IBM DB2 Universal Database

# SQL Reference

*Version 6*

**IBM**

IBM DB2 Universal Database

# SQL Reference

*Version 6*

Before using this information and the product it supports, be sure to read the general information under "Appendix R. Notices" on page 1245.

# Contents

# Chapter 1. Introduction

This introductory chapter:
- Identifies this book's purpose and audience,
- Explains how to use the book and its structure,
- Explains the syntax diagram notation, the naming and highlighting conventions used throughout the manual,
- Lists related documentation,
- Presents the product family overview.

## Who Should Use This Book

This book is intended for anyone who wants to use the Structured Query Language (SQL) to access a database. It is primarily for programmers and database administrators, but it can also be used by general users using the command line processor.

This book is a reference rather than a tutorial. It assumes that you will be writing application programs and therefore presents the full functions of the database manager.

## How To Use This Book

This book defines the SQL language used by DB2 Universal Database Version 6. Use it as a reference manual for information on relational database concepts, language elements, functions, the forms of queries, and the syntax and semantics of the SQL statements. The appendixes can be used to find limitations and information on important components.

### How This Book is Structured

This book has the following sections:
- "Chapter 1. Introduction", identifies the purpose, the audience, and the use of the book.
- "Chapter 2. Concepts" on page 7, discusses the basic concepts of relational databases and SQL.
- "Chapter 3. Language Elements" on page 61, describes the basic syntax of SQL and the language elements that are common to many SQL statements.
- "Chapter 4. Functions" on page 187, contains syntax diagrams, semantic descriptions, rules, and usage examples of SQL column and scalar functions.

**1**

- "Chapter 5. Queries" on page 367, describes the various forms of a query.
- "Chapter 6. Statements" on page 425, contains syntax diagrams, semantic descriptions, rules, and examples of all SQL statements.
- The appendixes contain the following information:
    - "Appendix A. SQL Limits" on page 919 contains the SQL limitations
    - "Appendix B. SQL Communications (SQLCA)" on page 927 contains the SQLCA structure
    - "Appendix C. SQL Descriptor Area (SQLDA)" on page 933 contains the SQLDA structure
    - "Appendix D. Catalog Views" on page 945 contains the catalog views for the database
    - "Appendix F. Sample Tables" on page 1059 contains the sample tables used for examples
    - "Appendix G. Reserved Schema Names and Reserved Words" on page 1079 contains the reserved schema names and the reserved words for the IBM SQL and ISO/ANS SQL92 standards
    - "Appendix H. Comparison of Isolation Levels" on page 1083 contains a summary of the isolation levels.
    - "Appendix I. Interaction of Triggers and Constraints" on page 1085 discusses the interaction of triggers and referential constraints.
    - "Appendix J. Incompatibilities Between Releases" on page 1089 contains the release to release incompatibilities.
    - "Appendix K. Explain Tables and Definitions" on page 1131 contains the Explain tables and how they are defined.
    - "Appendix L. Explain Register Values" on page 1165 describes the interaction of the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values with each other and the PREP and BIND commands.
    - "Appendix M. Recursion Example: Bill of Materials" on page 1169 contains an example of a recursive query.
    - "Appendix N. Exception Tables" on page 1175 contains information on user-created tables that are used with the SET INTEGRITY statement.
    - "Appendix O. Japanese and Traditional-Chinese EUC Considerations" on page 1181 lists considerations when using EUC character sets.

## How to Read the Syntax Diagrams

Throughout this book, syntax is described using the structure defined as follows:

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

The ►►── symbol indicates the beginning of a statement.

The ──► symbol indicates that the statement syntax is continued on the next line.

The ►── symbol indicates that a statement is continued from the previous line.

The ──►◄ symbol indicates the end of a statement.

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

```
►►──STATEMENT──required item──────────────────────────────────────────►◄
```

Optional items appear below the main path.

```
►►──STATEMENT───────────────────────────────────────────────────────────►◄
               └─optional item─┘
```

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

```
                ┌─optional item─┐
►►──STATEMENT───┴───────────────┴───────────────────────────────────────►◄
```

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.

```
►►──STATEMENT───┬─required choice1─┬─────────────────────────────────────►◄
                └─required choice2─┘
```

If choosing none of the items is an option, the entire stack appears below the
main path.

```
►►──STATEMENT──┬─────────────────┬──────────────────────►◄
               ├─optional choice1─┤
               └─optional choice2─┘
```

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

```
                ┌─default choice──┐
►►──STATEMENT───┼─────────────────┼─────────────────────►◄
               ├─optional choice─┤
               └─optional choice─┘
```
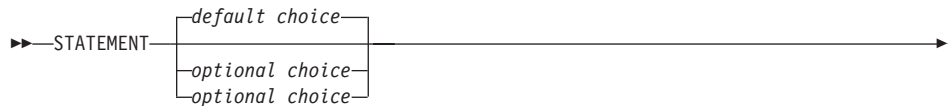
An arrow returning to the left, above the main line, indicates an item that can
be repeated. In this case, repeated items must be separated by one or more
blanks.

```
                  ┌─────────────────┐
►►──STATEMENT──────┴─repeatable item─┴──────────────────►◄
```

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

```
                  ┌──,──────────────┐
►►──STATEMENT──────┴─repeatable item─┴──────────────────►◄
```

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 set of several parameters. For example, in the following diagram, the variable parameter-block can be replaced by any of the interpretations of the diagram that is headed **parameter-block**:

```
►►──STATEMENT──┤ parameter-block ├────────────────────────►◄
```

**parameter-block:**

```
├──┬─parameter1────────────────────┬──────────────────────┤
   └─parameter2──┬─parameter3─┬─────┘
                 └─parameter4─┘
```

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

```
►►──STATEMENT──item1──●──item2──●──item3──●──item4──────────►◄
```

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

```
STATEMENT item1 item2 item3 item4
STATEMENT item1 item3 item2 item4
```

## Conventions Used in This Manual

This section specifies some conventions which are used consistently throughout this manual.

### Error Conditions

An error condition is indicated within the text of the manual by listing the SQLSTATE associated with the error in brackets. For example: A duplicate signature raises an SQL error (SQLSTATE 42723).

### Highlighting Conventions

The following conventions are used in this book.

| | |
|---|---|
| **Bold** | Indicates commands, keywords, and other items whose names are predefined by the system. |

| | |
|---|---|
| *Italics* | Indicates one of the following:<br>• Names or values (variables) that must be supplied by the user<br>• General emphasis<br>• The introduction of a new term<br>• A reference to another source of information. |
| `Monospace` | Indicates one of the following:<br>• Files and directories<br>• Information that you are instructed to type at a command prompt or in a window<br>• Examples of specific data values<br>• Examples of text similar to what may be displayed by the system<br>• Examples of system messages. |

## Related Documentation for This Book

The following publications may prove useful in preparing applications:

*Administration Guide*

Contains information required to design, implement, and maintain a database to be accessed either locally or in a client/server environment.

*Application Development Guide*

Discusses the application development process and how to code, compile, and execute application programs that use embedded SQL and APIs to access the database.

*IBM SQL Reference* SC26-8416

This manual contains all the common elements of SQL that span across IBM's library of database products. It provides limits and rules that assist in preparing portable programs using IBM databases. It provides a list of SQL extensions and incompatibilities among the following standards and products: SQL92E, XPGG4-SQL, IBM-SQL and the IBM relational database products.

*American National Standard X3.135-1992, Database Language SQL*

Contains the ANSI standard definition of SQL.

*ISO/IEC 9075:1992, Database Language SQL*

Contains the ISO standard definition of SQL.

