the schema name is SYSIBM, and the low order portion of the name is the name stored in the TYPENAME column of the DATATYPES catalog view. For example:

```
type name        length   sqldatatype_name
---------        ------   ----------------
A.B              10       A       .B
INTEGER          16       SYSIBM  .INTEGER
"Frank's".SMINT  13       Frank's .SMINT
MY."type  "      15       MY      .type
```

Effect of DESCRIBE on the SQLDA

For a DESCRIBE OUTPUT or PREPARE OUTPUT INTO statement, the database manager always sets SQLD to the number of columns in the result set, or the number of output parameter markers. For a DESCRIBE INPUT or PREPARE INPUT INTO statement, the database manager always sets SQLD to the number of input parameter markers in the statement. Note that a parameter marker that corresponds to an INOUT parameter in a CALL statement is described in both the input and output descriptors.

The SQLVARs in the SQLDA are set in the following cases:

- SQLN >= SQLD and no entry is either a LOB, user-defined type or reference type

  The first SQLD SQLVAR entries are set and SQLDOUBLED is set to blank.
- SQLN >= 2*SQLD and at least one entry is a LOB, user-defined type or reference type

  Two times SQLD SQLVAR entries are set, and SQLDOUBLED is set to "2".
- SQLD <= SQLN < 2*SQLD and at least one entry is a distinct, array, cursor, row, or reference type, but there are no LOB entries or structured type entries

  The first SQLD SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +237 (SQLSTATE 01594) is issued.

The SQLVARs in the SQLDA are NOT set (requiring allocation of additional space and another DESCRIBE) in the following cases:

- SQLN < SQLD and no entry is either a LOB, user-defined type or reference type

  No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +236 (SQLSTATE 01005) is issued.

  Allocate SQLD SQLVARs for a successful DESCRIBE.
- SQLN < SQLD and at least one entry is a distinct, array, cursor, row, or reference type, but there are no LOB entries or structured type entries

  No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +239 (SQLSTATE 01005) is issued.

  Allocate 2*SQLD SQLVARs for a successful DESCRIBE including the names of the distinct, array, cursor, and row types and target types of reference types.
- SQLN < 2*SQLD and at least one entry is a LOB or a structured type

  No SQLVAR entries are set and SQLDOUBLED is set to blank. A warning SQLCODE +238 (SQLSTATE 01005) is issued (regardless of the setting of the SQLWARN bind option).

  Allocate 2*SQLD SQLVARs for a successful DESCRIBE.

References in the previous lists to LOB entries include distinct type entries whose source type is a LOB type.

The SQLWARN option of the BIND or PREP command is used to control whether the DESCRIBE (or PREPARE INTO) will return the warning SQLCODEs +236, +237, +239. It is recommended that your application code always consider that these SQLCODEs could be returned. The warning SQLCODE +238 is always returned when there are LOB or structured type entries in the select list and there are insufficient SQLVARs in the SQLDA. This is the only way the application can know that the number of SQLVARs must be doubled because of a LOB or structured type entry in the result set.

If a structured type entry is being described, but no FROM SQL transform is defined (either because no TRANSFORM GROUP was specified using the CURRENT DEFAULT TRANSFORM GROUP special register (SQLSTATE 42741) or because the name group does not have a FROM SQL transform function defined (SQLSTATE 42744)), the DESCRIBE will return an error. This error is the same error returned for a DESCRIBE of a table with a structured type entry.

If the database manager returns identifiers that are longer than those that can be stored in the SQLDA, the identifier is truncated and a warning is returned (SQLSTATE 01665); however, when the name of a structured type is truncated, an error is returned (SQLSTATE 42622). For details on identifier length limitations, see "SQL and XQuery limits" .

SQLTYPE and SQLLEN

Table 321 on page 2152 shows the values that may appear in the SQLTYPE and SQLLEN fields of the SQLDA. In DESCRIBE and PREPARE INTO, an even value of SQLTYPE means that the column does not allow nulls, and an odd value means the column does allow nulls. In FETCH, OPEN, and EXECUTE, an even value of SQLTYPE means that no indicator variable is provided, and an odd value means that SQLIND contains the address of an indicator variable.

Table 321. SQLTYPE and SQLLEN values for DESCRIBE, FETCH, OPEN, and EXECUTE

SQLTYPE	Column data type for DESCRIBE and PREPARE INTO	SQLLEN for DESCRIBE and PREPARE INTO	Host variable data type for FETCH, OPEN, and EXECUTE	SQLLEN for FETCH, OPEN, and EXECUTE
384/385	date	10	fixed-length character string representation of a date	length attribute of the host variable
388/389	time	8	fixed-length character string representation of a time	length attribute of the host variable
392/393	timestamp	19 for TIMESTAMP(0) otherwise 20+*p* for TIMESTAMP(*p*)	fixed-length character string representation of a timestamp	length attribute of the host variable
400/401	N/A	N/A	NULL-terminated graphic string	length attribute of the host variable
404/405	BLOB	0 *	BLOB	Not used. *
408/409	CLOB	0 *	CLOB	Not used. *
412/413	DBCLOB	0 *	DBCLOB	Not used. *
448/449	varying-length character string	length attribute of the column	varying-length character string	length attribute of the host variable

Table 321. SQLTYPE and SQLLEN values for DESCRIBE, FETCH, OPEN, and EXECUTE (continued)

SQLTYPE	Column data type for DESCRIBE and PREPARE INTO	SQLLEN for DESCRIBE and PREPARE INTO	Host variable data type for FETCH, OPEN, and EXECUTE	SQLLEN for FETCH, OPEN, and EXECUTE
452/453	fixed-length character string	length attribute of the column	fixed-length character string	length attribute of the host variable
456/457	long varying-length character string	length attribute of the column	long varying-length character string	length attribute of the host variable
460/461	not applicable	not applicable	NULL-terminated character string	length attribute of the host variable
464/465	varying-length graphic string	length attribute of the column	varying-length graphic string	length attribute of the host variable
468/469	fixed-length graphic string	length attribute of the column	fixed-length graphic string	length attribute of the host variable
472/473	long varying-length graphic string	length attribute of the column	long graphic string	length attribute of the host variable
480/481	floating-point	8 for double precision, 4 for single precision	floating-point	8 for double precision, 4 for single precision
484/485	packed decimal	precision in byte 1; scale in byte 2	packed decimal	precision in byte 1; scale in byte 2
492/493	big integer	8	big integer	8
496/497	large integer	4	large integer	4
500/501	small integer	2	small integer	2
908/909	varying-length binary string	length attribute of the column	varying-length binary string	length attribute of the host variable
912/913	fixed-length binary string	length attribute of the column	fixed-length binary string	length attribute of the host variable
916/917	not applicable	not applicable	BLOB file reference variable	267
920/921	not applicable	not applicable	CLOB file reference variable	267
924/925	not applicable	not applicable	DBCLOB file reference variable.	267
960/961	not applicable	not applicable	BLOB locator	4
964/965	not applicable	not applicable	CLOB locator	4
968/969	not applicable	not applicable	DBCLOB locator	4
988/989	XML	0	not applicable; use an XML AS <string or binary LOB type> host variable instead	not used

Table 321. SQLTYPE and SQLLEN values for DESCRIBE, FETCH, OPEN, and EXECUTE (continued)

SQLTYPE	Column data type for DESCRIBE and PREPARE INTO	SQLLEN for DESCRIBE and PREPARE INTO	Host variable data type for FETCH, OPEN, and EXECUTE	SQLLEN for FETCH, OPEN, and EXECUTE
996	decimal floating-point	8 for DECFLOAT(16), 16 for DECFLOAT(34)	decimal floating-point	8 for DECFLOAT(16), 16 for DECFLOAT(34)
2440/2441	row	not applicable	row	not used
2440/2441	cursor	not applicable	row	not used

Note:

- The len.sqllonglen field in the secondary SQLVAR contains the length attribute of the column.
- The SQLTYPE has changed from previous versions for portability reasons. The values from the previous version (see previous version SQL Reference) continue to be supported.

Unrecognized and unsupported SQLTYPEs

The values that appear in the SQLTYPE field of the SQLDA are dependent on the level of data type support available at the sender as well as at the receiver of the data. This is particularly important as new data types are added to the product.

New data types may or may not be supported by the sender or receiver of the data and may or may not even be recognized by the sender or receiver of the data. Depending on the situation, the new data type may be returned, or a compatible data type agreed upon by both the sender and receiver of the data may be returned or an error may result.

When the sender and receiver agree to use a compatible data type, the following table indicates the mapping that will take place. This mapping will take place when at least one of the sender or the receiver does not support the data type provided. The unsupported data type can be provided by either the application or the database manager.

Data Type	Compatible Data Type
BIGINT	DECIMAL(19, 0)
ROWID	VARCHAR(40) FOR BIT DATA

Note that no indication is given in the SQLDA that the data type is substituted.

Packed decimal numbers

Packed decimal numbers are stored in a variation of Binary Coded Decimal (BCD) notation. In BCD, each nybble (four bits) represents one decimal digit. For example, 0001 0111 1001 represents 179. Therefore, read a packed decimal value nybble by nybble. Store the value in bytes and then read those bytes in hexadecimal representation to return to decimal. For example, 0001 0111 1001 becomes 00000001 01111001 in binary representation. By reading this number as hexadecimal, it becomes 0179.

The decimal point is determined by the scale. In the case of a DEC(12,5) column, for example, the rightmost 5 digits are to the right of the decimal point.

Sign is indicated by a nybble to the right of the nybbles representing the digits. A positive or negative sign is indicated as follows:

Table 322. Values for Sign Indicator of a Packed Decimal Number

Sign	Binary representation	Decimal representation	Hexadecimal representation
Positive (+)	1100	12	C
Negative (-)	1101	13	D

In summary:

- To store any value, allocate *p*/2+1 bytes, where *p* is precision.
- Assign the nybbles from left to right to represent the value. If a number has an even precision, a leading zero nybble is added. This assignment includes leading (insignificant) and trailing (significant) zero digits.
- The sign nybble will be the second nybble of the last byte.

For example:

Column	Value	Nybbles in Hexadecimal Grouped by Bytes
DEC(8,3)	6574.23	00 65 74 23 0C
DEC(6,2)	-334.02	00 33 40 2D
DEC(7,5)	5.2323	05 23 23 0C
DEC(5,2)	-23.5	02 35 0D

SQLLEN field for decimal

The SQLLEN field contains the precision (first byte) and scale (second byte) of the decimal column. If writing a portable application, the precision and scale bytes should be set individually, versus setting them together as a short integer. This will avoid integer byte reversal problems.

For example, in C:

```
((char *)&(sqlda->sqlvar[i].sqllen))[0] = precision;
((char *)&(sqlda->sqlvar[i].sqllen))[1] = scale;
```

Exception tables

Exception tables are user-created tables that mimic the definition of the tables that are specified to be checked using the SET INTEGRITY statement with the IMMEDIATE CHECKED option. They are used to store copies of the rows that violate constraints in the tables being checked.

The exception tables that are used by the load utility are identical to the ones described here, and can therefore be reused during checking with the SET INTEGRITY statement.

Rules for creating an exception table

The rules for creating an exception table are as follows:

- If the table is protected by a security policy, the exception table must be protected by the same security policy.
- The first "n" columns of the exception table are the same as the columns of the table being checked. All column attributes, including name, data type, and length should be identical. For protected columns, the security label protecting the column must be the same in both tables.
- All of the columns of the exception table must be free of constraints and triggers. Constraints include referential integrity and check constraints, as well as unique index constraints that could cause errors on insert.

- The "(n+1)" column of the exception table is an optional TIMESTAMP column. This serves to identify successive invocations of checking by the SET INTEGRITY statement on the same table, if the rows within the exception table have not been deleted before issuing the SET INTEGRITY statement to check the data. The timestamp precision can be any value from 0 to 12 and the value assigned will be the result of CURRENT TIMESTAMP special register
- The "(n+2)" column should be of type CLOB(32K) or larger. This column is optional but recommended, and will be used to give the names of the constraints that the data within the row violates. If this column is not provided (as could be warranted if, for example, the original table had the maximum number of columns allowed), then only the row where the constraint violation was detected is copied.
- The exception table should be created with both "(n+1)" and "(n+2)" columns.
- There is no enforcement of any particular name for the previously listed additional columns. However, the type specification must be exactly followed.
- No additional columns are allowed.
- If the original table has generated columns (including the IDENTITY property), the corresponding columns in the exception table should not specify the generated property.
- Users invoking the SET INTEGRITY statement to check data must hold the INSERT privilege on the exception tables.
- The exception table cannot be a data partitioned table, a range clustered table, or a detached table.
- The exception table cannot be a materialized query table or a staging table.
- The exception table cannot have any dependent refresh immediate materialized query tables or any dependent propagate immediate staging tables.

The information in the "message" column has the following structure:

Table 323. Exception Table Message Column Structure

Field number	Contents	Size	Comments
1	Number of constraint violations	5 bytes	Right justified padded with "0"
2	Type of first constraint violation	1 byte	• "D" - Delete Cascade violation • "F" - Foreign Key violation • "G" - Generated Column violation • "I" - Unique Index violation[a] • "K" - Check Constraint violation • "L" - LBAC Write rules violation • "P" - Data Partitioning violation • "S" - Invalid Row Security Label • "X" - Index defined on XML column violation[d]
3	Length of constraint/column[b] /index ID[c]	5 bytes	Right justified padded with "0"
4	Constraint name/Column name[b]/index ID[c]	length from the previous field	
5	Separator	3 bytes	<space><colon><space>

Table 323. Exception Table Message Column Structure (continued)

Field number	Contents	Size	Comments
6	Type of next constraint violation	1 byte	• "D" - Delete Cascade violation • "F" - Foreign Key violation • "G" - Generated Column violation • "I" - Unique Index violation • "K" - Check Constraint violation • "L" - LBAC Write rules violation • "P" - Data Partitioning violation • "S" - Invalid Row Security Label • "X" - Index defined on XML column violation[d]
7	Length of constraint/column/index ID	5 bytes	Right justified padded with "0"
8	Constraint name/Column name/Index ID	length from the previous field	
.....	.....	.....	Repeat Field 5 through 8 for each violation

- [a] Unique index violations will not occur during checking using the SET INTEGRITY statement, unless it is after an attach operation. This will be reported, however, when running LOAD if the FOR EXCEPTION option is chosen. However, LOAD will not report check constraint, generated column, foreign key, delete cascade, or data partitioning violations in the exception tables.
- [b] To retrieve the expression of a generated column from the catalog views, use a select statement. For example, if field 4 is MYSCHEMA.MYTABLE.GEN_1, then SELECT SUBSTR(TEXT, 1, 50) FROM SYSCAT.COLUMNS WHERE TABSCHEMA='MYSCHEMA' AND TABNAME='MYNAME' AND COLNAME='GEN_1'; will return the first fifty bytes of the expression, in the form "AS (<expression>)"
- [c] To retrieve an index ID from the catalog views, use a select statement. For example, if field 4 is 1234, then SELECT INDSCHEMA, INDNAME FROM SYSCAT.INDEXES WHERE IID=1234.
- [d] For Index defined on XML column violations, the constraint name, column name, or index ID field identifies the XML column that had an integrity violation in one of its indexes. It does not identify the index that had the integrity violation. It identifies only the name of the XML column on which the index violation occurs. For example, the value "X00006XTCOLZ" in the message column indicates an index violation occurred in one of the indexes on the XTCOL2 column.

Handling rows in an exception table

The information in exception tables can be processed in various ways. Data can be corrected and rows re-inserted into the original tables.

If there are no INSERT triggers on the original table, transfer the corrected rows by issuing an INSERT statement with a subquery on the exception table.

If there are INSERT triggers, and you want to complete the load operation with the corrected rows from exception tables without firing the triggers:

- Design the INSERT triggers to be fired depending on the value in a column that has been defined explicitly for the purpose.
- Unload data from the exception tables and append it using the load utility. In this case, if you want to recheck the data, note that constraints checking is not confined to the appended rows.
- Save the trigger definition text from the relevant system catalog view. Then drop the INSERT trigger and use INSERT to transfer the corrected rows from the exception tables. Finally, re-create the trigger using the saved trigger definition.

No explicit provision is made to prevent the firing of triggers when inserting rows from exception tables.

Only one violation per row is reported for unique index violations.

If values with LONG VARCHAR, LONG VARGRAPHIC, or LOB data types are in the table, the values are not inserted into the exception table in the case of unique index violations.

Querying exception tables

The message column structure in an exception table is a concatenated list of constraint names, lengths, and delimiters, as described earlier. This information can be queried.

For example, to retrieve a list of all violations, repeating each row with only the constraint name, assume that the original table T1 had two columns, C1 and C2. Assume also, that the corresponding exception table, E1, has columns C1 and C2, corresponding to those in T1, as well as a message column, MSGCOL. The following query uses recursion to list one constraint name per row (repeating rows that have more than one violation):

```
WITH IV  (C1, C2, MSGCOL, CONSTNAME, I, J) AS
 (SELECT C1, C2, MSGCOL,
     CHAR(SUBSTR(MSGCOL, 12,
                 INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0)))),
     1,
     15+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0))
    FROM E1
  UNION ALL
  SELECT C1, C2, MSGCOL,
      CHAR(SUBSTR(MSGCOL, J+6,
                  INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0)))),
      I+1,
      J+9+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0))
    FROM IV
    WHERE I < INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,1,5)),5,0))
 ) SELECT C1, C2, CONSTNAME FROM IV;
```

To list all of the rows that violated a particular constraint, the previous query could be extended as follows:

```
WITH IV  (C1, C2, MSGCOL, CONSTNAME, I, J) AS
 (SELECT C1, C2, MSGCOL,
      CHAR(SUBSTR(MSGCOL, 12,
                  INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0)))),
      1,
      15+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0))
    FROM E1
  UNION ALL
  SELECT C1, C2, MSGCOL,
      CHAR(SUBSTR(MSGCOL, J+6,
                 INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0)))),
      I+1,
      J+9+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0))
    FROM IV
    WHERE I < INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,1,5)),5,0))
 ) SELECT C1, C2, CONSTNAME FROM IV WHERE CONSTNAME = 'constraintname';
```

The following query could be used to obtain all of the check constraint violations:

```
WITH IV  (C1, C2, MSGCOL, CONSTNAME, CONSTTYPE, I, J) AS
  (SELECT C1, C2, MSGCOL,
       CHAR(SUBSTR(MSGCOL, 12,
                   INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0)))),
       CHAR(SUBSTR(MSGCOL, 6, 1)),
       1,
       15+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0))
     FROM E1
   UNION ALL
   SELECT C1, C2, MSGCOL,
       CHAR(SUBSTR(MSGCOL, J+6,
                  INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0)))),
       CHAR(SUBSTR(MSGCOL, J, 1)),
       I+1,
       J+9+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0))
     FROM IV
     WHERE I < INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,1,5)),5,0))
 ) SELECT C1, C2, CONSTNAME FROM IV WHERE CONSTTYPE = 'K';
```

Regular expression control characters

Control characters are metacharacters, operators, and replacement text characters that can be used in regular expressions.

Only half-width characters are recognized. Any full-width characters that correspond to the characters in the following tables are not recognized.

Table 324. Regular expression metacharacters

Character	Allowed outside of sets	Allowed inside of sets	Description
\a	Yes	Yes	Match a BELL, \u0007
\A	Yes		Match at the beginning of the input. Differs from ^ in that \A does not match after a new line within the input.
\b	Yes		Match if the current position is a word boundary. Boundaries occur at the transitions between word (\w) and non-word (\W) characters, with combining marks ignored.
\B	Yes		Match if the current position is not a word boundary.
\cX	Yes	Yes	Match a control-X character.
\d	Yes	Yes	Match any character with the Unicode General Category of Nd (Number, Decimal Digit.)
\D	Yes	Yes	Match any character that is not a decimal digit.
\e	Yes	Yes	Match an ESCAPE, \u001B
\E	Yes	Yes	Terminates a \Q ... \E quoted sequence.
\f	Yes	Yes	Match a FORM FEED, \u000C.
\G	Yes		Match if the current position is at the end of the previous match.
\n	Yes	Yes	Match a LINE FEED, \u000A
\N{UNICODE CHARACTER NAME}	Yes	Yes	Match the named character.
\p{UNICODE PROPERTY NAME}	Yes	Yes	Match any character with the specified Unicode Property.
\P{UNICODE PROPERTY NAME}	Yes	Yes	Match any character not having the specified Unicode Property.
\Q	Yes	Yes	Places quotation marks around all following characters until \E
\r	Yes	Yes	Match a CARRIAGE RETURN, \u000D
\s	Yes	Yes	Match a white space character. White space is defined as [\t\n\f\r\p{Z}]
\S	Yes	Yes	Match a non-white space character.
\t	Yes	Yes	Match a HORIZONTAL TABULATION, \u0009
\uhhhh	Yes	Yes	Match the character with the hex value hhhh

Table 324. Regular expression metacharacters (continued)

Character	Allowed outside of sets	Allowed inside of sets	Description
\Uhhhhhhhh	Yes	Yes	Match the character with the hex value hhhhhhhh. Exactly 8 hex digits must be provided, even though the largest Unicode code point is \U0010ffff

\w	Yes	Yes	Match a word character. Word characters are as follows:
			`[\p{Alphabetic}`
			`\p{Mark}`
			`\p{Decimal_Number}`
			`\p{Connector_Punctuation}`
			`\u200c`
			`\u200d]`

\W	Yes	Yes	Match a non-word character.
\x{hhhh}	Yes	Yes	Match the character with hex value hhhh. From one to 6 hex digits can be supplied.
\xhh	Yes	Yes	Match the character with two-digit hex value hh
\X	Yes		Match a Grapheme Cluster.
\Z	Yes		Match if the current position is at the end of input, but before the final line terminator, if one exists.
\z	Yes		Match if the current position is at the end of input.
\n	Yes		Back Reference. Match whatever the nth capturing group matched. n must be a number > 1 and < total number of capture groups in the pattern.
\0ooo	Yes	Yes	Match an Octal character. 'ooo' is from one to three octal digits. 0377 is the largest allowed Octal character. The leading zero is required; it distinguishes Octal constants from back references.
[pattern]	Yes	Yes	Match any one character from the set.
.	Yes		Match any character.
^	Yes		Match at the beginning of a line.
$	Yes		Match at the end of a line.
\	Yes		Places quotation marks around the character that follows. Characters that must have surrounding quotation marks to be treated as literals are * ? + [( ) { } ^ $ | \ . /
\		Yes	Places quotation marks around the character that follows. Characters that must be quoted to be treated as literals are [ ] \ Characters that might need to be quoted, depending on the context are - &

Table 325. Regular expression operators

Operator	Description
|	Alternation. A | B matches either A or B

Table 325. Regular expression operators (continued)

Operator	Description
*	Match 0 or more times. Match as many times as possible.
+	Match 1 or more times. Match as many times as possible.
?	Match zero or one time. Prefer one.
{n}	Match exactly n times.
{n,}	Match at least n times. Match as many times as possible.
{n,m}	Match between n and m times. Match as many times as possible, but not more than m.
*?	Match 0 or more times. Match as few times as possible.
+?	Match 1 or more times. Match as few times as possible.
??	Match zero or one time. Prefer zero.
{n}?	Match exactly n times.
{n,}?	Match at least n times, but no more than required for an overall pattern match.
{n,m}?	Match between n and m times. Match as few times as possible, but not less than n
*+	Match 0 or more times. Match as many times as possible when first encountered, do not retry with fewer even if overall match fails (Possessive Match)
++	Match 1 or more times. Possessive match.
?+	Match zero or 1 time. Possessive match.
{n}+	Match exactly n times.
{n,}+	Match at least n times. Possessive Match.
{n,m}+	Match between n and m times. Possessive Match.
(...)	Capturing parentheses. Range of input that matched the parenthesized subexpression is available after the match.
(?: ...)	Non-capturing parentheses. Groups the included pattern, but does not provide capturing of matching text. More efficient than capturing parentheses.
(?> ...)	Atomic-match parentheses. First match of the parenthesized subexpression is the only one tried. If it does not lead to an overall pattern match, back up the search for a match to a position before the "(?>"
(?# ...)	Free-format comment (?# comment)
(?= ...)	Look-ahead assertion. True if the parenthesized pattern matches at the current input position, but does not advance the input position.
(?! ...)	Negative look-ahead assertion. True if the parenthesized pattern does not match at the current input position. Does not advance the input position.
(?<= ...)	Look-behind assertion. True if the parenthesized pattern matches text that precedes the current input position. The last character of the match is the input character just before the current position. Does not alter the input position. The length of possible strings that is matched by the look-behind pattern must not be unbounded (no * or + operators.)

Table 325. Regular expression operators (continued)	
Operator	**Description**
`(?<!...)`	Negative Look-behind assertion. True if the parenthesized pattern does not match text that precedes preceding the current input position. The last character of the match is the input character just before the current position. Does not alter the input position. The length of possible strings that is matched by the look-behind pattern must not be unbounded (no * or + operators.)
`(?ismwx-ismwx: ... )`	Flag settings. Evaluate the parenthesized expression with the specified flags enabled or disabled.
`(?ismx-ismx)`	Flag settings. Change the flag settings. Changes apply to the portion of the pattern that follows the setting. For example, `(?i)` changes to a not case-sensitive match.

Table 326. Set expressions (character classes)	
Example expression	**Description**
`[abc]`	Match any of the characters a, b, or c
`[^abc]`	Negation - match any character except a, b, or c
`[A-M]`	Range - match any character from A to M. The characters to include are determined by Unicode code point order.
`[\u0000-\U0010ffff]`	Range - match all characters.

`[\p{Letter}]`
`[\p{General_Category=Letter}]`
`[\p{L}]`
	Characters with Unicode Category = `Letter`. All forms that are shown are equivalent.

`[\P{Letter}]`	Negated property. (Uppercase \P) Match everything except Letters.
`[\p{numeric_value=9}]`	Match all numbers with a numeric value of 9. Any Unicode Property might be used in set expressions.
`[\p{Letter}&&\p{script=cyrillic}]`	Logical AND or intersection. Match the set of all Cyrillic letters.
`[\p{Letter}--\p{script=latin}]`	Subtraction. Match all non-Latin letters.
`[[a-z][A-Z][0-9]] [a-zA-Z0-9]]`	Implicit Logical OR or Union of Sets. The examples match ASCII letters and digits. The two forms are equivalent.
`[:script=Greek:]`	Alternate POSIX-like syntax for properties. Equivalent to `\p{script=Greek}`

Explain tables

The Explain tables capture access plans when the Explain facility is activated.

The Explain tables must be created before Explain can be invoked. You can create them using one of the following methods:

- Call the SYSPROC.SYSINSTALLOBJECTS procedure:

```
CONNECT TO database-name
CALL SYSPROC.SYSINSTALLOBJECTS('EXPLAIN', 'C',
        CAST (NULL AS VARCHAR(128)), CAST (NULL AS VARCHAR(128)))
```

This call creates the explain tables under the SYSTOOLS schema. To create them under a different schema, specify a schema name as the last parameter in the call.

- Run the EXPLAIN.DDL command file:

```
CONNECT TO database-name
db2 -tf EXPLAIN.DDL
```

This command file creates explain tables under the current schema. The location of this command file depends on the operating system, as illustrated in the following table:

Operating system	Location of the EXPLAIN command file
Linux AIX	Located in the INSTHOME/sqllib/misc directory. INSTHOME is the instance home directory.
Windows	Located at the DB2PATH\misc directory on Windows operating systems. DB2PATH is the location where you install your Db2 copy

Calling the SYSPROC.SYSINSTALLOBJECTS procedure is preferred over using the EXPLAIN.DDL file since it can automatically adapt to different database configurations. For example, if **BLOCKNONLOGGED** parameter is set to yes, then some statements in EXPLAIN.DDL fail because **NOT LOGGED** clause is used for LOB columns. However, if **BLOCKNONLOGGED** parameter is set to yes then the SYSPROC.SYSINSTALLOBJECTS procedure automatically avoids the use of **NOT LOGGED** clause.

Db2 modification packs can occasionally contain changes to the format of one or more of the Explain tables. When such changes occur, existing Explain tables will need to be re-created, as discussed above, or be updated using either the **db2exmig** tool or the SYSINSTALLOBJECTS procedure (using the M action). Failure to update existing Explain tables can result in one of the following errors occurring during use of the Explain facility or one of its related tools: SQL0206N, SQL0220N, SQL1184N.

Note: The format of the Explain tables has been changed in version 11.5.5.

The Explain facility uses the following IDs as the schema when qualifying Explain tables that it is populating:

- The session authorization ID for dynamic SQL
- The statement authorization ID for static SQL
- The SYSTOOLS schema if explain tables do not exist with the authorization ID schema

The schema can be associated with a set of Explain tables, or aliases that point to a set of Explain tables under a different schema. If no Explain tables are found under the schema, the Explain facility checks for Explain tables under the SYSTOOLS schema and attempts to use those tables.

The population of the Explain tables by the Explain facility will not activate triggers or referential or check constraints. For example, if an insert trigger were defined on the EXPLAIN_INSTANCE table, and an eligible statement were explained, the trigger would not be activated.

To improve the performance of the Explain facility in a partitioned database system, it is recommended that the Explain tables be created in a single partition database partition group, preferably on the same database partition to which you will be connected when compiling the query.

ADVISE_INDEX table

The ADVISE_INDEX table represents the recommended indexes.

Table 327. ADVISE_INDEX Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable ?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	No	Authorization ID of initiator of this Explain request.