# Chapter 2. Concepts

The chapter provides an overview of the concepts commonly used in the Structured Query Language (SQL). The intent of the chapter is to provide a high-level view of the concepts. The reference material that follows provides a more detailed view.

## Relational Database

A *relational database* is a database that can be perceived as a set of tables and manipulated in accordance with the relational model of data. It contains a set of objects used to store, manage, and access data. Examples of such objects are tables, views, indexes, functions, triggers, and packages.

A *partitioned* relational database is a relational database where the data is managed across multiple partitions (also called nodes). This partitioning of data across partitions is transparent to users of most SQL statements. However, some DDL statements take partition information into consideration (e.g. create nodegroup).

## 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 perceived 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.

## Embedded SQL

Embedded SQL statements are SQL statements written within application programming languages such as C and preprocessed by an SQL preprocessor before the application program is compiled. There are two types of embedded SQL: static and dynamic.

### Static SQL

The source form of a static SQL statement is embedded within an application program written in a host language such as COBOL. The statement is prepared before the program is executed and the operational form of the statement persists beyond the execution of the program.

A source program containing static SQL statements must be processed by an SQL precompiler before it is compiled. The precompiler turns the SQL statements into host language comments, and generates host language statements to invoke the database manager. The syntax of the SQL statements is checked during the precompile process.

The preparation of an SQL application program includes precompilation, the binding of its static SQL statements to the target database, and compilation of the modified source program. The steps are specified in the *Application Development Guide*.

### Dynamic SQL

Programs containing embedded dynamic SQL statements must be precompiled like those containing static SQL, but unlike static SQL, the dynamic SQL statements are constructed and prepared at run time. The SQL statement text is prepared and executed using either the PREPARE and EXECUTE statements, or the EXECUTE IMMEDIATE statement. The statement can also be executed with the cursor operations if it is a SELECT statement.

## DB2 Call Level Interface (CLI)

The DB2 Call Level Interface is an application programming interface in which functions are provided to application programs to process dynamic SQL statements. CLI programs can also be compiled using an Open Database Connectivity (ODBC) software developer's kit, enabling them to access ODBC data sources. Through the interface, applications use procedure calls at execution time to connect to databases, to issue SQL statements, and to get returned data and status information. Unlike using embedded SQL, no

precompilation is required. Applications developed using this interface may be executed on a variety of databases without being compiled against each of the databases.

The DB2 CLI interface provides many features not available in embedded SQL. A few of these are:

- CLI provides function calls which support a consistent way to query and retrieve database system catalog information across the DB2 family of database management systems. This reduces the need to write database server specific catalog queries.
- CLI provides support for scrollable cursors.
- Stored procedures called from application programs written using CLI can return result sets to those programs.

The *CLI Guide and Reference* describes the APIs supported with this interface.

## Java Database Connectivity (JDBC) and Embedded SQL for Java (SQLJ) Programs

DB2 Universal Database implements two standards-based Java programming APIs: Java Database Connectivity (JDBC) and embedded SQL for Java (SQLJ). Both can be used to create Java applications and applets that access DB2.

JDBC calls are translated to calls to DB2 CLI through Java native methods. JDBC requests flow from the DB2 client through DB2 CLI to the DB2 server. Static SQL cannot be used by JDBC.

SQLJ applications use JDBC as a foundation for such tasks as connecting to databases and handling SQL errors, but can also contain embedded static SQL statements in the SQLJ source files. An SQLJ source file has to be translated with the SQLJ translator before the resulting Java source code can be compiled.

For more information on JDBC and SQLJ applications, refer to the *Application Development Guide*.

## Interactive SQL

*Interactive* SQL statements are entered by a user through an interface like the command line processor or the command center. These statements are processed as dynamic SQL statements. For example, an interactive SELECT statement can be processed dynamically using the DECLARE CURSOR, PREPARE, DESCRIBE, OPEN, FETCH, and CLOSE statements.

The *Command Reference* lists the commands that can be issued using the command line processor or similar facilities and products.

## Schemas

A *schema* is a collection of named objects. Schemas provide a logical classification of objects in the database. Some of the objects that a schema may contain include tables, views, nicknames, triggers, functions and packages.

A schema is also an object in the database. It is explicitly created using the CREATE SCHEMA statement with a user recorded as owner. It can also be implicitly created when another object is created, provided the user has IMPLICIT_SCHEMA authority.

A *schema name* is used as the high-order part of a two-part object name. An object that is contained in a schema is assigned to the schema when the object is created. The schema to which it is assigned is determined by the name of the object if specifically qualified with a schema name or by the default schema name if not qualified.

For example, a user with DBADM authority creates a schema called C for user A.

```
CREATE SCHEMA C AUTHORIZATION A
```

User A can then issue the following statement to create a table called X in schema C:

```
CREATE TABLE C.X (COL1 INT)
```

### Controlling Use of Schemas

When a database is created, all users have IMPLICIT_SCHEMA authority. This allows any user to create objects in any schema that does not already exist. An implicitly created schema allows any user to create other objects in this schema. [1]

If IMPLICIT_SCHEMA authority is revoked from PUBLIC, schemas are either explicitly created using the CREATE SCHEMA statement or implicitly created by users (such as those with DBADM authority) who are granted IMPLICIT_SCHEMA authority. While revoking IMPLICIT_SCHEMA authority from PUBLIC increases control over the use of schema names, it may result in authorization errors in existing applications when they attempt to create objects.

---

1. The default privileges on an implicitly created schema provide upward compatibility with previous versions. Alias, distinct type, function and trigger creation is extended to implicitly created schemas.

There are also privileges associated with a schema that control which users have the privilege to create, alter and drop objects in the schema. A schema owner is initially given all of these privileges on a schema with the ability to grant them to others. An implicitly created schema is owned by the system and all users are initially given the privilege to create objects in such a schema. A user with DBADM or SYSADM authority can change the privileges held by users on any schema. Therefore, access to create, alter and drop objects in any schema (even one that is implicitly created) can be controlled.

## Tables

Tables are logical structures maintained by the database manager. Tables are made up of columns and rows. The rows are not necessarily ordered within a table (order is determined by the application program). At the intersection of every column and row is a specific data item called a *value*. A *column* is a set of values of the same data type. A *row* is a sequence of values such that the *n*th value is a value of the *n*th column of the table.

A *base table* is created with the CREATE TABLE statement and is used to hold persistent user data. A *result table* is a set of rows that the database manager selects or generates from one or more base tables to satisfy a query.

A summary table is a table that is defined by a query that is also used to determine the data in the table. Summary tables can be used to improve the performance of queries. If the database manager determines that a portion of a query could be resolved using a summary table, the query may be rewritten by the database manager to use the summary table. This decision is based on certain settings such as CURRENT REFRESH AGE and CURRENT QUERY OPTIMIZATION special registers.

A table can have the data type of each column defined separately, or have the types for the columns based on the attributes of a user-defined structured type. This is called a *typed table*. A user-defined structured type may be part of a type hierarchy. A *subtype* is said to inherit attributes from its *supertype*. Similarly, a typed table can be part of a table hierarchy. A *subtable* is said to inherit columns from its *supertable*. Note that the term *subtype* applies to a user-defined structured type and all user-defined structured types that are below it in the type hierarchy. A *proper subtype* of a structured type T is a structured type below T in the type hierarchy. Similarly the term *subtable* applies to a typed table and all typed tables that are below it in the table hierarchy. A *proper subtable* of a table T is a table below T in the table hierarchy.

## Views

A *view* provides an alternative way of looking at the data in one or more tables.