Table 327. ADVISE_INDEX Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable ?	Key?	Description
EXPLAIN_TIME	TIMESTAMP	No	No	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	No	Name of the package running when the dynamic statement was explained or name of the source file when static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	No	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	No	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	No	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	No	Statement number within package to which this explain information is related.
SECTNO	INTEGER	No	No	Section number within package to which this explain information is related.
QUERYNO	INTEGER	No	No	Numeric identifier for explained SQL statement. For dynamic SQL statements (excluding the EXPLAIN SQL statement) issued through CLP or CLI, the default value is a sequentially incremented value. Otherwise, the default value is the value of STMTNO for static SQL statements and 1 for dynamic SQL statements.
QUERYTAG	CHAR(20)	No	No	Identifier tag for each explained SQL statement. For dynamic SQL statements issued through CLP (excluding the EXPLAIN SQL statement), the default value is "CLP". For dynamic SQL statements issued through CLI (excluding the EXPLAIN SQL statement), the default value is "CLI". Otherwise, the default value used is blanks.
NAME	VARCHAR(128)	No	No	Name of the index.
CREATOR	VARCHAR(128)	No	No	Qualifier of the index name.
TBNAME	VARCHAR(128)	No	No	Name of the table or nickname on which the index is defined.
TBCREATOR	VARCHAR(128)	No	No	Qualifier of the table name.
COLNAMES	CLOB(2M)	No	No	List of column names.
UNIQUERULE	CHAR(1)	No	No	Unique rule: • D = Duplicates allowed • P = Primary index • U = Unique entries only allowed
COLCOUNT	SMALLINT	No	No	Number of columns in the key plus the number of include columns if any.
IID	SMALLINT	No	No	Internal index ID.

Table 327. ADVISE_INDEX Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable?	Key?	Description
NLEAF	BIGINT	No	No	Number of leaf pages; -1 if statistics are not gathered.
NLEVELS	SMALLINT	No	No	Number of index levels; -1 if statistics are not gathered.
FIRSTKEYCARD	BIGINT	No	No	Number of distinct first key values; -1 if statistics are not gathered.
FULLKEYCARD	BIGINT	No	No	Number of distinct full key values; -1 if statistics are not gathered.
CLUSTERRATIO	SMALLINT	No	No	Degree of data clustering with the index; -1 if statistics are not gathered or if detailed index statistics are gathered (in which case, CLUSTERFACTOR will be used instead).
AVGPARTITION_ CLUSTERRATIO	SMALLINT	No	No	Degree of data clustering within a single data partition. -1 if the table is not table partitioned, if statistics are not gathered, or if detailed statistics are gathered (in which case AVGPARTITION_CLUSTERFACTOR will be used instead).
AVGPARTITION_ CLUSTERFACTOR	DOUBLE	No	No	Finer measurement of the degree of clustering within a single data partition. -1 if the table is not table partitioned, if statistics are not gathered, or if the index is defined on a nickname.
AVGPARTITION_PAGE_ FETCH_PAIRS	VARCHAR(520)	No	No	A list of paired integers in character form. Each pair represents a potential buffer pool size and the corresponding page fetches required to access a single data partition from the table. Zero-length string if no data is available, or if the table is not table partitioned.
DATAPARTITION_ CLUSTERFACTOR	DOUBLE	No	No	A statistic measuring the "clustering" of the index keys with regard to data partitions. This field holds a number between zero and one, with one representing perfect clustering and zero representing no clustering.
CLUSTERFACTOR	DOUBLE	No	No	Finer measurement of degree of clustering, or -1 if detailed index statistics have not been gathered or if the index is defined on a nickname.
USERDEFINED	SMALLINT	No	No	Defined by the user.

Table 327. ADVISE_INDEX Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable?	Key?	Description
SYSTEM_REQUIRED	SMALLINT	No	No	• 1 if one or the other of the following conditions is met: – This index is required for a primary or unique key constraint, or this index is a dimension block index or composite block index for a multi-dimensional clustering (MDC) table. – This is an index on the (OID) column of a typed table. • 2 if both of the following conditions are met: – This index is required for a primary or unique key constraint, or this index is a dimension block index or composite block index for an MDC table. – This is an index on the (OID) column of a typed table. • 0 otherwise.
CREATE_TIME	TIMESTAMP	No	No	Time when the index was created.
STATS_TIME	TIMESTAMP	Yes	No	Last time when any change was made to recorded statistics for this index. Null if no statistics available.
PAGE_FETCH_PAIRS	VARCHAR(520)	No	No	A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the table with this index using that hypothetical buffer. (Zero-length string if no data available.)
REMARKS	VARCHAR(254)	Yes	No	User-supplied comment, or null.
DEFINER	VARCHAR(128)	No	No	User who created the index.
CONVERTED	CHAR(1)	No	No	Reserved for future use.
SEQUENTIAL_PAGES	BIGINT	No	No	Number of leaf pages located on disk in index key order with few or no large gaps between them. (-1 if no statistics are available.)
DENSITY	INTEGER	No	No	Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percent (integer between 0 and 100, -1 if no statistics are available.)
FIRST2KEYCARD	BIGINT	No	No	Number of distinct keys using the first two columns of the index (-1 if no statistics or inapplicable)
FIRST3KEYCARD	BIGINT	No	No	Number of distinct keys using the first three columns of the index (-1 if no statistics or inapplicable)

Table 327. ADVISE_INDEX Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable ?	Key?	Description
FIRST4KEYCARD	BIGINT	No	No	Number of distinct keys using the first four columns of the index (-1 if no statistics or inapplicable)
PCTFREE	SMALLINT	No	No	Percentage of each index leaf page to be reserved during initial building of the index. This space is available for future inserts after the index is built.
UNIQUE_COLCOUNT	SMALLINT	No	No	The number of columns required for a unique key. Always <=COLCOUNT. < COLCOUNT only if there a include columns. -1 if index has no unique key (permits duplicates)
MINPCTUSED	SMALLINT	No	No	If not zero, then online index defragmentation is enabled, and the value is the threshold of minimum used space before merging pages.
REVERSE_SCANS	CHAR(1)	No	No	• Y = Index supports reverse scans • N = Index does not support reverse scans
USE_INDEX	CHAR(1)	Yes	No	• Y = index recommended or evaluated • N = index not to be recommended • R = an existing clustering RID index was recommended (by the Design Advisor) to be unclustered; this is the case when a new clustering RID index is recommended for the table • I = Ignore an existing non-unique index. The EXISTS column should be "Y" in this case or the index will not be ignored.
CREATION_TEXT	CLOB(2M)	No	No	The SQL statement used to create the index.
PACKED_DESC	BLOB(1M)	Yes	No	Internal description of the table.
RUN_ID	TIMESTAMP	Yes	FK	A value corresponding to the START_TIME of a row in the ADVISE_INSTANCE table, linking it to the same Design Advisor run.
INDEXTYPE	VARCHAR(4)	No	No	Type of index. • CLUS = Clustering • REG = Regular • DIM = Dimension block index • BLOK = Block index
EXISTS	CHAR(1)	No	No	Set to "Y" if the index exists in the database catalog or "N" if the index does not currently exist in the catalog.
RIDTOBLOCK	CHAR(1)	No	No	Set to "Y" if the RID index was used to make a block index in the Design Advisor.

Table 327. ADVISE_INDEX Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable?	Key?	Description
NULLKEYS	CHAR(1)	No	No	Specifies whether null keys are indexed. • N = Keys that contain all null values are not indexed (not considering columns or expressions from the INCLUDE clause) • Y = Keys that contain all null values are indexed (not considering columns or expressions from the INCLUDE clause)

ADVISE_INSTANCE table

The ADVISE_INSTANCE table contains information about db2advis execution, including start time.

Contains one row for each execution of db2advis. Other ADVISE tables have a foreign key (RUN_ID) that links to the START_TIME column of the ADVISE_INSTANCE table for rows created during the same Design Advisor run.

Table 328. ADVISE_INSTANCE Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
START_TIME	TIMESTAMP	No	PK	Time at which db2advis execution begins.
END_TIME	TIMESTAMP	No	No	Time at which db2advis execution ends.
MODE	VARCHAR(4)	No	No	The value that was specified with the -m option on the Design Advisor; for example, 'MC' to specify MQT and MDC.
WKLD_COMPRESSION	CHAR(4)	No	No	The workload compression under which the Design Advisor was run.
STATUS	CHAR(9)	No	No	The status of a Design Advisor run. Status can be 'STARTED', 'COMPLETED' (if successful), or an error number that is prefixed by 'EI' for internal errors or 'EX' for external errors, in which case the error number represents the SQLCODE.

ADVISE_MQT table

The ADVISE_MQT table contains information about materialized query tables (MQT) recommended by the Design Advisor.

Table 329. ADVISE_MQT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	No	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	No	Time of initiation for Explain request.

Table 329. ADVISE_MQT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable ?	Key?	Description
SOURCE_NAME	VARCHAR(128)	No	No	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	No	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	No	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	No	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	No	Statement number within package to which this Explain information is related.
SECTNO	INTEGER	No	No	Statement number within package to which this Explain information is related.
NAME	VARCHAR(128)	No	No	MQT name.
CREATOR	VARCHAR(128)	No	No	MQT creator name.
IID	SMALLINT	No	No	Internal identifier.
CREATE_TIME	TIMESTAMP	No	No	Time at which the MQT was created.
STATS_TIME	TIMESTAMP	Yes	No	Time at which statistics were taken.
NUMROWS	DOUBLE	No	No	The number of estimated rows in the MQT.
NUMCOLS	SMALLINT	No	No	Number of columns defined in the MQT.
ROWSIZE	DOUBLE	No	No	Average length (in bytes) of a row in the MQT.
BENEFIT	FLOAT	No	No	Reserved for future use.
USE_MQT	CHAR(1)	Yes	No	Set to 'Y' when the MQT is recommended.
MQT_SOURCE	CHAR(1)	Yes	No	Indicates where the MQT candidate was generated. Set to 'I' if the MQT candidate is a refresh-immediate MQT, or 'D' if it can only be created as a full refresh-deferred MQT.
QUERY_TEXT	CLOB(2M)	No	No	Contains the query that defines the MQT.
CREATION_TEXT	CLOB(2M)	No	No	Contains the CREATE TABLE DDL for the MQT.
SAMPLE_TEXT	CLOB(2M)	No	No	Contains the sampling query that is used to get detailed statistics for the MQT. Only used when detailed statistics are required for the Design Advisor. The resulting sampled statistics will be shown in this table. If null, then no sampling query was created for this MQT.
COLSTATS	CLOB(2M)	No	No	Contains the column statistics for the MQT (if not null). These statistics are in XML format and include the column name, column cardinality and, optionally, the HIGH2KEY and LOW2KEY values.
EXTRA_INFO	BLOB(2M)	No	No	Reserved for miscellaneous output.

Table 329. ADVISE_MQT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable?	Key?	Description
TBSPACE	VARCHAR(128)	No	No	The table space that is recommended for the MQT.
RUN_ID	TIMESTAMP	Yes	FK	A value corresponding to the START_TIME of a row in the ADVISE_INSTANCE table, linking it to the same Design Advisor run.
REFRESH_TYPE	CHAR(1)	No	No	Set to 'I' for immediate or 'D' for deferred.
EXISTS	CHAR(1)	No	No	Set to 'Y' if the MQT exists in the database catalog.

ADVISE_PARTITION table

The ADVISE_PARTITION table contains information about database partitions recommended by the Design Advisor, and can only be populated in a partitioned database environment.

Table 330. ADVISE_PARTITION Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	No	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	No	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	No	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	No	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	No	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	No	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	No	Statement number within package to which this Explain information is related.
SECTNO	INTEGER	No	No	Statement number within package to which this Explain information is related.
QUERYNO	INTEGER	No	No	Numeric identifier for explained SQL statement. For dynamic SQL statements (excluding the EXPLAIN SQL statement) issued through CLP or CLI, the default value is a sequentially incremented value. Otherwise, the default value is the value of STMTNO for static SQL statements and 1 for dynamic SQL statements.

Table 330. ADVISE_PARTITION Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable ?	Key?	Description
QUERYTAG	CHAR(20)	No	No	Identifier tag for each explained SQL statement. For dynamic SQL statements issued through CLP (excluding the EXPLAIN SQL statement), the default value is 'CLP'. For dynamic SQL statements issued through CLI (excluding the EXPLAIN SQL statement), the default value is 'CLI'. Otherwise, the default value used is blanks.
TBNAME	VARCHAR(128)	Yes	No	Specifies the table name.
TBCREATOR	VARCHAR(128)	Yes	No	Specifies the table creator name.
PMID	SMALLINT	Yes	No	Specifies the distribution map ID.
TBSPACE	VARCHAR(128)	Yes	No	Specifies the table space in which the table resides.
COLNAMES	CLOB(2M)	Yes	No	Specifies database partition column names, separated by commas.
COLCOUNT	SMALLINT	Yes	No	Specifies the number of database partitioning columns.
REPLICATE	CHAR(1)	Yes	No	Specifies whether or not the database partition is replicated.
COST	DOUBLE	Yes	No	Specifies the cost of using the database partition.
USEIT	CHAR(1)	Yes	No	Specifies whether or not the database partition is used in EVALUATE PARTITION mode. A database partition is used if USEIT is set to 'Y' or 'y'.
RUN_ID	TIMESTAMP	Yes	FK	A value corresponding to the START_TIME of a row in the ADVISE_INSTANCE table, linking it to the same Design Advisor run.

ADVISE_TABLE table

The ADVISE_TABLE table stores the data definition language (DDL) for table creation, using the final Design Advisor recommendations for materialized query tables (MQTs), multidimensional clustered tables (MDCs), and database partitioning.

Table 331. ADVISE_TABLE Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable ?	Key?	Description
RUN_ID	TIMESTAMP	Yes	FK	A value corresponding to the START_TIME of a row in the ADVISE_INSTANCE table, linking it to the same Design Advisor run.
TABLE_NAME	VARCHAR(128)	No	No	Name of the table.
TABLE_SCHEMA	VARCHAR(128)	No	No	Name of the table creator.
TABLESPACE	VARCHAR(128)	No	No	The table space in which the table is to be created.

Table 331. ADVISE_TABLE Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable ?	Key?	Description
SELECTION_FLAG	VARCHAR(4)	No	No	Indicates the recommendation type. Valid values are 'M' for MQT, 'P' for database partitioning, and 'C' for MDC. This field can include any subset of these values. For example, 'MC' indicates that the table is recommended as an MQT and an MDC table.
TABLE_EXISTS	CHAR(1)	No	No	Set to 'Y' if the table exists in the database catalog.
USE_TABLE	CHAR(1)	No	No	Set to 'Y' if the table has recommendations from the Design Advisor.
GEN_COLUMNS	CLOB(2M)	No	No	Contains a generated columns string if this row includes an MDC recommendation that requires generated columns in the create table DDL.
ORGANIZE_BY	CLOB(2M)	No	No	For MDC recommendations, contains the ORGANIZE BY clause of the create table DDL.
CREATION_TEXT	CLOB(2M)	No	No	Contains the create table DDL.
ALTER_COMMAND	CLOB(2M)	No	No	Contains an ALTER TABLE statement for the table.

ADVISE_WORKLOAD table

The ADVISE_WORKLOAD table represents the statement that makes up the workload.