A view is a named specification of a result table. The specification is a SELECT statement that is executed whenever the view is referenced in an SQL statement. Thus, a view can be thought of as having columns and rows just like a base table. For retrieval, all views can be used just like base tables. Whether a view can be used in an insert, update, or delete operation depends on its definition as explained in the description of CREATE VIEW. (See "CREATE VIEW" on page 714 for more information.)

When the column of a view is directly derived from a column of a base table, that column inherits any constraints that apply to the column of the base table. For example, if a view includes a foreign key of its base table, INSERT and UPDATE operations using that view are subject to the same referential constraint as the base table. Also, if the base table of a view is a parent table, DELETE and UPDATE operations using that view are subject to the same rules as DELETE and UPDATE operations on the base table.

A view can have the data type of each column derived from the result table, or have the types for the columns based on the attributes of a user-defined structured type. This is called a *typed view*. Similar to a typed table, a typed view can be part of a view hierarchy. A *subview* is said to inherit columns from its *superview*. The term *subview* applies to a typed view and all typed views that are below it in the view hierarchy. A *proper subview* of a view V is a view below V in the typed view hierarchy.

A view may become inoperative, in which case it is no longer available for SQL statements.

## Aliases

An *alias* is an alternate name for a table or view. It can be used to reference a table or view in those cases where an existing table or view can be referenced.[2] Like tables and views, an alias may be created, dropped, and have comments associated with it. Unlike tables, aliases may refer to each other in a process called chaining. Aliases are publicly referenced names so no special authority or privilege is required to use an alias. Access to the tables and views referred to by the alias, however, still require the appropriate authorization for the current context.

---

2. An alias cannot be used in all contexts. For example, it cannot be used in the check condition of a check constraint.

In addition to table aliases, there are other types of aliases such as database and network aliases. Aliases can also be created for nicknames.

Refer to "Aliases" on page 68 and "CREATE ALIAS" on page 528 for more information about aliases.

## Indexes

An *index* is an ordered set of pointers to rows of a base table. Each index is based on the values of data in one or more table columns. An index is an object that is separate from the data in the table. When an index is created, the database manager builds this structure and maintains it automatically.

Indexes are used by the database manager to:

- Improve performance. In most cases, access to data is faster than without an index.

  An index cannot be created for a view. However, an index created for a table on which a view is based may improve the performance of operations on the view.
- Ensure uniqueness. A table with a unique index cannot have rows with identical keys.

## Keys

A *key* is a set of columns that can be used to identify or access a particular row or rows. The key is identified in the description of a table, index, or referential constraint. The same column can be part of more than one key.

A key composed of more than one column is called a *composite key*. In a table with a composite key, the ordering of the columns within the composite key is not constrained by their ordering within the table. The term *value* when used with respect to a composite key denotes a composite value. Thus, a rule such as "the value of the foreign key must be equal to the value of the primary key" means that each component of the value of the foreign key must be equal to the corresponding component of the value of the primary key.

### Unique Keys

A *unique key* is a key that is constrained so that no two of its values are equal. The columns of a unique key cannot contain null values. The constraint is enforced by the database manager during the execution of any operation that changes data values, such as INSERT or UPDATE. The mechanism used to enforce the constraint is called a *unique index*. Thus, every unique key is a key

of a unique index. Such an index is also said to have the UNIQUE attribute. See "Unique Constraints" on page 15 for a more detailed description.

## Primary Keys

A *primary key* is a special case of a unique key. A table cannot have more than one primary key. See "Unique Keys" on page 13 for a more detailed description.

## Foreign Keys

A *foreign key* is a key that is specified in the definition of a referential constraint. See "Referential Constraints" on page 15 for a more detailed description.

## Partitioning Keys

A *partitioning key* is a key that is part of the definition of a table in a partitioned database. The partitioning key is used to determine the partition on which the row of data is stored. If a partitioning key is defined, unique keys and primary keys must include the same columns as the partitioning key (they may have more columns).

## Constraints

A *constraint* is a rule that the database manager enforces.

There are three types of constraints:

- A *unique constraint* is a rule that forbids duplicate values in one or more columns within a table. Unique and primary keys are the supported unique constraints. For example, a unique constraint could be defined on the supplier identifier in the supplier table to ensure that the same supplier identifier is not given to two suppliers.

- A *referential constraint* is a logical rule about values in one or more columns in one or more tables. For example, a set of tables shares information about a corporation's suppliers. Occasionally, a supplier's name changes. A referential constraint could be defined stating that the ID of the supplier in a table must match a supplier id in the supplier information. This constraint prevents inserts, updates or deletes that would otherwise result in missing supplier information.

- A *table check constraint* sets restrictions on data added to a specific table. For example, it could define the salary level for an employee to never be less than $20,000.00 when salary data is added or updated in a table containing personnel information.

Referential and table check constraints may be turned on or off. Loading large amounts of data into the database is typically a time to turn off checking the enforcement of a constraint. The details of setting constraints on or off are discussed in "SET INTEGRITY" on page 881.

## Unique Constraints

A *unique constraint* is the rule that the values of a key are valid only if they are unique within the table. Unique constraints are optional and can be defined in the CREATE TABLE or ALTER TABLE statement using the PRIMARY KEY clause or the UNIQUE clause. The columns specified in a unique constraint must be defined as NOT NULL. A unique index is used by the database manager to enforce the uniqueness of the key during changes to the columns of the unique constraint.

A table can have an arbitrary number of unique constraints, with at most one unique constraint defined as a primary key. A table cannot have more than than one unique constraint on the same set of columns.

A unique constraint that is referenced by the foreign key of a referential constraint is called the *parent key*.

When a unique constraint is defined in a CREATE TABLE statement, a unique index is automatically created by the database manager and designated as a primary or unique system-required index.

When a unique constraint is defined in an ALTER TABLE statement and an index exists on the same columns, that index is designated as unique and system-required. If such an index does not exist, the unique index is automatically created by the database manager and designated as a primary or unique system-required index.

Note that there is a distinction between defining a unique constraint and creating a unique index. Although both enforce uniqueness, a unique index allows nullable columns and generally cannot be used as a parent key.

## Referential Constraints

*Referential integrity* is the state of a database in which all values of all foreign keys are valid. A *foreign key* is a column or set of columns in a table whose

values are required to match at least one primary key or unique key value of a row of its parent table. A *referential constraint* is the rule that the values of the foreign key are valid only if:

- they appear as values of a parent key, or
- some component of the foreign key is null.

The table containing the parent key is called the *parent table* of the referential constraint, and the table containing the foreign key is said to be a *dependent* of that table.

Referential constraints are optional and can be defined in CREATE TABLE statements and ALTER TABLE statements. Referential constraints are enforced by the database manager during the execution of INSERT, UPDATE, DELETE, ALTER TABLE ADD CONSTRAINT, and SET INTEGRITY statements. The enforcement is effectively performed at the completion of the statement.

Referential constraints with a delete or update rule of RESTRICT are enforced before all other referential constraints. Referential constraints with a delete or update rule of NO ACTION behave like RESTRICT in most cases. However, in certain SQL statements there can be a difference.

Note that referential integrity, check constraints and triggers can be combined in execution. For further information on the combination of these elements, see "Appendix I. Interaction of Triggers and Constraints" on page 1085.

The rules of referential integrity involve the following concepts and terminology:

**Parent key**    A primary key or unique key of a referential constraint.

**Parent row**    A row that has at least one dependent row.

**Parent table**    A table that contains the parent key of a referential constraint. A table can be a parent in an arbitrary number of referential constraints. A table which is the parent in a referential constraint may also the dependent of a referential constraint.