Table 332. ADVISE_WORKLOAD Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable ?	Key?	Description
WORKLOAD_NAME	CHAR(128)	No	No	Name of the collection of SQL statements (workload) to which this statement belongs.
STATEMENT_NO	INTEGER	No	No	Statement number within the workload to which this explain information is related.
STATEMENT_TEXT	CLOB(1M)	No	No	Content of the SQL statement.
STATEMENT_TAG	VARCHAR(256)	No	No	Identifier tag for each explained SQL statement.
FREQUENCY	INTEGER	No	No	The number of times this statement appears within the workload.
IMPORTANCE	DOUBLE	No	No	Importance of the statement.
WEIGHT	DOUBLE	No	No	Priority of the statement.
COST_BEFORE	DOUBLE	Yes	No	The cost of the query (in timerons) if the recommendations are not created.

Table 332. ADVISE_WORKLOAD Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable?	Key?	Description
COST_AFTER	DOUBLE	Yes	No	The cost of the query (in timerons) if the recommendations are created. COST_AFTER reflects all recommendations except those that pertain to clustered indexes and multidimensional clustering (MDC).
COMPILABLE	CHAR(17)	Yes	No	Indicates any query compile errors that occurred while trying to prepare the statement. If this column is NULL or does not start with SQLCA, the SQL query could be compiled by db2advis. If a compile error is found by db2advis or the Design Advisor, the COMPILABLE column value consists of an 8 byte long SQLCA.sqlcaid field, followed by a colon (:) and an 8 byte long SQLCA.sqlstate field, which is the return code for the SQL statement.

EXPLAIN_ACTUALS table

The EXPLAIN_ACTUALS table contains Explain section actuals information.

Table 333. EXPLAIN_ACTUALS Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	FK	Statement number within package to which this Explain information is related.
SECTNO	INTEGER	No	FK	Section number within package to which this Explain information is related.
OPERATOR_ID	INTEGER	No	FK	Unique ID for this operator within this query.
DBPARTITIONNUM	INTEGER	No	No	The partition number of the database partition where the operator has run.
PREDICATE_ID	INTEGER	Yes	No	ID of the predicate applied on this operator. NULL if the actuals are operator actuals.
HOW_APPLIED	CHAR(10)	Yes	No	How predicate is used by this operator. NULL if PREDICATE_ID is NULL.
ACTUAL_TYPE	VARCHAR(12)	No	No	The type of the actual.
ACTUAL_VALUE	DOUBLE	Yes	No	The value of the actual. NULL if actual is not available for this operator.

EXPLAIN_ARGUMENT table

The EXPLAIN_ARGUMENT table represents the unique characteristics for each individual operator, if there are any.

Table 334. EXPLAIN_ARGUMENT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	FK	Statement number within package to which this Explain information is related.
SECTNO	INTEGER	No	FK	Section number within package to which this Explain information is related.
OPERATOR_ID	INTEGER	No	No	Unique ID for this operator within this query.
ARGUMENT_TYPE	CHAR(8)	No	No	The type of argument for this operator.
ARGUMENT_VALUE	VARCHAR(1024)	Yes	No	The value of the argument for this operator. NULL if the value is in LONG_ARGUMENT_VALUE.
LONG_ARGUMENT_ VALUE	CLOB(2M)	Yes	No	The value of the argument for this operator, when the text will not fit in ARGUMENT_VALUE. NULL if the value is in ARGUMENT_VALUE.

Table 335. ARGUMENT_TYPE and ARGUMENT_VALUE column values

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
AGGMODE	COMPLETE PARTIAL HASHED PARTIAL HASHED COMPLETE INTERMEDIATE FINAL COMPLETE UNIQUE PARTIAL UNIQUE INTERMEDIATE UNIQUE FINAL UNIQUE	Indicates how the operator aggregates values; for example, whether the aggregation is complete or partial aggregation. HASHED COMPLETE identifies a column-organized data grouping.
APREUSE	TRUE	Indicates if access plan reuse bind option is in effect for this statement.
BACKJOIN	TRUE FALSE	Indicates whether the ZZJOIN operator is used as a backjoin in the all-probe list-prefetch plan.
BITFLTR	INTEGER FALSE	Size of a hash join bit filter. A hash join bit filter can sometimes also be used by a table queue.

Table 335. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
BLDLEVEL	Database build identifier.	Internal identification string for source code version. Db2 vVV.RR.MM.FF : nYYMMDDHHMM, for example Db2 v11.1.1.1: n1610171423
BLKLOCK	EXCLUSIVE INTENT EXCLUSIVE INTENT SHARE NONE SHARE UPDATE	Block level lock intent.
BUFFSCAN **Attention:** This column is available in the container-only release of Db2 Version 11.5 Mod Pack 1 and later versions.	FALSE TRUE	Buffered scan.
BUFFSORT	TRUE	Indicates whether SORT is used as a buffering operation.
BUSTSENS	YES NO	Indicates whether the BUSTIMESENSITIVE bind option is in effect for this statement.
BY DPART	TRUE FALSE	Indicates whether ZZJN is performed across the dimensions of a data partitioned table.

Table 335. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description

CONCACCR	Each row of this type will contain:
	• Level of the setting for this statement:
	BIND
	`Application BIND with CONCURRENT ACCESS RESOLUTION option`
	PREP
	`Statement prepared with CONCURRENT ACCESS RESOLUTION attributes`
	• The concurrent access resolution in effect:
	USE CURRENTLY COMMITTED
	`Concurrent access resolution of application bind or statement prepare is USE CURRENTLY COMMITTED`
	WAIT FOR OUTCOME
	`Concurrent access resolution of application bind or statement prepare is WAIT FOR OUTCOME`
		Indicates the concurrent access resolution used to generate the access plan for this statement.

CSERQY	TRUE FALSE	Remote query is a common subexpression.
CSETEMP	TRUE FALSE	Temporary Table over Common Subexpression Flag.
CUR_COMM	TRUE	Access currently committed rows when the value for the database configuration parameter **cur_commit** is not DISABLE. This access plan is enabled for applicable statements by using either: • CONCURRENT ACCESS RESOLUTION with the USE CURRENTLY COMMITTED option on bind or prepare • The database configuration parameter **cur_commit** with a value of ON

Table 335. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
DEGREE	INTEGER	If the RETURN operator represents the return from column-organized data processing of the insertion, update, or deletion of rows, the DEGREE of the RETURN operator represents the maximum degree of all the subsections involved in the statement. The degree of parallelism for specific subsections is indicated by the DEGREE argument on plan operators below the RETURN operator. If the insert/update/delete statements is modifying a column-organized target table, the DEGREE of INSERT/UPDATE/DELETE operator is the number of parallel subagents that are used for the INSERT/UPDATE/DELETE operators.
DIRECT	TRUE	Direct fetch indicator.
DPESTFLG	TRUE FALSE	Indicates whether or not the DPNUMPRT value is based on an estimate. Possible values are "TRUE" (DPNUMPRT represents the estimated number of accessed data partitions) or "FALSE" (DPNUMPRT represents the actual number of accessed data partitions).
DPFXMLMV	REFERENCE COMBINATION	Indicates whether XML column data is moved between DPF partitions.
DPLSTPRT	NONE CHARACTER	Represents accessed data partitions. It is a comma-delimited list (for example: 1,3,5) or a hyphenated list (for example: 1-5) of accessed data partitions. A value of "NONE" means that no data partition remains after specified predicates have been applied.
DPNUMPRT	INTEGER	Represents the actual or estimated number of data partitions accessed.
DSTSEVER	Server name	Destination (ship from) server.
DUPLWARN	TRUE FALSE	Duplicates Warning flag.

Table 335. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
EARLYOUT	LEFT LEFT (REMOVE INNER DUPLICATES) RIGHT GROUPBY NONE	Early out indicator. LEFT indicates that each row from the outer table only needs to be joined with at most one row from the inner table. LEFT (REMOVE INNER DUPLICATES) indicates that an attempt to remove some duplicate rows from the inner table has taken place. RIGHT indicates that each row from the inner table only needs to be joined with at most one row from the outer table. NONE indicates no early out processing. GROUPBY indicates that early out processing is allowed because of a group by operation.
ENVVAR	Each row of this type will contain: • Environment variable name • Environment variable value	Environment variable affecting the optimizer
ERRTOL	Each row of this type will contain an SQLSTATE and SQLCODE pair.	A list of errors to be tolerated.
EXTROWS	Y N The argument does not show in the explain tables and formatted output when the value is "N".	Indicates that the maximum row size of the system temporary table might be too large to fit in a 32K page. Some rows might need to be represented as a large object (LOB).
EVALUNCO	TRUE	Evaluate uncommitted data using lock deferral. This is enabled with the **DB2_EVALUNCOMMITTED** registry variable.
EXECUTID	An opaque binary token formatted as an hexadecimal string representing the executable ID.	Indicates the executable ID of the section being explained.
FETCHMAX	IGNORE INTEGER	Override value for MAXPAGES argument on FETCH operator.

Table 335. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description

FILTER	Each row of this type contains the following items:
	• Filter type identifier (BIT or RANGE)
	• Target operator identifier
	• Size, in bytes, for bit filters; blank for range filters
	Each item in the preceding list is separated by a colon and a space.
		Indicates that a hash join bit filter or a hash join range filter is built at the current operator and provides details for that filter. The use of a filter allows the elimination of rows as early as possible during query execution.
		A hash join bit filter is a space efficient data structure that is used to test whether an element is a member of a set. The following is an example of a hash join bit filter argument value:
		`BIT: 13: 8192`
		A range filter consists of a minimum and a maximum value that defines the valid range for a column value. The following is an example of a hash join range filter argument value:
		`RANGE: 13`

FLTRAPPL	TQ PUSHDOWN	Indicates the bit filter application method used by the optimizer. A value of "TQ PUSHDOWN" indicates that the bit filter operation has been pushed down. This argument will not be included at all when the optimizer does not use a pushdown with the hash join.
GREEDY	TRUE	Indicates whether the optimizer used a greedy algorithm to plan access.
GLOBLOCK	EXCLUSIVE INTENT EXCLUSIVE INTENT NONE INTENT SHARE NO LOCK OBTAINED SHARE SHARE INTENT EXCLUSIVE SUPER EXCLUSIVE UPDATE	Represents global lock intent information for a partitioned table object.
GROUPBYC	TRUE FALSE	Whether Group By columns were provided. This argument can be associated with a GRPBY operator or with a TEMP operator when it is part of a query with multiple distinct aggregations.

Table 335. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
GROUPBYN	Integer	Number of comparison columns. This number may be less than the number of columns present in the GROUP BY clause of the SQL statement if predicates eliminated the need to compare some columns. This argument can be associated with a GRPBY operator or with a TEMP operator when it is part of a query with multiple distinct aggregations.
GROUPBYR	Each row of this type will contain: • Ordinal value of column in group by clause (followed by a colon and a space) • Name of column	Group By requirement. This argument can be associated with a GRPBY operator or with a TEMP operator when it is part of a query with multiple distinct aggregations.
GROUPS	Integer	Number of times the operator will repeat.
HASHCODE	24 32	Size (in bits) of hash join hash code used for hash joins. A hash join hash code can sometimes also be used by a table queue.
HASHTBSZ	INTEGER	The number of expected entries in the hash table of a hash join.
IDXMSTLY	TRUE	Indicates whether the FETCH is performed over block identifiers returned from a multi dimensional clustered index.
IDXOVTMP	TRUE FALSE	Indicates whether the scan builds an index or a fast integer sort structure for random access of the temporary tables. If value is "TRUE", the scan builds an index over the temporary tables for random access of the temporary tables. If value is "FALSE", the scan builds a fast integer sort structure for random access of the temporary tables.
INNERCOL	Each row of this type will contain: • Ordinal value of column in order (followed by a colon and a space) • Name of column • Order value **(A)** Ascending **(D)** Descending	Inner order columns.
INPUTXID	A context node identifier	INPUTXID identifies the input context node used by the XSCAN operator.

Table 335. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
ISCANMAX	IGNORE INTEGER	Override value for MAXPAGES argument on ISCAN operator.
JN INPUT	INNER OUTER	Indicates if operator is the operator feeding the inner or outer of a join.
JUMPSCAN	TRUE FALSE	Indicates that the index scan is a jump scan.
LCKAVOID	TRUE	Lock avoidance: row access will avoid locking committed data.
LISTENER	TRUE FALSE	Listener Table Queue indicator.
MAX CARD	INTEGER	The maximum possible output cardinality for an operator. The value represents the sum across all database partitions on which the operator is executing in a partitioned database environment.
MAXPAGES	ALL NONE INTEGER	Maximum pages expected for Prefetch.
MAXRIDS	NONE INTEGER	Maximum Row Identifiers to be included in each list prefetch request.
MXPPSCAN	TRUE FALSE	Provides additional information about how MAXPAGES is calculated in the case of a jump scan. A jump scan can be conceptualized as multiple contiguous scans separated by jumps. If the value is "TRUE", then MAXPAGES is the number of pages that are expected to be accessed by each contiguous scan individually. If the value is "FALSE", then MAXPAGES is the number of pages that are expected to be accessed by all the contiguous scans in total.
NUMROWS	INTEGER	Number of rows expected to be sorted.
ONEFETCH	TRUE FALSE	Indicates the GROUP BY conditions are satisfied by the first row produced by the input stream.
OPROFERR	TRUE FALSE	Indicates that one or more errors occurred while parsing or applying the optimization profile. For details, see explain diagnostic messages.

Table 335. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
OUTERCOL	Each row of this type will contain: • Ordinal value of column in order (followed by a colon and a space) • Name of column • Order value **(A)** Ascending **(D)** Descending	Outer order columns.
OUTERJN	LEFT RIGHT FULL LEFT (ANTI) RIGHT (ANTI)	Outer join indicator.
OVERHEAD	DOUBLE	Optimizer used OVERHEAD value.
PARTCOLS	Name of Column	Partitioning columns for operator.
PBLKLOCK	EXCLUSIVE INTENT EXCLUSIVE INTENT NONE INTENT SHARE REUSE SHARE SHARE INTENT EXCLUSIVE SUPER EXCLUSIVE UPDATE	Positioning scan table lock intent.
PGLOLOCK	EXCLUSIVE INTENT EXCLUSIVE INTENT NONE INTENT SHARE REUSE SHARE SHARE INTENT EXCLUSIVE SUPER EXCLUSIVE UPDATE	Positioning scan global table lock intent.
PLANID	Hexadecimal string representing a BIGINT value	Identifier uniquely representing a query plan configuration for a given statement. The layout of the operators, accessed objects and relevant operator arguments and other plan properties affecting performance are represented by this value.
PREFETCH	DYNAMIC LIST LIST NONE READAHEAD SEQUENTIAL SEQUENTIAL, READAHEAD	Type of prefetch eligible.

Table 335. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
PFTCHSZ	INTEGER	Optimizer used PREFETCHSIZE value.
PROWLOCK	EXCLUSIVE NONE REUSE SHARE SHORT (INSTANT) SHARE UPDATE	Positioning scan row lock intent.
PTABLOCK	EXCLUSIVE INTENT EXCLUSIVE INTENT NONE INTENT SHARE REUSE SHARE SHARE INTENT EXCLUSIVE SUPER EXCLUSIVE UPDATE	Positioning scan table lock intent.
RAND ACC	TRUE	Indicates that the regular TEMP table allows random access. Random access is required for the ZZJN operator.
REOPT	ALWAYS ONCE	The statement is optimized using bind-in values for parameter markers, host variables, and special registers.
RMTQTXT	Query text	Remote Query Text
RNG_PROD	Function name	Range producing function for extended index access.
ROWLOCK	EXCLUSIVE NONE REUSE SHARE SHORT (INSTANT) SHARE UPDATE	Row Lock Intent.
ROWWIDTH	INTEGER	Width of row to be sorted.
RSUFFIX	Query text	Remote SQL suffix.
SCANDIR	FORWARD REVERSE	Scan Direction.
SCANGRAN	INTEGER	Intrapartition parallelism, granularity of the intrapartition parallel scan, expressed in SCANUNITs.
SCANTYPE	LOCAL PARALLEL	Intrapartition parallelism, index scan, table scan, or column-organized data scan.
SCANUNIT	ROW PAGE	Intrapartition parallelism, scan granularity unit.
SEMEVID	Hexadecimal string representing a BIGINT value	Identifier for semantic environment at the time the statement was compiled. SEMEVID corresponds to monitoring element SEMANTIC_ENV_ID.

Table 335. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
SEMIJOIN	TRUE FALSE	This argument on the HSJOIN operator indicates whether column-organized processing should deploy the semi-join optimization technique to reduce the size of the hash table that is created for large hash join inner tables.
SHARED	TRUE	Intrapartition parallelism, shared TEMP indicator.
SHRCSE	TRUE	Temporary table over common subexpression shared between subsections.
SKIP_INS	TRUE	Skip inserted. Row access will skip uncommitted inserted rows. This behavior is enabled with the **DB2_SKIPINSERTED** registry variable or when currently committed semantics are in effect.
SKIPDKEY	TRUE	Skip deleted keys. Row access will skip uncommitted deleted keys. This behavior is enabled with the **DB2_SKIPDELETED** registry variable.
SKIPDROW	TRUE	Skip deleted rows. Row access will skip uncommitted deleted rows. This behavior is enabled with the **DB2_SKIPDELETED** registry variable.
SKIPLOCK	TRUE	The concurrent access resolution "skip locked data" is in effect.
SLOWMAT	TRUE FALSE	Slow Materialization flag.
SNGLPROD	TRUE FALSE	Intrapartition parallelism sort or temp produced by a single agent.
SORTKEY	Each row of this type will contain: • Ordinal value of column in key (followed by a colon and a space) • Name of column • Order value **(A)** Ascending **(D)** Descending **(R)** Random	Sort key columns.

Table 335. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
SORTTYPE	GLOBAL MERGE PARTITIONED ROUND ROBIN REPLICATED SHARED	Intrapartition parallelism, sort type.
SRCSEVER	Server name	Source (ship to) server.
SPEED	SLOW FAST	"SLOW" indicates that the scan is expected to progress slowly over the table. For example, if the scan is the outer of a nested loop join). "FAST" indicates that the scan is expected to progress with higher speed. This information is used to group scans together for efficient sharing of bufferpool records.
SPILLED	INTEGER	Estimated number of pages in SORT spill.
SQLCA	Warning information	Warnings and reason codes issued during Explain operation.
STARJOIN	YES	The IXAND operator is part of a star join
STMTHEAP	INTEGER	Size of statement heap at start of statement compile.
STMTID	Hexadecimal string representing a BIGINT value	Identifier uniquely representing a normalized form of the SQL statement. The statement normalization follows the optimization profile's inexact matching rules.
STREAM	TRUE FALSE	Remote source is streaming.
SYSTSENS	YES NO	Indicates that the SYSTIMESENSITIVE bind option is in effect
TABLOCK	EXCLUSIVE INTENT EXCLUSIVE INTENT NONE INTENT SHARE REUSE SHARE SHARE INTENT EXCLUSIVE SUPER EXCLUSIVE UPDATE	Table Lock Intent.
TBISOLVL	UNCOMMITED READ READ STABILITY CURSOR STABILITY REPEATABLE READ	Indicates the isolation level used by the operator to access the specific table
TEMPSIZE	INTEGER	Temporary table page size.

Table 335. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
THROTTLE	TRUE FALSE	Throttling improves the performance of other scans that would otherwise lag behind and be forced to reread the same pages. "TRUE" if the scan can be throttled. "FALSE" if the scan must not be throttled.
TMPCMPRS	YES ELIGIBLE	The value YES indicates that compression is applied. The value ELIGIBLE indicates that compression may be applied if the table becomes large enough. The absence of TMPCMPRS indicates that the temporary table is not compressed.
TQDEGREE	INTEGER	Intrapartition parallelism, number of subagents accessing Table Queue. If the TQ operator represents the transition between column-organized data processing and row-organized data processing, the TQDEGREE argument indicates the number of column-organized processing subagents that are used to process the query in parallel.
TQMERGE	TRUE FALSE	Merging (sorted) Table Queue indicator.
TQNUMBER	INTEGER	Table queue identification number.
TQREAD	READ AHEAD STEPPING SUBQUERY STEPPING	Table Queue reading property.
TQSECNFM	INTEGER	Subsection number at the sending end of the table queue.
TQSECNTO	INTEGER	Subsection number at the receiving end of the table queue.
TQSEND	BROADCAST DIRECTED SCATTER SUBQUERY DIRECTED	Table Queue send property.
TQ TYPE	LOCAL	Intrapartition parallelism, Table Queue.

Table 335. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
TQORIGIN	ASYNCHRONY XTQ COLUMN-ORGANIZED DATA ROW-ORGANIZED DATA	The reason that Table Queue was introduced into the access plan. `COLUMN-ORGANIZED DATA` indicates that the TQ operator is being used to transfer data from column-organized processing to row-organized processing. `ROW-ORGANIZED DATA` indicates that the TQ operator is being used to transfer data from row-organized processing to column-organized processing. `COLUMN-ORGANIZED DATA` or `ROW-ORGANIZED DATA` values result in the TQ operator being displayed as `CTQ` in the access plan.
TRUNCTQ	INPUT OUTPUT INPUT AND OUTPUT	Truncated Table Queue indicator. INPUT indicates that truncation occurs on input to the Table Queue. OUPUT indicates that truncation occurs on output from the Table Queue. INPUT and OUTPUT indicates that truncation occurs on both input to the Table Queue and on output from the Table Queue.
TRUNCSRT	TRUE	Truncated sort (limits number of rows produced).
TUPBLKSZ	INTEGER	Component of the total sort heap required to perform a hash join that determines the number of bytes that a tuple will be stored in. This can be used by service to diagnose memory, temporary table and to some degree sort heap usage.
UNIQUE	TRUE FALSE HASHED PARTIAL HASHED COMPLETE	This operator eliminates rows having duplicate values for a set of columns. HASHED PARTIAL indicates that a partial early distinct operation was performed to efficiently remove many, if not all, duplicates. This reduces the amount of data that must be processed later in the query evaluation. HASHED COMPLETE indicates that hashing is used to eliminate duplicates during column-organized data processing.
UNIQKEY	Each row of this type will contain: • Ordinal value of column in key (followed by a colon and a space) • Name of Column	Unique key columns.

Table 335. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
UR_EXTRA	TRUE	Uncommitted read isolation, but with extra processing to ensure correct isolation. This access has extra table level locking; the same table level locking as cursor stability. Also, when the statement is executing, the isolation level might upgrade to cursor stability, for example, if an online load is running concurrently. Another part of the statement execution plan will ensure the isolation level is correct, such as a FETCH operator at a higher isolation level.
USAGE	SCALAR_SUBQUERY	Indicates how the NLJOIN operator is being used. SCALAR_SUBQUERY indicates that the NLJOIN operator is being used to apply a scalar subquery predicate. Hash join is the only join method used for column-organized tables. However, the optimizer uses the NLJOIN operator to model the application of scalar subqueries during column-organized data processing. One of the input legs to the NLJOIN operator is the single value that is computed by the scalar subquery. The other input to the operator is the data stream whose predicate references the scalar value.
VISIBLE	TRUE FALSE	Whether shared scans are visible to other shared scans. A shared scan that is visible can influence the behavior of other scans. Examples of affected behavior include start location and throttling.
VOLATILE	TRUE	Volatile table
WRAPPING	TRUE FALSE	Whether a shared scan is allowed to start at any record in the table and wrap once it reaches the last record. Wrapping allows bufferpool records to be shared with other ongoing scans.
XFERRATE	DOUBLE	Optimizer used TRANSFERRATE value.
XDFOUT	DECIMAL	XDFOUT indicates the expected number of documents to be returned by the XISCAN operator for each context node.
XLOGID	An identifier consisting of an SQL schema name and the name of an index over XML data	XLOGID identifies the index over XML data chosen by the optimizer for the XISCAN operator.

Table 335. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
XPATH	An XPATH expression and result set in an internal format	This argument indicates the evaluation of an XPATH expression by the XSCAN operator.
XPHYID	An identifier consisting of an SQL schema name and the name of a physical index over XML data	XPHYID identifies the physical index that is associated with an index over XML data used by the XISCAN operator.

EXPLAIN_DIAGNOSTIC table

The EXPLAIN_DIAGNOSTIC table contains an entry for each diagnostic message produced for a particular instance of an explained statement in the EXPLAIN_STATEMENT table.

The EXPLAIN_GET_MSGS table function queries the EXPLAIN_DIAGNOSTIC and EXPLAIN_DIAGNOSTIC_DATA Explain tables and returns formatted messages.

Table 336. EXPLAIN_DIAGNOSTIC Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	PK, FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	PK, FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	PK, FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	PK, FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	PK, FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	PK, FK	Level of Explain information for which this row is relevant. Valid values are: **O** Original text (as entered by user) **P** PLAN SELECTION
STMTNO	INTEGER	No	PK, FK	Statement number within package to which this Explain information is related. Set to 1 for dynamic Explain SQL statements. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS catalog view.

Table 336. EXPLAIN_DIAGNOSTIC Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable?	Key?	Description
SECTNO	INTEGER	No	PK, FK	Section number within package that contains this SQL statement. For dynamic Explain SQL statements, this is the section number used to hold the section for this statement at run time. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS catalog view.
DIAGNOSTIC_ID	INTEGER	No	PK	ID of the diagnostic for a particular instance of a statement in the EXPLAIN_STATEMENT table.
CODE	INTEGER	No	No	A unique number assigned to each diagnostic message. The number can be used by a message API to retrieve the full text of the diagnostic message.

EXPLAIN_DIAGNOSTIC_DATA table

The EXPLAIN_DIAGNOSTIC_DATA table contains message tokens for specific diagnostic messages that are recorded in the EXPLAIN_DIAGNOSTIC table. The message tokens provide additional information that is specific to the execution of the SQL statement that generated the message.

The EXPLAIN_GET_MSGS table function queries the EXPLAIN_DIAGNOSTIC and EXPLAIN_DIAGNOSTIC_DATA Explain tables, and returns formatted messages.

Table 337. EXPLAIN_DIAGNOSTIC_DATA Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant. Valid values are: **O** Original text (as entered by user) **P** PLAN SELECTION

Table 337. EXPLAIN_DIAGNOSTIC_DATA Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable?	Key?	Description
STMTNO	INTEGER	No	FK	Statement number within package to which this Explain information is related. Set to 1 for dynamic Explain SQL statements. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS catalog view.
SECTNO	INTEGER	No	FK	Section number within package that contains this SQL statement. For dynamic Explain SQL statements, this is the section number used to hold the section for this statement at run time. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS catalog view.
DIAGNOSTIC_ID	INTEGER	No	PK	ID of the diagnostic for a particular instance of a statement in the EXPLAIN_STATEMENT table.
ORDINAL	INTEGER	No	No	Position of token in the full message text.
TOKEN	VARCHAR(1000)	Yes	No	Message token to be inserted into the full message text; might be truncated.
TOKEN_LONG	BLOB(3M)	Yes	No	More detailed information, if available.

EXPLAIN_INSTANCE table

The EXPLAIN_INSTANCE table is the main control table for all Explain information. Each row of data in the Explain tables is explicitly linked to one unique row in this table.

The EXPLAIN_INSTANCE table gives basic information about the source of the SQL statements being explained as well as information about the environment in which the explanation took place.

Table 338. EXPLAIN_INSTANCE Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	PK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	PK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	PK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	PK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	PK	Version of the source of the Explain request.

Table 338. EXPLAIN_INSTANCE Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_OPTION	CHAR(1)	No	No	Indicates what Explain Information was requested for this request. Possible values are: **P** PLAN SELECTION **S** Section Explain
SNAPSHOT_TAKEN	CHAR(1)	No	No	Indicates whether an Explain Snapshot was taken for this request. Possible values are: **Y** Yes, an Explain Snapshot(s) was taken and stored in the EXPLAIN_STATEMENT table. Regular Explain information was also captured. **N** No Explain Snapshot was taken. Regular Explain information was captured. **O** Only an Explain Snapshot was taken. Regular Explain information was not captured.
DB2_VERSION	CHAR(7)	No	No	Release number for the Db2 product that processed this explain request. Format is: VV.RR.M., where: **vv** Version number **rr** Release number **m** Modification pack To see the complete product signature information for an EXPLAIN instance, review BLDLEVEL argument row recorded in the "EXPLAIN_ARGUMENT table" on page 2174.
SQL_TYPE	CHAR(1)	No	No	Indicates whether the Explain Instance was for static or dynamic SQL. Possible values are: **S** Static SQL **D** Dynamic SQL

Table 338. EXPLAIN_INSTANCE Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable?	Key?	Description
QUERYOPT	INTEGER	No	No	Indicates the query optimization class used by the SQL Compiler at the time of the Explain invocation. The value indicates what level of query optimization was performed by the SQL Compiler for the SQL statements being explained.
BLOCK	CHAR(1)	No	No	Indicates what type of cursor blocking was used when compiling the SQL statements. Possible values are: **N** No Blocking **U** Block Unambiguous Cursors **B** Block All Cursors
ISOLATION	CHAR(2)	No	No	Indicates what type of isolation was used when compiling the SQL statements. Possible values are: **RR** Repeatable Read **RS** Read Stability **CS** Cursor Stability **UR** Uncommitted Read
BUFFPAGE	INTEGER	No	No	Contains the value of the **buffpage** database configuration setting at the time of the Explain invocation. **Important:** The **buffpage** database configuration is deprecated and might be remove in a future release.
AVG_APPLS	INTEGER	No	No	Contains the value of the **avg_appls** configuration parameter at the time of the Explain invocation.
SORTHEAP	INTEGER	No	No	Contains the value of the **sortheap** database configuration parameter at the time of the Explain invocation.
LOCKLIST	INTEGER	No	No	Contains the value of the **locklist** database configuration parameter at the time of the Explain invocation.