**Dependent table**    A table that contains at least one referential constraint in its definition. A table can be a dependent in an arbitrary number of referential constraints. A table which is the dependent in a referential constraint may also the parent of a referential constraint.

| | |
|---|---|
| **Descendent table** | A table is a descendent of table T if it is a dependent of T or a descendent of a dependent of T. |
| **Dependent row** | A row that has at least one parent row. |
| **Descendent row** | A row is a descendent of row p if it is a dependent of p or a descendent of a dependent of p. |
| **Referential cycle** | A set of referential constraints such that each table in the set is a descendent of itself. |
| **Self-referencing row** | A row that is a parent of itself. |
| **Self-referencing table** | A table that is a parent and a dependent in the same referential constraint. The constraint is called a *self-referencing constraint*. |

### Insert Rule

The insert rule of a referential constraint is that a non-null insert value of the foreign key must match some value of the parent key of the parent table. The value of a composite foreign key is null if any component of the value is null. This rule is implicit when a foreign key is specified.

### Update Rule

The update rule of a referential constraint is specified when the referential constraint is defined. The choices are NO ACTION and RESTRICT. The update rule applies when a row of the parent or a row of the dependent table is updated.

In the case of a parent row, when a value in a column of the parent key is updated:
- if any row in the dependent table matched the original value of the key, the update is rejected when the update rule is RESTRICT
- if any row in the dependent table does not have a corresponding parent key when the update statement is completed (excluding AFTER triggers), the update is rejected when the update rule is NO ACTION.

In the case of a dependent row, the update rule that is implicit when a foreign key is specified is NO ACTION. NO ACTION means that a non-null update value of a foreign key must match some value of the parent key of the parent table when the update statement is completed.

The value of a composite foreign key is null if any component of the value is null.

**Delete Rule**

The delete rule of a referential constraint is specified when the referential constraint is defined. The choices are NO ACTION, RESTRICT, CASCADE, or SET NULL. SET NULL can be specified only if some column of the foreign key allows null values.

The delete rule of a referential constraint applies when a row of the parent table is deleted. More precisely, the rule applies when a row of the parent table is the object of a delete or propagated delete operation (defined below) and that row has dependents in the dependent table of the referential constraint. Let P denote the parent table, let D denote the dependent table, and let p denote a parent row that is the object of a delete or propagated delete operation. If the delete rule is:

- RESTRICT or NO ACTION, an error occurs and no rows are deleted
- CASCADE, the delete operation is propagated to the dependents of p in D
- SET NULL, each nullable column of the foreign key of each dependent of p in D is set to null

Each referential constraint in which a table is a parent has its own delete rule, and all applicable delete rules are used to determine the result of a delete operation. Thus, a row cannot be deleted if it has dependents in a referential constraint with a delete rule of RESTRICT or NO ACTION or the deletion cascades to any of its descendents that are dependents in a referential constraint with the delete rule of RESTRICT or NO ACTION.

The deletion of a row from parent table P involves other tables and may affect rows of these tables:

- If table D is a dependent of P and the delete rule is RESTRICT or NO ACTION, D is involved in the operation but is not affected by the operation.
- If D is a dependent of P and the delete rule is SET NULL, D is involved in the operation, and rows of D may be updated during the operation.
- If D is a dependent of P and the delete rule is CASCADE, D is involved in the operation and rows of D may be deleted during the operation.

    If rows of D are deleted, the delete operation on P is said to be propagated to D. If D is also a parent table, the actions described in this list apply, in turn, to the dependents of D.

Any table that may be involved in a delete operation on P is said to be *delete-connected* to P. Thus, a table is delete-connected to table P if it is a dependent of P or a dependent of a table to which delete operations from P cascade.

## Table Check Constraints

A *table check constraint* is a rule that specifies the values allowed in one or more columns of every row of a table. They are optional and can be defined using the SQL statements CREATE TABLE and ALTER TABLE. The specification of table check constraints is a restricted form of a search condition. One of the restrictions is that a column name in a table check constraint on table *T* must identify a column of *T*.

A table can have an arbitrary number of table check constraints. They are enforced when:
- a row is inserted into the table
- a row of the table is updated.

A table check constraint is enforced by applying its search condition to each row that is inserted or updated. An error occurs if the result of the search condition is false for any row.

When one or more table check constraints are defined in the ALTER TABLE statement for a table with existing data, the existing data is checked against the new condition before the ALTER TABLE statement succeeds. The table can be placed in *check pending state* which will allow the ALTER TABLE statement to succeed without checking the data. The SET INTEGRITY statement is used to place the table into check pending state. It is also used to resume the checking of each row against the constraint.

## Triggers

A *trigger* defines a set of actions that are executed at, or triggered by, a delete, insert, or update operation on a specified table. When such an SQL operation is executed, the trigger is said to be *activated*.

Triggers can be used along with referential constraints and check constraints to enforce data integrity rules. Triggers can also be used to cause updates to other tables, automatically generate or transform values for inserted or updated rows, or invoke functions to perform tasks such as issuing alerts.

Triggers are a useful mechanism to define and enforce *transitional* business rules which are rules that involve different states of the data (for example, salary cannot be increased by more than 10 percent). For rules that do not involve more than one state of the data, check and referential integrity constraints should be considered.

Using triggers places the logic to enforce the business rules in the database and relieves the applications using the tables from having to enforce it.

Centralized logic enforced on all the tables means easier maintenance, since no application program changes are required when the logic changes.

Triggers are optional and are defined using the CREATE TRIGGER statement.

There are a number of criteria that are defined when creating a trigger which are used to determine when a trigger should be activated.

- The *subject table* defines the table for which the trigger is defined.
- The *trigger event* defines a specific SQL operation that modifies the subject table. The operation could be delete, insert or update.
- The *trigger activation time* defines whether the trigger should be activated before or after the trigger event is performed on the subject table.

The statement that causes a trigger to be activated will include a *set of affected rows*. These are the rows of the subject table that are being deleted, inserted or updated. The *trigger granularity* defines whether the actions of the trigger will be performed once for the statement or once for each of the rows in the set of affected rows.

The *triggered action* consists of an optional search condition and a set of SQL statements that are executed whenever the trigger is activated. The SQL statements are only executed if the search condition evaluates to true. When the trigger activation time is before the trigger event, triggered action can include statements that select, set transition variables, and signal sqlstates. When the trigger activation time is after the trigger event, triggered action can include statements that select, update, insert, delete, and signal sqlstates.

The triggered action may refer to the values in the set of affected rows. This is supported through the use of *transition variables*. Transition variables use the names of the columns in the subject table qualified by a specified name that identifies whether the reference is to the old value (prior to the update) or the new value (after the update). The new value can also be changed using the SET transition-variable statement in before update or insert triggers. Another means of referring to the values in the set of affected rows is using *transition tables*. Transition tables also use the names of the columns of the subject table but have a name specified that allows the complete set of affected rows to be treated as a table. As with transition variables, a transition table can be defined for the old values and the new values but only in after triggers.

Multiple triggers can be specified for a combination of table, event, or activation time. The order in which the triggers are activated is the same as the order in which they were created. Thus, the most recently created trigger will be the last trigger activated.

The activation of a trigger may cause *trigger cascading*. This is the result of the activation of one trigger that executes SQL statements that cause the activation of other triggers or even the same trigger again. The triggered actions may also cause updates as a result of the original modification, or as a result of referential integrity delete rules which may result in the activation of additional triggers. With trigger cascading, a significant chain of triggers and referential integrity delete rules may be activated causing significant change to the database as a result of a single delete, insert or update statement.

## Event Monitors

An *event monitor* tracks specific data as the result of an event. For example, starting the database might be an event that causes an event monitor to track the number of users on the system by taking an hourly snapshot of authorization IDs using the database.

Event monitors are activated or deactivated by a statement (SET EVENT MONITOR STATE). A function (EVENT_MON_STATE) can be used to find the current state of an event monitor; that is, if it is active or not active.

## Queries

A *query* is a component of certain SQL statements that specifies a (temporary) result table.

## Table Expressions

A *table expression* creates a (temporary) result table from a simple query. Clauses further refine the result table. For example, a table expression could be a query that selects all the managers from several departments and further specifies that they have over 15 years of working experience and are located at the New York branch office.

### Common Table Expressions

A *common table expression* is like a temporary view within a complex query, and can be referenced in other places within the query; for example, in place of a view, to avoid creating the 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 bill of materials (BOM), airline reservation

systems, and network planning. A set of examples from a BOM application is contained in "Appendix M. Recursion Example: Bill of Materials" on page 1169.

## Packages

A *package* is an object that contains all the *sections* from a single source file. A section is the compiled form of an SQL statement. While every section corresponds to one statement, every statement does not necessarily have a section. The sections created for static SQL can be thought of as the bound or operational form of SQL statements. The sections created for dynamic SQL can be thought of as placeholder control structures which are used at execution time. Packages are produced during program preparation. See the *Application Development Guide* for more information on packages.

## Catalog Views

The database manager maintains a set of views and base tables that contain information about the data under its control. These views and base tables are collectively known as the *catalog*. They contain information about objects in the database such as tables, views, indexes, packages and functions.

The catalog views are like any other database views. SQL statements can be used to look at the data in the catalog views in the same way that data is retrieved from any other view in the system. The database manager ensures that the catalog contains accurate descriptions of the objects in the database at all times. A set of updatable catalog views can be used to modify certain values in the catalog (see "Updatable Catalog Views" on page 946).

Statistical information is also contained in the catalog. The statistical information is updated by utilities executed by an administrator, or through update statements by appropriately authorized users.

The catalog views are listed in "Appendix D. Catalog Views" on page 945.

## Application Processes, Concurrency, and Recovery

All SQL programs execute as part of an *application process* or agent. An application process involves the execution of one or more programs, and is the unit to which the database manager allocates resources and locks. Different application processes may involve the execution of different programs, or different executions of the same program.

More than one application process may request access to the same data at the same time. *Locking* is the mechanism used to maintain data integrity under such conditions, preventing, for example, two application processes from updating the same row of data simultaneously.

The database manager acquires locks in order to prevent uncommitted changes made by one application process from being accidentally perceived by any other. The database manager releases all locks it has acquired and retained on behalf of an application process when that process ends. However, an application process can explicitly request that locks be released sooner. This is done using a *commit* operation which releases locks acquired during the unit of work and also commits database changes made during the unit of work.

The database manager provides a means of *backing out* uncommitted changes made by an application process. This might be necessary in the event of a failure on the part of an application process, or in a *deadlock* or lock timeout situation. An application process itself, however, can explicitly request that its database changes be backed out. This operation is called *rollback*.

A *unit of work* is a recoverable sequence of operations within an application process. At any time, an application process can have multiple independent units of work. However, a given connection can only have a single unit of work [3].

A unit of work is initiated when an application process is initiated. A unit of work is also initiated when the previous unit of work is ended by something other than the termination of the application process. A unit of work is ended by a commit operation, a rollback operation, or the end of an application process. A commit or rollback operation affects only the database changes made within the unit of work it ends. While these changes remain uncommitted, other application processes are unable to perceive them and they can be backed out. [4] Once committed, these database changes are accessible by other application processes and can no longer be backed out by a rollback.

Locks acquired by the database manager on behalf of an application process are held until the end of a unit of work. The exception to this rule is with a read stability or cursor stability isolation level, or an uncommitted read level, in which case the lock is released as the cursor moves from row to row (see "Isolation Level" on page 25).

---

3. DB2 CLI and embedded SQL support a connection mode called *concurrent transactions* which supports multiple connections, each of which is an independent transaction. An application can have multiple concurrent connections to the same database. Refer to *Application Development Guide* for details on multiple thread database access.

4. Except for isolation level uncommitted read, described in "Uncommitted Read (UR)" on page 27.

The initiation and termination of a unit of work define points of consistency within an application process. For example, a banking transaction might involve the transfer of funds from one account to another.
Such a transaction would require that these funds be subtracted from the first
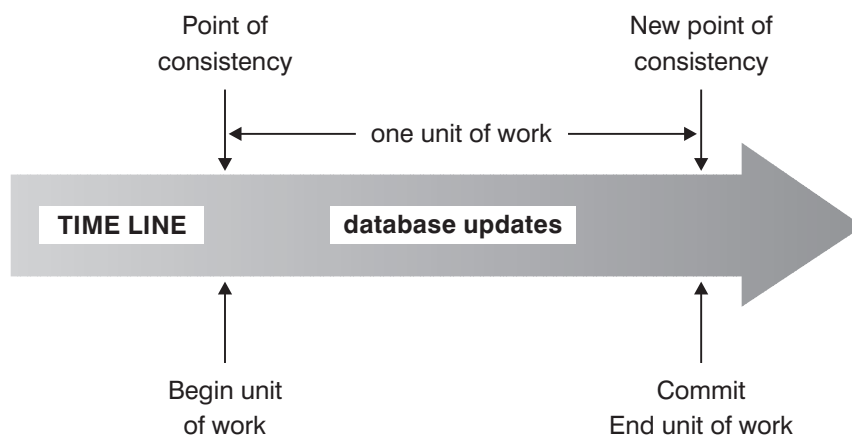


*Figure 1. Unit of Work with a Commit Statement*

account, and added to the second.
Following the subtraction step, the data is inconsistent. Only after the funds
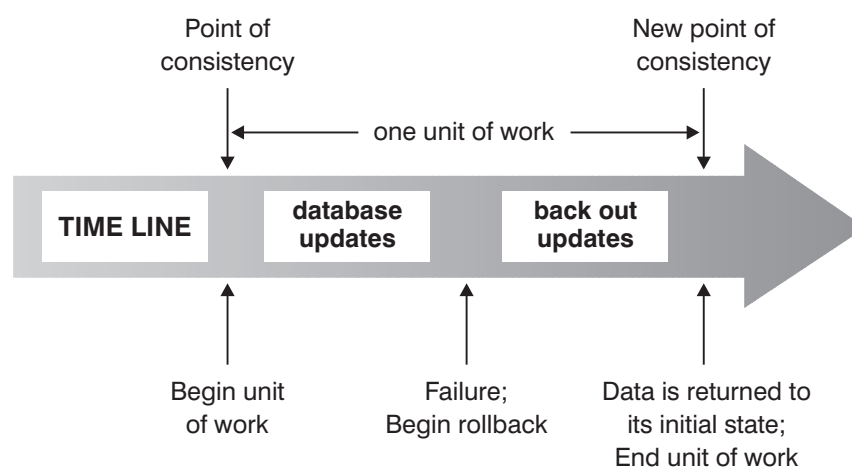


*Figure 2. Unit of Work with a Rollback Statement*

have been added to the second account is consistency reestablished. When both steps are complete, the commit operation can be used to end the unit of work, thereby making the changes available to other application processes.

If a failure occurs before the unit of work ends, the database manager will roll back uncommitted changes to restore the data consistency that it assumes existed when the unit of work was initiated.

**Note:** An application process is never prevented from performing operations because of its own locks.

## Isolation Level

The *isolation level* associated with an application process defines the degree of isolation of that application process from other concurrently executing application processes. The isolation level of an application process, P, therefore specifies:

- The degree to which rows read and updated by P are available to other concurrently executing application processes
- The degree to which update activity of other concurrently executing application processes can affect P.