Table 338. EXPLAIN_INSTANCE Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable ?	Key?	Description
MAXLOCKS	SMALLINT	No	No	Contains the value of the **maxlocks** database configuration parameter at the time of the Explain invocation.
LOCKS_AVAIL	INTEGER	No	No	Contains the number of locks assumed to be available by the optimizer for each user. (Derived from **locklist** and **maxlocks**.)
CPU_SPEED	DOUBLE	No	No	Contains the value of the **cpuspeed** database manager configuration parameter at the time of the Explain invocation.
REMARKS	VARCHAR(254)	Yes	No	User-provided comment.
DBHEAP	INTEGER	No	No	Contains the value of the **dbheap** database configuration parameter at the time of Explain invocation.
COMM_SPEED	DOUBLE	No	No	Contains the value of the **comm_bandwidth** database configuration parameter at the time of Explain invocation.
PARALLELISM	CHAR(2)	No	No	Possible values are: • N = No parallelism • P = Intrapartition parallelism • IP = Interpartition parallelism • BP = Intrapartition parallelism and interpartition parallelism
DATAJOINER	CHAR(1)	No	No	Possible values are: • N = Non-federated systems plan • Y = Federated systems plan
EXECUTABLE_ID	VARCHAR(32) FOR BIT DATA	Yes	No	A binary token generated on the data server that uniquely identifies the SQL statement section that was executed.
EXECUTION_TIME	TIMESTAMP	Yes	No	Time the section started execution.

EXPLAIN_OBJECT table

The EXPLAIN_OBJECT table identifies those data objects required by the access plan generated to satisfy the SQL statement.

Table 339. EXPLAIN_OBJECT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable ?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.

Table 339. EXPLAIN_OBJECT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable ?	Key?	Description
SOURCE_NAME	VARCHAR(128)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	FK	Statement number within package to which this explain information is related.
SECTNO	INTEGER	No	FK	Section number within package to which this explain information is related.
OBJECT_SCHEMA	VARCHAR(128)	No	No	Schema to which this object belongs.
OBJECT_NAME	VARCHAR(128)	No	No	Name of the object.
OBJECT_TYPE	CHAR(2)	No	No	Descriptive label for the type of object.
CREATE_TIME	TIMESTAMP	Yes	No	Time of Object's creation; null if a table function.
STATISTICS_TIME	TIMESTAMP	Yes	No	Last time of update to statistics for this object; null if statistics do not exist for this object.
COLUMN_COUNT	SMALLINT	No	No	Number of columns in this object.
ROW_COUNT	INTEGER	No	No	Estimated number of rows in this object.
WIDTH	INTEGER	No	No	The average width of the object in bytes. Set to -1 for an index.
PAGES	BIGINT	No	No	Estimated number of pages that the object occupies in the buffer pool. Set to -1 for a table function.
DISTINCT	CHAR(1)	No	No	Indicates whether the rows in the object are distinct (that is, whether there are duplicates). Possible values are: **Y** Yes **N** No
TABLESPACE_NAME	VARCHAR(128)	Yes	No	Name of the table space in which this object is stored; set to null if no table space is involved.

Table 339. EXPLAIN_OBJECT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable ?	Key?	Description
OVERHEAD	DOUBLE	No	No	Total estimated overhead, in milliseconds, for a single random I/O to the specified table space. Includes controller overhead, disk seek, and latency times. Set to -1 if: • no table space is involved • a partitioned object is involved and the respective table spaces for the partitions have different values • a partitioned object is involved and the EXPLAIN_OBJECT table is populated using section explain
TRANSFER_RATE	DOUBLE	No	No	Estimated time to read a data page, in milliseconds, from the specified table space. Set to -1 if: • no table space is involved • a partitioned object is involved and the respective table spaces for the partitions have different values • a partitioned object is involved and the EXPLAIN_OBJECT table is populated using section explain
PREFETCHSIZE	INTEGER	No	No	Number of data pages to be read when prefetch is performed. Set to -1 if: • no table space is involved • a partitioned object is involved and the respective table spaces for the partitions have different values • a partitioned object is involved and the EXPLAIN_OBJECT table is populated using section explain
EXTENTSIZE	INTEGER	No	No	Size of extent, in data pages. This many pages are written to one container in the table space before switching to the next container. Set to -1 for a table function.
CLUSTER	DOUBLE	No	No	Degree of data clustering with the index. If >= 1, this is the CLUSTERRATIO. If >= 0 and < 1, this is the CLUSTERFACTOR. Set to -1 for a table, table function, or if this statistic is not available.
NLEAF	BIGINT	No	No	Number of leaf pages this index object's values occupy. Set to -1 for a table, table function, or if this statistic is not available.
NLEVELS	INTEGER	No	No	Number of index levels in this index object's tree. Set to -1 for a table, table function, or if this statistic is not available.

Table 339. EXPLAIN_OBJECT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable ?	Key?	Description
FULLKEYCARD	BIGINT	No	No	Number of distinct full key values contained in this index object. Set to -1 for a table, table function, or if this statistic is not available.
OVERFLOW	BIGINT	No	No	Total number of overflow records in the table. Set to -1 for an index, table function, or if this statistic is not available.
FIRSTKEYCARD	BIGINT	No	No	Number of distinct first key values. Set to -1 for a table, table function, or if this statistic is not available.
FIRST2KEYCARD	BIGINT	No	No	Number of distinct first key values using the first 2 columns of the index. Set to -1 for a table, table function, or if this statistic is not available.
FIRST3KEYCARD	BIGINT	No	No	Number of distinct first key values using the first 3 columns of the index. Set to -1 for a table, table function, or if this statistic is not available.
FIRST4KEYCARD	BIGINT	No	No	Number of distinct first key values using the first 4 columns of the index. Set to -1 for a table, table function, or if this statistic is not available.
SEQUENTIAL_PAGES	BIGINT	No	No	Number of leaf pages located on disk in index key order with few or no large gaps between them. Set to -1 for a table, table function, or if this statistic is not available.
DENSITY	INTEGER	No	No	Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percentage (integer between 0 and 100). Set to -1 for a table, table function, or if this statistic is not available.
STATS_SRC	CHAR(1)	No	No	Indicates the source for the statistics. Set to 1 if from single node.
AVERAGE_SEQUENCE_GAP	DOUBLE	No	No	Gap between sequences.
AVERAGE_SEQUENCE_FETCH_GAP	DOUBLE	No	No	Gap between sequences when fetching using the index.
AVERAGE_SEQUENCE_PAGES	DOUBLE	No	No	Average number of index pages accessible in sequence.
AVERAGE_SEQUENCE_FETCH_PAGES	DOUBLE	No	No	Average number of table pages accessible in sequence when fetching using the index.
AVERAGE_RANDOM_PAGES	DOUBLE	No	No	Average number of random index pages between sequential page accesses.
AVERAGE_RANDOM_FETCH_PAGES	DOUBLE	No	No	Average number of random table pages between sequential page accesses when fetching using the index.
NUMRIDS	BIGINT	No	No	Total number of row identifiers in the index.

Table 339. EXPLAIN_OBJECT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable?	Key?	Description
NUMRIDS_DELETED	BIGINT	No	No	Total number of psuedo-deleted row identifiers in the index.
NUM_EMPTY_LEAFS	BIGINT	No	No	Total number of empty leaf pages in the index.
ACTIVE_BLOCKS	BIGINT	No	No	Total number of active multidimensional clustering (MDC) blocks in the table.
NUM_DATA_PART	INTEGER	No	No	Number of data partitions for a partitioned table. Set to 1 if the table is not partitioned.
NULLKEYS	CHAR(1)	Yes	No	Specifies whether null keys are indexed. • N = Keys that contain all null values are not indexed (not considering columns or expressions from the INCLUDE clause) • Y = Keys that contain all null values are indexed (not considering columns or expressions from the INCLUDE clause) A null value indicates that this object is not an index.
OBJECT_TENANTID	INTEGER	No	No	ID for tenant where object is defined.

Table 340. Possible OBJECT_TYPE Values

Value	Description
IX	Index
NK	Nickname
RX	RCT index
DP	Data partitioned table
TA	Table
TF	Table function
+A	Compiler-referenced alias
+C	Compiler-referenced constraint
+F	Compiler-referenced function
+G	Compiler-referenced trigger
+N	Compiler-referenced nickname
+T	Compiler-referenced table
+V	Compiler-referenced view
XI	Logical XML index
PI	Physical XML index
LI	Partitioned index

Table 340. Possible OBJECT_TYPE Values (continued)

Value	Description
LX	Partitioned logical XML index
LP	Partitioned physical XML index
CO	Column-organized table

EXPLAIN_OPERATOR table

The EXPLAIN_OPERATOR table contains all the operators needed to satisfy the query statement by the query compiler.

Table 341. EXPLAIN_OPERATOR Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable ?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	PK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	PK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	PK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	PK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	PK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	PK	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	PK	Statement number within package to which this explain information is related.
SECTNO	INTEGER	No	PK	Section number within package to which this explain information is related.
OPERATOR_ID	INTEGER	No	PK	Unique ID for this operator within this query.
OPERATOR_TYPE	CHAR(6)	No	No	Descriptive label for the type of operator.
TOTAL_COST	DOUBLE	No	No	Estimated cumulative total cost (in timerons) of executing the chosen access plan up to and including this operator.
IO_COST	DOUBLE	No	No	Estimated cumulative I/O cost (in data page I/Os) of executing the chosen access plan up to and including this operator.
CPU_COST	DOUBLE	No	No	Estimated cumulative CPU cost (in instructions) of executing the chosen access plan up to and including this operator.
FIRST_ROW_COST	DOUBLE	No	No	Estimated cumulative cost (in timerons) of fetching the first row for the access plan up to and including this operator. This value includes any initial overhead required.

Table 341. EXPLAIN_OPERATOR Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable?	Key?	Description
RE_TOTAL_COST	DOUBLE	No	No	Estimated cumulative cost (in timerons) of fetching the next row for the chosen access plan up to and including this operator.
RE_IO_COST	DOUBLE	No	No	Estimated cumulative I/O cost (in data page I/Os) of fetching the next row for the chosen access plan up to and including this operator.
RE_CPU_COST	DOUBLE	No	No	Estimated cumulative CPU cost (in instructions) of fetching the next row for the chosen access plan up to and including this operator.
COMM_COST	DOUBLE	No	No	Estimated cumulative communication cost (in TCP/IP frames) of network traffic flowing across a single network adapter when executing the chosen access plan up to and including this operator. (See notes 1 and 2.)
FIRST_COMM_COST	DOUBLE	No	No	Estimated cumulative communications cost (in TCP/IP frames) of network traffic flowing across a single network adapter when fetching the first row for the chosen access plan up to and including this operator. This value includes any initial overhead required. (See notes 1 and 2.)
BUFFERS	DOUBLE	No	No	Estimated buffer requirements for this operator and its inputs.
REMOTE_TOTAL_COST	DOUBLE	No	No	Estimated cumulative total cost (in timerons) of performing operation(s) on remote database(s).
REMOTE_COMM_COST	DOUBLE	No	No	Estimated cumulative communication cost of executing the chosen remote access plan up to and including this operator.

Note:

1. If there is more than one network adapter involved, the cumulative communication cost for the adapter with the highest value is returned.
2. This value only includes the costs of network traffic between physical machines. It does not include the virtual communication costs between node partitions on the same physical machine in a partitioned database environment.

Table 342. OPERATOR_TYPE values

Value	Description
CMPEXP	Represent the projection of columns and expression from a subselect in the optimized version of an SQL statement
DELETE	Delete
EISCAN	Extended Index Scan
FETCH	Fetch
FILTER	Filter rows

Table 342. OPERATOR_TYPE values (continued)

Value	Description
GENROW	Generate Row
GRPBY	Group By
HSJOIN	Hash Join
INSERT	Insert
IXAND	Dynamic Bitmap Index ANDing
IXSCAN	Relational index scan
MSJOIN	Merge Scan Join
NLJOIN	Nested loop Join
REBAL	Rebalance rows between SMP subagents
RETURN	Result
RIDSCN	Row Identifier (RID) Scan
RPD	Remote PushDown
SHIP	Ship query to remote system
SORT	Sort
TBFUNC	In-stream table function operator
TBSCAN	Table Scan
TEMP	Temporary Table Construction
TQ	Table Queue
UNION	Union
UNIQUE	Duplicate Elimination
UPDATE	Update
XISCAN	Index scan over XML data
XSCAN	XML document navigation scan
XANDOR	Index ANDing and ORing over XML data
ZZJOIN	Zigzag join

EXPLAIN_PREDICATE table

The EXPLAIN_PREDICATE table identifies which predicates are applied by a specific operator.

Table 343. EXPLAIN_PREDICATE Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable ?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.

Table 343. EXPLAIN_PREDICATE Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable?	Key?	Description
SOURCE_NAME	VARCHAR(128)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	FK	Statement number within package to which this explain information is related.
SECTNO	INTEGER	No	FK	Section number within package to which this explain information is related.
OPERATOR_ID	INTEGER	No	No	Unique ID for this operator within this query.
PREDICATE_ID	INTEGER	No	No	Unique ID for this predicate for the specified operator. A value of "-1" is shown for operator predicates constructed by the Explain tool which are not optimizer objects and do not exist in the optimizer plan.
HOW_APPLIED	CHAR(10)	No	No	How predicate is being used by the specified operator.
WHEN_EVALUATED	CHAR(3)	No	No	Indicates when the subquery used in this predicate is evaluated. Possible values are: **blank** This predicate does not contain a subquery. **EAA** The subquery used in this predicate is evaluated at application (EAA). That is, it is re-evaluated for every row processed by the specified operator, as the predicate is being applied. **EAO** The subquery used in this predicate is evaluated at open (EAO). That is, it is re-evaluated only once for the specified operator, and its results are re-used in the application of the predicate for each row. **MUL** There is more than one type of subquery in this predicate.
RELOP_TYPE	CHAR(2)	No	No	The type of relational operator used in this predicate.

Table 343. EXPLAIN_PREDICATE Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable?	Key?	Description
SUBQUERY	CHAR(1)	No	No	Whether or not a data stream from a subquery is required for this predicate. There may be multiple subquery streams required. Possible values are: **N** No subquery stream is required **Y** One or more subquery streams is required
FILTER_FACTOR	DOUBLE	No	No	The estimated fraction of rows that will be qualified by this predicate. A value of "-1" is shown when FILTER_FACTOR is not applicable. FILTER_FACTOR is not applicable for operator predicates constructed by the Explain tool which are not optimizer objects and do not exist in the optimizer plan.
PREDICATE_TEXT	CLOB(2M)	Yes	No	The text of the predicate as recreated from the internal representation of the SQL or XQuery statement. If the value of a host variable, special register, or parameter marker is used during compilation of the statement, this value will appear at the end of the predicate text enclosed in a comment. The value will be stored in the EXPLAIN_PREDICATE table only if the statement is executed by a user who has DBADM authority, or if the DB2_VIEW_REOPT_VALUES registry variable is set to YES; otherwise, an empty comment will appear at the end of the predicate text. Null if not available.
RANGE_NUM	INTEGER	Yes	No	Range of data partition elimination predicates, which enables the grouping of predicates that are used for data partition elimination by range. Null value for all other predicate types.
INDEX_COLSEQ	INTEGER	No	No	Indicates the index column that the predicate belongs to if it is part of a key predicate. A key predicate always belongs to one index key part. A value of "-1" is shown for predicates that are not part of a key predicate.