Isolation level is specified as an attribute of a package and applies to the application processes that use the package. The isolation level is specified in the program preparation process. Depending on the type of lock, this limits or prevents access to the data by concurrent application processes. For details on different types and attributes of specific locks refer to *Administration Guide*. The database manager supports three general categories of locks:

| | |
|---|---|
| **Share** | Limits concurrent application processes to read-only operations on the data. |
| **Exclusive** | Prevents concurrent application processes from accessing the data in any way except for application processes with an isolation level of *uncommitted read*, which can read but not modify the data. (See "Uncommitted Read (UR)" on page 27.) |
| **Update** | Limits concurrent application processes to read-only operations on the data providing these processes have not declared they might update the row. The database manager assumes the process looking at the row presently may update the row. |

The following descriptions of isolation levels refer to locking data in row units. Logically, locking occurs at the base table row. The database manager,

however, can replace multiple row locks with a single table lock. This is called *lock escalation*. An application process is guaranteed at least the minimum requested lock level.

The DB2 Universal Database database manager supports four isolation levels. Regardless of the isolation level, the database manager places exclusive locks on every row that is inserted, updated, or deleted. Thus, all isolation levels ensure that any row that is changed by this application process during a unit of work is not changed by any other application processes until the unit of work is complete. The isolation levels are:

## Repeatable Read (RR)

Level RR ensures that:
- Any row read during a unit of work [5] is not changed by other application processes until the unit of work is complete. [6]
- Any row changed by another application process cannot be read until it is committed by that application process.

RR does not allow phantom rows (see Read Stability) to be seen.

In addition to any exclusive locks, an application process running at level RR acquires at least share locks on all the rows it references. Furthermore, the locking is performed so that the application process is completely isolated [6] from the effects of concurrent application processes.

## Read Stability (RS)

Like level RR, level RS ensures that:
- Any row read during a unit of work [5] is not changed by other application processes until the unit of work is complete [7]
- Any row changed by another application process cannot be read until it is committed by that application process.

Unlike RR, RS does not completely isolate the application process from the effects of concurrent application processes. At level RS, application processes that issue the same query more than once might see additional rows. These additional rows are called *phantom rows*.

For example, a phantom row can occur in the following situation:

---

5. The rows must be read in the same unit of work as the corresponding OPEN statement. See WITH HOLD in "DECLARE CURSOR" on page 731.

6. Use of the optional WITH RELEASE clause on the CLOSE statement means that any guarantees against non-repeatable read and phantoms no longer apply to any previously accessed rows if the cursor is reopened.

1. Application process P1 reads the set of rows *n* that satisfy some search condition.
2. Application process P2 then INSERTs one or more rows that satisfy the search condition and COMMITs those INSERTs.
3. P1 reads the set of rows again with the same search condition and obtains both the original rows and the rows inserted by P2.

In addition to any exclusive locks, an application process running at level RS acquires at least share locks on all the qualifying rows.

## Cursor Stability (CS)

Like the RR level:
- CS ensures that any row that was changed by another application process cannot be read until it is committed by that application process.

Unlike the RR level:
- CS only ensures that the current row of every updatable cursor is not changed by other application processes. Thus, the rows that were read during a unit of work can be changed by other application processes.

In addition to any exclusive locks, an application process running at level CS has at least a share lock for the current row of every cursor.

## Uncommitted Read (UR)

For a SELECT INTO, FETCH with a read-only cursor, fullselect used in an INSERT, row fullselect in an UPDATE, or scalar fullselect (wherever used), level UR allows:
- Any row that is read during the unit of work to be changed by other application processes.
- Any row that was changed by another application process to be read even if the change has not been committed by that application process.

For other operations, the rules of level CS apply.

## Comparison of Isolation Levels

A comparison of the four isolation levels can be found on "Appendix H. Comparison of Isolation Levels" on page 1083.

---

7. Use of the optional WITH RELEASE clause on the CLOSE statement means that any guarantees against non-repeatable read no longer apply to any previously accessed rows if the cursor is reopened.

## Distributed Relational Database

A *distributed relational database* consists of a set of tables and other objects that are spread across different but interconnected computer systems. Each computer system has a relational database manager to manage the tables in its environment. The database managers communicate and cooperate with each other in a way that allows a given database manager to execute SQL statements on another computer system.

Distributed relational databases are built on formal requester-server protocols and functions. An *application requester* supports the application end of a connection. It transforms a database request from the application into communication protocols suitable for use in the distributed database network. These requests are received and processed by an *application server* at the other end of the connection. Working together, the application requester and application server handle the communication and location considerations so that the application is isolated from these considerations and can operate as if it were accessing a local database. A simple distributed relational database environment is illustrated in Figure 3.
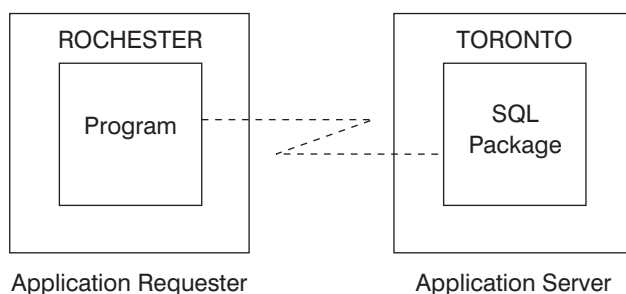


*Figure 3. A Distributed Relational Database Environment*

For more information on Distributed Relational Database Architecture (DRDA) communication protocols, see *Distributed Relational Database Architecture Reference* SC26-4651.

## Application Servers

An application process must be connected to the application server of a database manager before SQL statements that reference tables or views can be executed. A CONNECT statement establishes a connection between an application process and its server[3]. The server can change when a CONNECT statement is executed.

The application server can be local to or remote from the environment where the process is initiated. (An application server is present, even when not using distributed relational databases.) This environment includes a local directory that describes the application servers that can be identified in a CONNECT statement. For a description of local directories, see the *Administration Guide*

To execute a static SQL statement that references tables or views, the application server uses the bound form of the statement. This bound statement is taken from a package that the database manager previously created through a bind operation.

For the most part, an application can use the statements and clauses that are supported by the database manager of the application server to which it is currently connected, even though that application might be running via the application requester of a database manager that does not support some of those statements and clauses.

See the *IBM SQL Reference* for information on considerations for using a distributed relational database on DB2 application servers on other platforms. See "Server Definitions and Server Options" on page 42 for information about using an application server to submit queries in a system of distributed data sources.

## CONNECT (Type 1) and CONNECT (Type 2)

There are two types of CONNECT statements:
- CONNECT (Type 1) supports the single database per unit of work (Remote Unit of Work) semantics. See "CONNECT (Type 1)" on page 512.
- CONNECT (Type 2) supports the multiple database per unit of work (Application-Directed Distributed Unit of Work) semantics. See "CONNECT (Type 2)" on page 520.

## Remote Unit of Work

The *remote unit of work* facility provides for the remote preparation and execution of SQL statements. An application process at computer system A can connect to an application server at computer system B and, within one or more units of work, execute any number of static or dynamic SQL statements that reference objects at B. After ending a unit of work at B, the application process can connect to an application server at computer system C, and so on.

Most SQL statements can be remotely prepared and executed with the following restrictions:
- All objects referenced in a single SQL statement must be managed by the same application server

- All of the SQL statements in a unit of work must be executed by the same application server

## Remote Unit of Work Connection Management

This section outlines the connection states that an application process may enter.

*Connection States*:
> An application process is in one of four states at any time:
> > Connectable and connected
> > Unconnectable and connected
> > Connectable and unconnected
> > Implicitly connectable (if implicit connect is available).