Table 344. Possible HOW_APPLIED Values

Value	Description
BIT_FLTR	Predicate is applied as a bit filter

Table 344. Possible HOW_APPLIED Values (continued)

Value	Description
BSARG	Evaluated as a sargable predicate once for every block
DPSTART	Start key predicate used in data partition elimination
DPSTOP	Stop key predicate used in data partition elimination
ESARG	Evaluated as a sargable predicate by external reader.
JOIN	Used to join tables
RANGE_FLTR	Predicate is applied as a range filter
RESID	Evaluated as a residual predicate
SARG	Evaluated as a sargable predicate for index or data page
GAP_START	Used as a start condition on an index gap
GAP_STOP	Used as a stop condition on an index gap
START	Used as a start condition
STOP	Used as a stop condition
FEEDBACK	Zigzag join feedback predicate

Table 345. Possible RELOP_TYPE Values

Value	Description
blanks	Not Applicable
EQ	Equals
GE	Greater Than or Equal
GT	Greater Than
IN	In list
IC	In list, sorted during query optimization
IR	In list, sorted at runtime
LE	Less Than or Equal
LK	Like
LT	Less Than
NE	Not Equal
NL	Is Null
NN	Is Not Null
RE	REGEXP_LIKE

EXPLAIN_STATEMENT table

The EXPLAIN_STATEMENT table contains the text of the SQL statement as it exists for the different levels of Explain information.

The original SQL statement as entered by the user is stored in this table along with the version used (by the optimizer) to choose an access plan to satisfy the SQL statement. The latter version may bear

little resemblance to the original as it may have been rewritten or enhanced with additional predicates as determined by the SQL Compiler. In addition, if statement concentrator is enabled and the statement was changed as a result of statement concentrator, the effective SQL statement will also be stored in this table. This statement will resemble the original statement except that the literal values will be replaced with system generated named parameter markers. The plan information will be based on the effective statement in this case.

Table 346. EXPLAIN_STATEMENT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	PK, FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	PK, FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	PK, FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	PK, FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	PK, FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	PK	Level of Explain information for which this row is relevant. Valid values are: **E** Effective SQL text **F** Statement with row and column access control applied, before optimization **O** Original Text (as entered by user) **P** PLAN SELECTION **S** Section Explain
STMTNO	INTEGER	No	PK	Statement number within package to which this explain information is related. Set to 1 for dynamic Explain SQL statements. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS catalog view.
SECTNO	INTEGER	No	PK	Section number within package that contains this SQL statement. For dynamic Explain SQL statements, this is the section number used to hold the section for this statement at runtime. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS catalog view.

Table 346. EXPLAIN_STATEMENT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable?	Key?	Description
QUERYNO	INTEGER	No	No	Numeric identifier for explained SQL statement. For dynamic SQL statements (excluding the EXPLAIN SQL statement) issued through CLP or CLI, the default value is a sequentially incremented value. Otherwise, the default value is the value of STMTNO for static SQL statements and 1 for dynamic SQL statements.
QUERYTAG	CHAR(20)	No	No	Identifier tag for each explained SQL statement. For dynamic SQL statements issued through CLP (excluding the EXPLAIN SQL statement), the default value is "CLP". For dynamic SQL statements issued through CLI (excluding the EXPLAIN SQL statement), the default value is "CLI". Otherwise, the default value used is blanks.
STATEMENT_TYPE	CHAR(2)	No	No	Descriptive label for type of query being explained. Possible values are: **CL** Call **CP** Compound SQL (Dynamic) **D** Delete **DC** Delete where current of cursor **I** Insert **M** Merge **S** Select **SI** Set Integrity or Refresh Table **U** Update **UC** Update where current of cursor

Table 346. EXPLAIN_STATEMENT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable?	Key?	Description
UPDATABLE	CHAR(1)	No	No	Indicates if this statement is considered updatable. This is particularly relevant to SELECT statements which may be determined to be potentially updatable. Possible values are: **' '** Not applicable (blank) **N** No **Y** Yes
DELETABLE	CHAR(1)	No	No	Indicates if this statement is considered deletable. This is particularly relevant to SELECT statements which may be determined to be potentially deletable. Possible values are: **' '** Not applicable (blank) **N** No **Y** Yes
TOTAL_COST	DOUBLE	No	No	Estimated total cost (in timerons) of executing the chosen access plan for this statement; set to 0 (zero) if EXPLAIN_LEVEL is 0 or E (original or effective text) since no access plan has been chosen at this time.
STATEMENT_TEXT	CLOB(2M)	No	No	Text or portion of the text of the SQL statement being explained. The text shown for the Plan Selection or Section Explain levels of Explain has been reconstructed from the internal representation and is SQL-like in nature; that is, the reconstructed statement is not guaranteed to follow correct SQL syntax.
SNAPSHOT	BLOB(10M)	Yes	No	Snapshot of internal representation for this SQL statement at the Explain_Level shown. Column is set to NULL if EXPLAIN_LEVEL is not P (Plan Selection) since no access plan has been chosen at the time that this specific version of the statement is captured.

Table 346. EXPLAIN_STATEMENT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable?	Key?	Description
QUERY_DEGREE	INTEGER	No	No	Indicates the degree of intrapartition parallelism at the time of Explain invocation. For the original statement, this contains the directed degree of intrapartition parallelism. Otherwise, this contains the degree of intrapartition parallelism generated for the plan to use.

EXPLAIN_STREAM table

The EXPLAIN_STREAM table represents the input and output data streams between individual operators and data objects. The data objects themselves are represented in the EXPLAIN_OBJECT table. The operators involved in a data stream are to be found in the EXPLAIN_OPERATOR table.

Table 347. EXPLAIN_STREAM Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	FK	Statement number within package to which this explain information is related.
SECTNO	INTEGER	No	FK	Section number within package to which this explain information is related.
STREAM_ID	INTEGER	No	No	Unique ID for this data stream within the specified operator.
SOURCE_TYPE	CHAR(1)	No	No	Indicates the source of this data stream: **O** Operator **D** Data Object
SOURCE_ID	SMALLINT	No	No	Unique ID for the operator within this query that is the source of this data stream. Set to -1 if SOURCE_TYPE is 'D'.

Table 347. EXPLAIN_STREAM Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable ?	Key?	Description
TARGET_TYPE	CHAR(1)	No	No	Indicates the target of this data stream: **O** Operator **D** Data Object
TARGET_ID	SMALLINT	No	No	Unique ID for the operator within this query that is the target of this data stream. Set to -1 if TARGET_TYPE is 'D'.
OBJECT_SCHEMA	VARCHAR(128)	Yes	No	Schema to which the affected data object belongs. Set to null if both SOURCE_TYPE and TARGET_TYPE are 'O'.
OBJECT_NAME	VARCHAR(128)	Yes	No	Name of the object that is the subject of data stream. Set to null if both SOURCE_TYPE and TARGET_TYPE are 'O'.
STREAM_COUNT	DOUBLE	No	No	Estimated cardinality of data stream.
COLUMN_COUNT	SMALLINT	No	No	Number of columns in data stream.
PREDICATE_ID	INTEGER	No	No	If this stream is part of a subquery for a predicate, the predicate ID will be reflected here, otherwise the column is set to -1.
COLUMN_NAMES	CLOB(2M)	Yes	No	This column contains the names and ordering information of the columns involved in this stream. These names will be in the format of: `NAME1(A) + NAME2(D) + NAME3(E) + NAME4` Where *(A)* indicates a column in ascending order, *(D)* indicates a column in descending order, and no ordering information indicates that either the column is not ordered or ordering is not relevant. *(E)* indicates a column in a column-organized table which is encoded as a result of compression, and can occur with or without ordering information.
PMID	SMALLINT	No	No	Distribution map ID.

Table 347. EXPLAIN_STREAM Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key. *(continued)*

Column Name	Data Type	Nullable?	Key?	Description
SINGLE_NODE	CHAR(5)	Yes	No	Indicates whether this data stream is on a single or on multiple database partitions: **MULT** On multiple database partitions **COOR** On coordinator node **HASH** Directed using hashing **RID** Directed using the row ID **FUNC** Directed using a function (HASHEDVALUE() or DBPARTITIONNUM()) **CORR** Directed using a correlation value **Numeric** Directed to predetermined single node
PARTITION_COLUMNS	CLOB(2M)	Yes	No	List of the columns on which this data stream is distributed.
SEQUENCE_SIZES	CLOB(2M)	Yes	No	Lists the expected sequence size for XML columns, or shows "NA" (not applicable) for any non-XML columns in the data stream. Set to null if there is not at least one XML column in the data stream.
OBJECT_TENANTID	INTEGER	Yes	No	ID for tenant where object is defined.

OBJECT_METRICS table

The OBJECT_METRICS table contains runtime statistics collected for each object referenced in a specific execution of a section (identified by executable ID) at a specific time (identified by execution time).

If object statistics are collected on multiple database members, there will be a row per database member for each object. If it is a partitioned object, then there will be one row per data partition.

The OBJECT_METRICS table is populated only if section actuals were captured by the activity event monitor.

Table 348. OBJECT_METRICS table

Column Name	Data Type	Nullable?	Key?	Description
EXECUTABLE_ID	VARCHAR(32) FOR BIT DATA	No	PK	A binary token generated on the data server that uniquely identifies the SQL statement section that was executed.
EXECUTION_TIME	TIMESTAMP	No	PK	Time the section started execution.
OBJECT_SCHEMA	VARCHAR(128)	No	PK	Schema to which this object belongs.

Table 348. OBJECT_METRICS table (continued)

Column Name	Data Type	Nullable ?	Key?	Description
OBJECT_NAME	VARCHAR(128)	No	PK	Name of the object.
OBJECT_TYPE	CHAR(2)	No	PK	Descriptive label for the type of object. Possible values: **IX** Index **DP** Data partitioned table **TA** Table **PI** Physical XML Index **LI** Partitioned index **LP** Partitioned physical XML index
OBJECT_TENANTID	INTEGER	No	PK	ID for tenant where object is defined.
MEMBER	SMALLINT	No	PK	The database member where the object statistics is collected.
DATA_PARTITION_ID	INTEGER	No	PK	The identifier of the data partition for which information is returned . This element is applicable only to partitioned tables or indexes.
ROWS_READ	BIGINT	Yes	No	The total number of rows read.
ROWS_INSERTED	BIGINT	Yes	No	The total number of row insertions attempted.
ROWS_UPDATED	BIGINT	Yes	No	The total number of row updates attempted.
ROWS_DELETED	BIGINT	Yes	No	The total number of row deletions attempted.
OVERFLOW_CREATES	BIGINT	Yes	No	The number of overflowed rows created on this table object.
OVERFLOW_ACCESSES	BIGINT	Yes	No	The number of accesses (reads and writes) to overflowed rows of this table object.
LOCK_WAIT_TIME	BIGINT	Yes	No	The total elapsed time spent waiting for local locks. The value is given in milliseconds.
LOCK_WAIT_TIME_ GLOBAL	BIGINT	Yes	No	The total elapsed time spent waiting for global locks. The value is given in milliseconds.
LOCK_WAITS	BIGINT	Yes	No	The total number of times that the section waited on locks.
LOCK_WAITS_ GLOBAL	BIGINT	Yes	No	The total number of times that the section waited on global locks.
LOCK_ESCALS	BIGINT	Yes	No	The total number of times that local locks have been escalated.
LOCK_ESCALS_ GLOBAL	BIGINT	Yes	No	The total number of times that global locks have been escalated.

Table 348. OBJECT_METRICS table (continued)

Column Name	Data Type	Nullable?	Key?	Description
DIRECT_WRITES	BIGINT	Yes	No	The total number of write operations that do not use a buffer pool.
DIRECT_WRITE_REQS	BIGINT	Yes	No	The total number of requests to perform a direct write of one or more sectors of data.
DIRECT_READS	BIGINT	Yes	No	The total number of read operations that do not use a buffer pool.
DIRECT_READ_REQS	BIGINT	Yes	No	The total number of requests to perform a direct read of one or more sectors of data.
OBJECT_DATA_L_READS	BIGINT	Yes	No	Indicates the number of data pages which have been requested for the table (logical).
OBJECT_DATA_P_READS	BIGINT	Yes	No	Indicates the number of data pages read in for the table (physical).
OBJECT_DATA_GBP_L_READS	BIGINT	Yes	No	The number of times a Group Buffer Pool (GBP) dependent data page was attempted to be read for the table from the group buffer pool because the page was either invalid or not present in the Local Buffer Pool (LBP).
OBJECT_DATA_GBP_P_READS	BIGINT	Yes	No	The number of times a Group Buffer Pool (GBP) dependent data page was read for the table into the local buffer pool from disk because it was not found in the group buffer pool.
OBJECT_DATA_GBP_INVALID_PAGES	BIGINT	Yes	No	The number of times a data page for XML Storage (XDAs) was attempted to be read for the table from the group buffer pool because the page was invalid in the local buffer pool.
OBJECT_DATA_LBP_PAGES_FOUND	BIGINT	Yes	No	The number of times a data page for the table was present in the local buffer pool.
OBJECT_DATA_GBP_INDEP_PAGES_FOUND_IN_LBP	BIGINT	Yes	No	The number of group buffer pool (GBP) independent data pages found in the local buffer pool (LBP) by the agent.
OBJECT_XDA_L_READS	BIGINT	Yes	No	Indicates the number of data pages for XML Storage (XDAs) which have been requested for the table (logical).
OBJECT_XDA_P_READS	BIGINT	Yes	No	Indicates the number of data pages for XML storage (XDAs) read in for the table (physical).
OBJECT_XDA_GBP_L_READS	BIGINT	Yes	No	The number of times a Group Buffer Pool (GBP) dependent data page for XML Storage (XDAs) was attempted to be read for the table from the group buffer pool because the page was either invalid or not present in the Local Buffer Pool (LBP).
OBJECT_XDA_GBP_P_READS	BIGINT	Yes	No	The number of times a Group Buffer Pool (GBP) dependent data page for XML Storage (XDAs) was read for the table into the local buffer pool from disk because it was not found in the GBP.