> If implicit connect is available (see Figure 4 on page 32), the application process is initially in the *implicitly connectable* state. If implicit connect is not available (see Figure 5 on page 33), the application process is initially in the *connectable and unconnected* state.

> Availability of implicit connect is determined by installation options, environment variables, and authentication settings. See the *Quick Beginnings* for information on setting implicit connect on installation and the *Administration Guide* for information on environment variables and authentication settings.

*The implicitly connectable state*:

> If implicit connect is available, this is the initial state of an application process. The CONNECT RESET statement causes a transition to this state. Issuing a COMMIT or ROLLBACK statement in the unconnectable and connected state followed by a DISCONNECT statement in the connectable and connected state also results in this state.

*The connectable and connected state*:
> An application process is connected to an application server and CONNECT statements can be executed.
> If implicit connect is available:
> - The application process enters this state when a CONNECT TO statement or a CONNECT without operands statement is successfully executed from the connectable and unconnected state.
> - The application process may also enter this state from the implicitly connectable state if any SQL statement other than CONNECT RESET, DISCONNECT, SET CONNECTION, or RELEASE is issued.

> Whether or not implicit connect is available, this state is entered when:

– A CONNECT TO statement is successfully executed from the connectable and unconnected state.
– A COMMIT or ROLLBACK statement is successfully issued or a forced rollback occurs from the unconnectable and connected state.

*The unconnectable and connected state*:
An application process is connected to an application server, but a CONNECT TO statement cannot be successfully executed to change application servers. The process enters this state from the connectable and connected state when it executes any SQL statement other than the following statements: CONNECT TO, CONNECT with no operand, CONNECT RESET, DISCONNECT, SET CONNECTION, RELEASE, COMMIT or ROLLBACK.

*The connectable and unconnected state*:
An application process is not connected to an application server. The only SQL statement that can be executed is CONNECT TO, otherwise an error (SQLSTATE 08003) is raised.

Whether or not implicit connect is available:

– The application process enters this state if an error occurs when a CONNECT TO statement is issued or an error occurs in a unit of work which causes the loss of a connection and a rollback. An error caused because the application process is not in the connectable state or the server-name is not listed in the local directory does not cause a transition to this state.

If implicit connect is not available:

– the CONNECT RESET and DISCONNECT statements cause a transition to this state.

*State Transitions* are shown in the following diagrams.
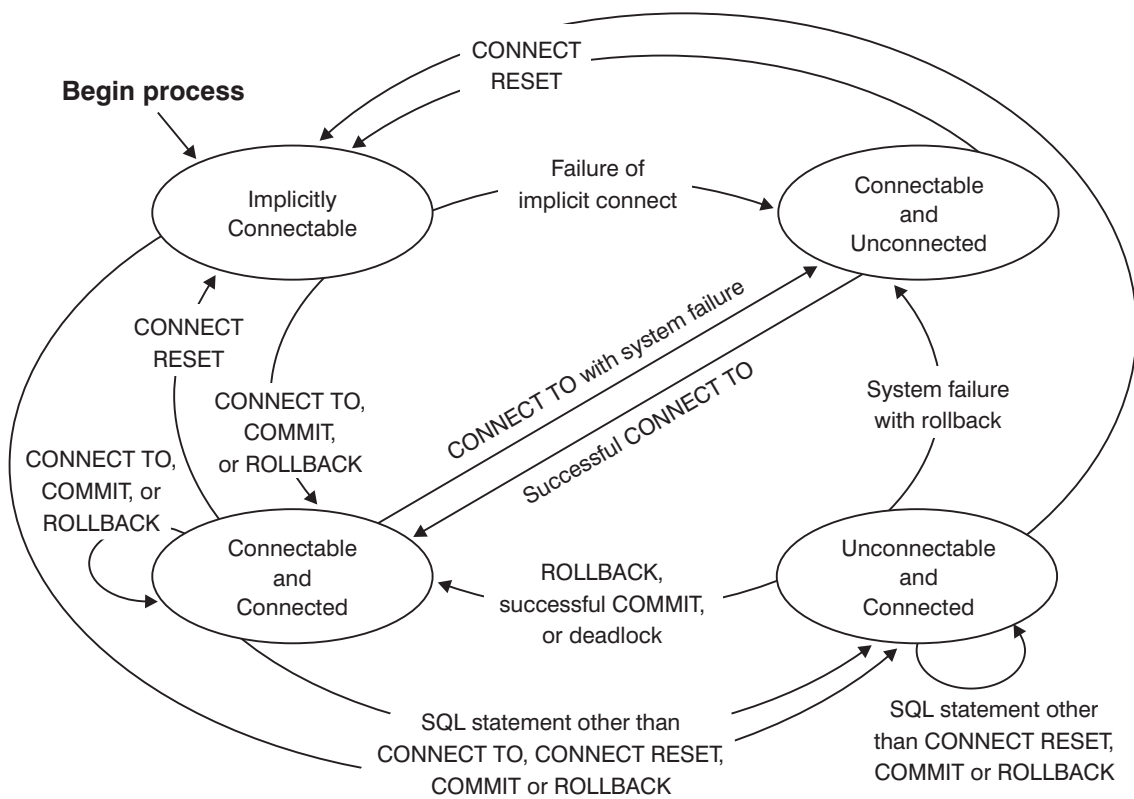
*Figure 4. Connection State Transitions If Implicit Connect Is Available*
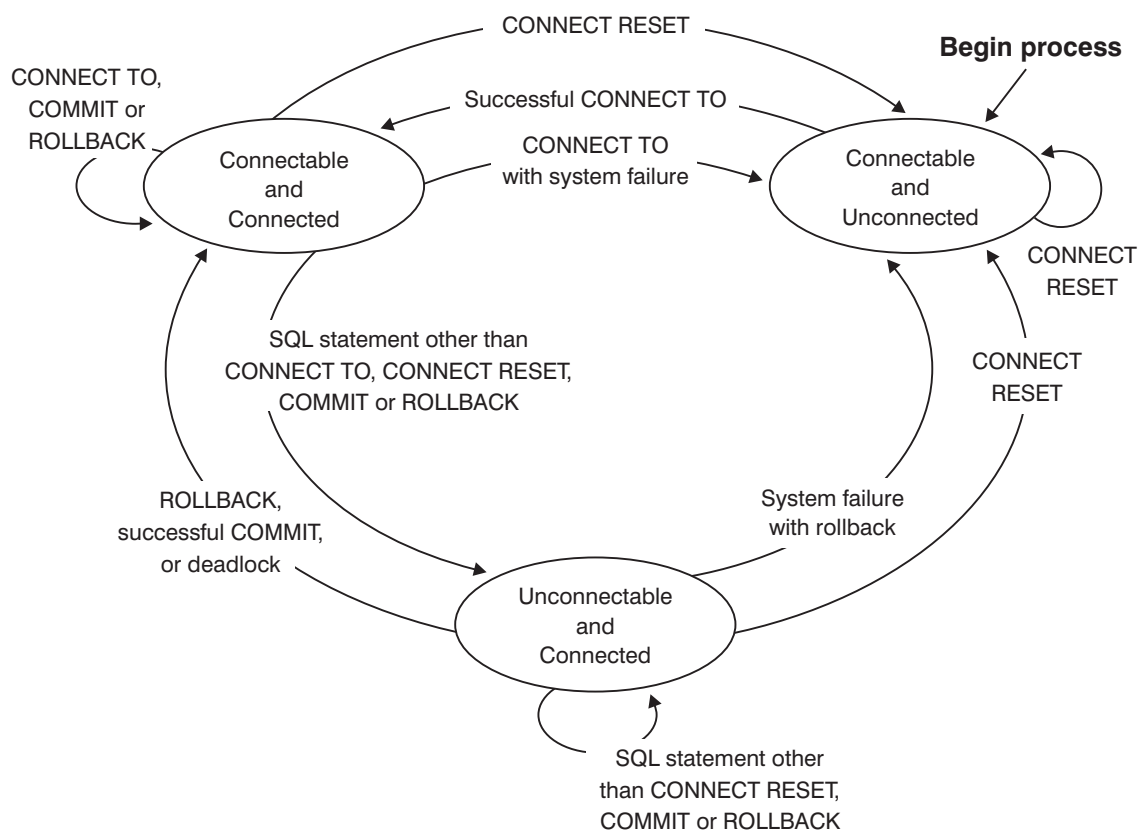
*Figure 5. Connection State Transitions If Implicit Connect Is Not Available*

**Additional Rules**:

- It is not an error to execute consecutive CONNECT statements because CONNECT itself does not remove the application process from the connectable state.