Table 348. OBJECT_METRICS table (continued)

Column Name	Data Type	Nullable ?	Key?	Description
OBJECT_XDA_ GBP_INVALID_PAGES	BIGINT	Yes	No	The number of times a data page was attempted to be read for the table from the group buffer pool because the page was invalid in the local buffer pool.
OBJECT_XDA_ LBP_PAGES_FOUND	BIGINT	Yes	No	The number of times a XML Storage (XDAs) data page for the table was present in the local buffer pool.
OBJECT_XDA_ GBP_INDEP_PAGES_ FOUND_IN_LBP	BIGINT	Yes	No	The number of group buffer pool (GBP) independent XML storage object (XDA) data pages found in the local buffer pool (LBP) by the agent.
OBJECT_INDEX_ L_READS	BIGINT	Yes	No	Indicates the number of index pages which have been requested for the index (logical).
OBJECT_INDEX_ P_READS	BIGINT	Yes	No	Indicates the number of index pages read in for the index (physical).
OBJECT_INDEX_ GBP_L_READS	BIGINT	Yes	No	The number of times a Group Buffer Pool (GBP) dependent index page was attempted to be read for the index from the group buffer pool because the page was either invalid or not present in the Local Buffer Pool (LBP).
OBJECT_INDEX_ GBP_P_READS	BIGINT	Yes	No	The number of times a Group Buffer Pool (GBP) dependent index page was read for the index into the local buffer pool from disk because it was not found in the GBP.
OBJECT_INDEX_ GBP_INVALID_PAGES	BIGINT	Yes	No	The number of times an index page was attempted to be read for the index from the group buffer pool because the page was invalid in the local buffer pool.
OBJECT_INDEX_ LBP_PAGES_FOUND	BIGINT	Yes	No	The number of times an index page for the index was present in the local buffer pool.
OBJECT_INDEX_ GBP_INDEP_PAGES_ FOUND_IN_LBP	BIGINT	Yes	No	The number of group buffer pool (GBP) independent index pages found in the local buffer pool (LBP) by the agent.
OBJECT_COL_L_READS	BIGINT	Yes	No	The number of column-organized pages that are logically read from the buffer pool for a table.
OBJECT_COL_P_READS	BIGINT	Yes	No	The number of column-organized pages that are physically read for a table.
OBJECT_COL_GBP_ L_READS	BIGINT	Yes	No	The number of times that a group buffer pool (GBP) dependent column-organized page is requested from the GBP for a table. The page is requested because a valid version of the page does not exist in the local buffer pool (LBP).

Table 348. OBJECT_METRICS table (continued)

Column Name	Data Type	Nullable?	Key?	Description
OBJECT_COL_GBP_P_READS	BIGINT	Yes	No	The number of times that a group buffer pool (GBP) dependent column-organized page is read into the local buffer pool (LBP) from disk for a table. The page is read from disk into the LBP because the page is not in the GBP.
OBJECT_COL_GBP_INVALID_PAGES	BIGINT	Yes	No	The number of times that a column-organized page is requested from the group buffer pool (GBP) for a table. The page is requested because the version of the page in the local buffer pool (LBP) is invalid. Outside of a Db2 pureScale environment, this value is null.
OBJECT_COL_LBP_PAGES_FOUND	BIGINT	Yes	No	The number of times that a column-organized page for a table is present in the local buffer pool (LBP).
OBJECT_COL_GBP_INDEP_PAGES_FOUND_IN_LBP	BIGINT	Yes	No	The number of group buffer pool (GBP) independent column-organized pages found in a local buffer pool (LBP) by an agent.
OBJECT_DATA_CACHING_TIER_L_READS	BIGINT	Yes	No	Reserved for future use.
OBJECT_DATA_CACHING_TIER_PAGES_FOUND	BIGINT	Yes	No	Reserved for future use.
OBJECT_DATA_CACHING_TIER_GBP_INVALID_PAGES	BIGINT	Yes	No	Reserved for future use.
OBJECT_DATA_CACHING_TIER_GBP_INDEP_PAGES_FOUND	BIGINT	Yes	No	Reserved for future use.
OBJECT_XDA_CACHING_TIER_L_READS	BIGINT	Yes	No	Reserved for future use.
OBJECT_XDA_CACHING_TIER_PAGES_FOUND	BIGINT	Yes	No	Reserved for future use.
OBJECT_XDA_CACHING_TIER_GBP_INVALID_PAGES	BIGINT	Yes	No	Reserved for future use.
OBJECT_XDA_CACHING_TIER_GBP_INDEP_PAGES_FOUND	BIGINT	Yes	No	Reserved for future use.
OBJECT_INDEX_CACHING_TIER_L_READS	BIGINT	Yes	No	Reserved for future use.

Table 348. OBJECT_METRICS table (continued)

Column Name	Data Type	Nullable ?	Key?	Description
OBJECT_INDEX_ CACHING_TIER_ PAGES_FOUND	BIGINT	Yes	No	Reserved for future use.
OBJECT_INDEX_ CACHING_TIER_ GBP_INVALID_PAGES	BIGINT	Yes	No	Reserved for future use.
OBJECT_INDEX_ CACHING_TIER_ GBP_INDEP_ PAGES_FOUND	BIGINT	Yes	No	Reserved for future use.
OBJECT_COL_ CACHING_TIER_ L_READS	BIGINT	Yes	No	Reserved for future use.
OBJECT_COL_ CACHING_TIER_ PAGES_FOUND	BIGINT	Yes	No	Reserved for future use.
OBJECT_COL_ CACHING_TIER_ GBP_INVALID_PAGES	BIGINT	Yes	No	Reserved for future use.
OBJECT_COL_ CACHING_TIER_ GBP_INDEP_ PAGES_FOUND	BIGINT	Yes	No	Reserved for future use.
EXT_TABLE_RECV_ WAIT_TIME	BIGINT	Yes	No	Total time the agent spent waiting for the external table readers to read and process data from external tables. The value is given in milliseconds.
EXT_TABLE_ RECVS_TOTAL	BIGINT	Yes	No	Total number of buffers that were received by the agent from the external table readers.
EXT_TABLE_ RECV_VOLUME	BIGINT	Yes	No	Total volume that were received by the agent from the external table readers. The value is given in bytes.
EXT_TABLE_ READ_VOLUME	BIGINT	Yes	No	Total volume that was read by the external table reader from physical devices, such as disks. The value is given in bytes.
EXT_TABLE_SEND_ WAIT_TIME	BIGINT	Yes	No	Total time the agent spent waiting for the sent data to be processed and written by the external table writers. The value is given in milliseconds.
EXT_TABLE_ SENDS_TOTAL	BIGINT	Yes	No	Total number of buffers that were sent to the external table writers.
EXT_TABLE_ SEND_VOLUME	BIGINT	Yes	No	Total volume that was sent by the agent to the external table writers. The value is given in bytes.
EXT_TABLE_ WRITE_VOLUME	BIGINT	Yes	No	Total volume that was written by the external writers to physical devices, such as disks. The value is given in bytes.

Explain register values

The tables in this topic describe the interaction of the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values, both with each other and with the PREP and BIND commands.

With dynamic SQL, the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values interact as follows.

Table 349. Interaction of Explain Special Register Values (Dynamic SQL)

EXPLAIN SNAPSHOT values	EXPLAIN MODE values					
	NO	**YES**	**EXPLAIN**	**REOPT**	**RECOMMEND INDEXES**	**EVALUATE INDEXES**
NO	• Results of query returned.	• Explain tables populated. • Results of query returned.	• Explain tables populated. • Results of query not returned (dynamic statements not executed).	• Explain tables populated when a statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain tables populated. • Results of query not returned (dynamic statements not executed). • Indexes recommended.	• Explain tables populated. • Results of query not returned (dynamic statements not executed). • Indexes evaluated.
YES	• Explain Snapshot taken. • Results of query returned.	• Explain tables populated. • Explain Snapshot taken. • Results of query returned.	• Explain tables populated. • Explain Snapshot taken. • Results of query not returned (dynamic statements not executed).	• Explain tables populated when a statement qualifies for reoptimization at execution time. • Explain Snapshot taken. • Results of query returned.	• Explain tables populated. • Explain Snapshot taken. • Results of query not returned (dynamic statements not executed). • Indexes recommended.	• Explain tables populated. • Explain Snapshot taken. • Results of query not returned (dynamic statements not executed). • Indexes evaluated.
EXPLAIN	• Explain Snapshot taken. • Results of query not returned (dynamic statements not executed).	• Explain tables populated. • Explain Snapshot taken. • Results of query not returned (dynamic statements not executed).	• Explain tables populated. • Explain Snapshot taken. • Results of query not returned (dynamic statements not executed).	• Explain tables populated when a statement qualifies for reoptimization at execution time. • Explain Snapshot taken when a statement qualifies for reoptimization at execution time. • Results of query not returned (dynamic or incremental-bind statements not executed).	• Explain tables populated. • Explain Snapshot taken. • Results of query not returned (dynamic statements not executed). • Indexes recommended.	• Explain tables populated. • Explain Snapshot taken. • Results of query not returned (dynamic statements not executed). • Indexes evaluated.

Table 349. Interaction of Explain Special Register Values (Dynamic SQL) (continued)

EXPLAIN SNAPSHOT values	EXPLAIN MODE values					
	NO	YES	EXPLAIN	REOPT	RECOMMEND INDEXES	EVALUATE INDEXES
REOPT	• Explain Snapshot taken when a statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain tables populated. • Explain Snapshot taken when a statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain tables populated. • Explain Snapshot taken when a statement qualifies for reoptimization at execution time. • Results of query not returned (dynamic or incremental-bind statements not executed).	• Explain tables populated when a statement qualifies for reoptimization at execution time. • Explain Snapshot taken when a statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain tables populated. • Explain Snapshot taken when a statement qualifies for reoptimization at execution time. • Results of query not returned (dynamic or incremental-bind statements not executed). • Indexes recommended.	• Explain tables populated. • Explain Snapshot taken when a statement qualifies for reoptimization at execution time. • Results of query not returned (dynamic or incremental-bind statements not executed). • Indexes evaluated.

The CURRENT EXPLAIN MODE special register interacts with the EXPLAIN bind option in the following way for dynamic SQL.

Table 350. Interaction of EXPLAIN Bind Option and CURRENT EXPLAIN MODE

EXPLAIN MODE values	EXPLAIN Bind option values			
	NO	YES	REOPT	ALL
NO	• Results of query returned.	• Explain tables populated for static SQL. • Results of query returned.	• Explain tables populated for static SQL when statement qualifies for reoptimization at execution time. • Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain tables populated for static SQL. • Explain tables populated for dynamic SQL. • Results of query returned.

Table 350. Interaction of EXPLAIN Bind Option and CURRENT EXPLAIN MODE (continued)

EXPLAIN MODE values	EXPLAIN Bind option values			
	NO	**YES**	**REOPT**	**ALL**
YES	• Explain tables populated for dynamic SQL. • Results of query returned.	• Explain tables populated for static SQL. • Explain tables populated for dynamic SQL. • Results of query returned.	• Explain tables populated for static SQL when statement qualifies for reoptimization at execution time. • Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain tables populated for static SQL. • Explain tables populated for dynamic SQL. • Results of query returned.
EXPLAIN	• Explain tables populated for dynamic SQL. • Results of query not returned (dynamic statements not executed).	• Explain tables populated for static SQL. • Explain tables populated for dynamic SQL. • Results of query not returned (dynamic statements not executed).	• Explain tables populated for static SQL when statement qualifies for reoptimization at execution time. • Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query not returned (dynamic statements not executed).	• Explain tables populated for static SQL. • Explain tables populated for dynamic SQL. • Results of query not returned (dynamic statements not executed).

Table 350. Interaction of EXPLAIN Bind Option and CURRENT EXPLAIN MODE (continued)

EXPLAIN MODE values	EXPLAIN Bind option values			
	NO	**YES**	**REOPT**	**ALL**
REOPT	• Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain tables populated for static SQL when statement qualifies for reoptimization at execution time. • Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain tables populated for static SQL when statement qualifies for reoptimization at execution time. • Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain tables populated for static SQL when statement qualifies for reoptimization at execution time. • Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.
RECOMMEND INDEXES	• Explain tables populated for dynamic SQL. • Results of query not returned (dynamic statements not executed). • Recommend indexes.	• Explain tables populated for static SQL. • Explain tables populated for dynamic SQL. • Results of query not returned (dynamic statements not executed). • Recommend indexes.	• Explain tables populated for static SQL when statement qualifies for reoptimization at execution time. • Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query not returned (dynamic statements not executed). • Recommend indexes.	• Explain tables populated for static SQL. • Explain tables populated for dynamic SQL. • Results of query not returned (dynamic statements not executed). • Recommend indexes.

Table 350. Interaction of EXPLAIN Bind Option and CURRENT EXPLAIN MODE (continued)

EXPLAIN MODE values	EXPLAIN Bind option values			
	NO	YES	REOPT	ALL
EVALUATE INDEXES	• Explain tables populated for dynamic SQL. • Results of query not returned (dynamic statements not executed). • Evaluate indexes.	• Explain tables populated for static SQL. • Explain tables populated for dynamic SQL. • Results of query not returned (dynamic statements not executed). • Evaluate indexes.	• Explain tables populated for static SQL when statement qualifies for reoptimization at execution time. • Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query not returned (dynamic statements not executed). • Evaluate indexes.	• Explain tables populated for static SQL. • Explain tables populated for dynamic SQL. • Results of query not returned (dynamic statements not executed). • Evaluate indexes.

The CURRENT EXPLAIN SNAPSHOT special register interacts with the EXPLSNAP bind option in the following way for dynamic SQL.

Table 351. Interaction of EXPLSNAP bind Option and CURRENT EXPLAIN SNAPSHOT

EXPLAIN SNAPSHOT values	EXPLSNAP Bind option values			
	NO	YES	REOPT	ALL
NO	• Results of query returned.	• Explain Snapshot taken for static SQL. • Results of query returned.	• Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time. • Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain Snapshot taken for static SQL. • Explain Snapshot taken for dynamic SQL. • Results of query returned.

Table 351. Interaction of EXPLSNAP bind Option and CURRENT EXPLAIN SNAPSHOT (continued)

EXPLAIN SNAPSHOT values	EXPLSNAP Bind option values			
	NO	**YES**	**REOPT**	**ALL**
YES	• Explain Snapshot taken for dynamic SQL. • Results of query returned.	• Explain Snapshot taken for static SQL. • Explain Snapshot taken for dynamic SQL. • Results of query returned.	• Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time. • Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain Snapshot taken for static SQL. • Explain Snapshot taken for dynamic SQL. • Results of query returned.
EXPLAIN	• Explain Snapshot taken for dynamic SQL. • Results of query not returned (dynamic statements not executed).	• Explain Snapshot taken for static SQL. • Explain Snapshot taken for dynamic SQL. • Results of query not returned (dynamic statements not executed).	• Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time. • Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query not returned (dynamic statements not executed).	• Explain Snapshot taken for static SQL. • Explain Snapshot taken for dynamic SQL. • Results of query not returned (dynamic statements not executed).

Table 351. Interaction of EXPLSNAP bind Option and CURRENT EXPLAIN SNAPSHOT (continued)

EXPLAIN SNAPSHOT values	EXPLSNAP Bind option values			
	NO	**YES**	**REOPT**	**ALL**
REOPT	• Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time. • Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time. • Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time. • Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y