- It is an error to execute consecutive CONNECT RESET statements.

- It is an error to execute any SQL statement other than CONNECT TO, CONNECT RESET, CONNECT with no operand, SET CONNECTION, RELEASE, COMMIT, or ROLLBACK, and then execute a CONNECT TO statement. To avoid the error, a CONNECT RESET, DISCONNECT (preceded by a COMMIT or ROLLBACK statement), COMMIT, or ROLLBACK statement should be executed before executing the CONNECT TO.

## Application-Directed Distributed Unit of Work

The *application-directed distributed unit of work facility* also provides for the remote preparation and execution of SQL statements in the same fashion as remote unit of work. An application process at computer system A can connect to an application server at computer system B by issuing a CONNECT or SET CONNECTION statement. The application process can then execute any number of static and dynamic SQL statements that reference objects at B before ending the unit of work. All objects referenced in a single SQL statement must be managed by the same application server. However, unlike remote unit of work, any number of application servers can participate in the same unit of work. A commit or rollback operation ends the unit of work.

### Application-Directed Distributed Unit of Work Connection Management

An application-directed distributed unit of work uses a Type 2 connection. A Type 2 connection connects an application process to the identified application server and establishes the rules for application-directed distributed unit of work.

#### Overview of Application Process and Connection States

At any time a type 2 application process:

– Is always connectable

– Is in the *connected* state or *unconnected* state.

– Has a set of zero or more connections.

Each connection of an application process is uniquely identified by the database alias of the application server of the connection.

At any time an individual connection has one of the following sets of connection states:
– *current* and *held*
– *current* and *release-pending*
– *dormant* and *held*
– *dormant* and *release-pending*

**Initial States and State Transitions:** A type 2 application process is initially in the *unconnected state* and does not have any connections.

A connection initially is in the *current* and *held state*.

The following diagram shows the state transitions:
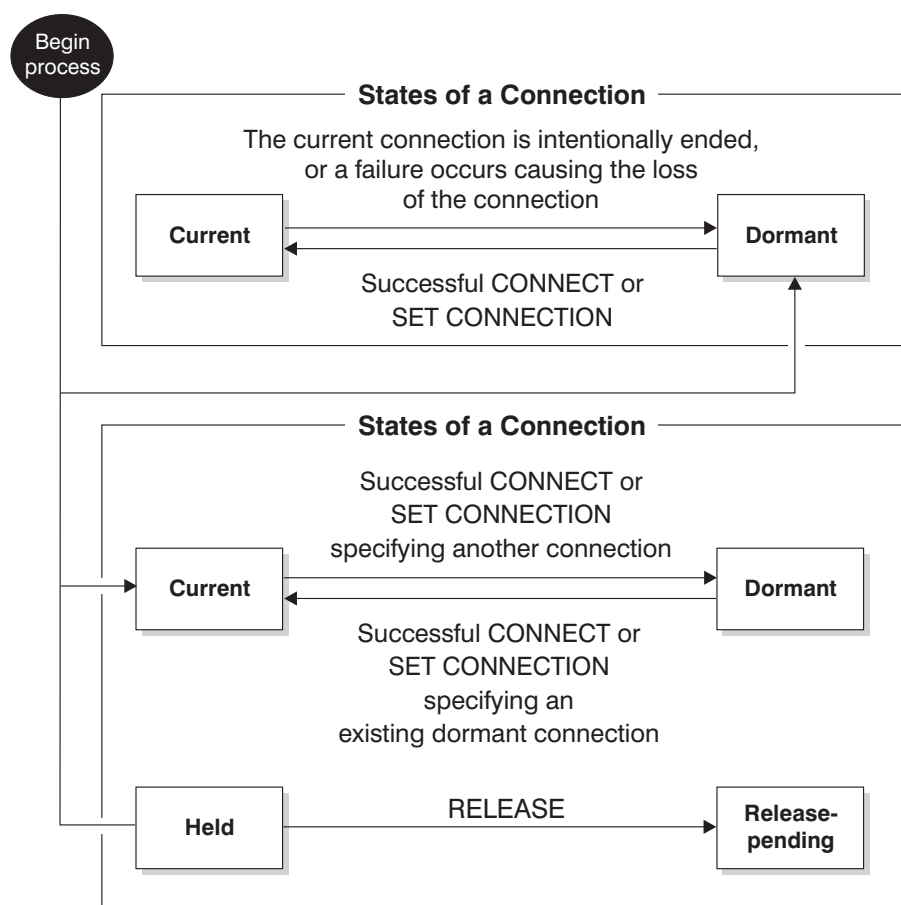
*Figure 6. Application-Directed Distributed Unit of Work Connection and Application Process Connection State Transitions*

**Application Process Connection States:**  A different application server can be established by the explicit or implicit execution of a CONNECT statement. [8] The following rules apply:

- A context can not have more than one connection to the same application server at the same time. See *Administration Guide* and *Application Development Guide* for information on support of multiple connections to the same DB2 Universal Database at the same time.

- When an application process executes a SET CONNECTION statement, the specified location name must be an existing connection in the set of connections of the application process.

---

8. Note that a Type 2 implicit connection is more restrictive than a Type 1. See "CONNECT (Type 2)" on page 520 for details.

- When an application process executes a CONNECT statement, and the SQLRULES(STD) option is in effect the specified server name must not be an existing connection in the set of connections of the application process. See "Options that Govern Distributed Unit of Work Semantics" on page 37 for a description of the SQLRULES option.

**If an application process has a current connection**, the application process is in the *connected* state. The CURRENT SERVER special register contains the name of the application server of the current connection. The application process can execute SQL statements that refer to objects managed by that application server.

An application process in the unconnected state enters the connected state when it successfully executes a CONNECT or SET CONNECTION statement. If there is no connection in the application but SQL statements are issued, an implicit connect will be made provided the DB2DBDFT environment variable has been defined with a default database.

**If an application process does not have a current connection**, the application process is in the *unconnected* state. The only SQL statements that can be executed are CONNECT, DISCONNECT ALL, DISCONNECT specifying a database, SET CONNECTION, RELEASE, COMMIT and ROLLBACK.

An application process in the *connected state* enters the *unconnected state* when its current connection is intentionally ended or the execution of an SQL statement is unsuccessful because of a failure that causes a rollback operation at the application server and loss of the connection. Connections are intentionally ended either by the successful execution of a DISCONNECT statement or by the successful execution of a commit operation when the connection is in the *release-pending state*. Different options specified in the DISCONNECT precompiler option affect intentionally ending a connection. If set to AUTOMATIC, then all connections are ended. If set to CONDITIONAL, then all connections that do not have open WITH HOLD cursors are ended.

**States of a Connection:** If an application process executes a CONNECT statement and the server name is known to the application requester and is not in the set of existing connections of the application process, then:
- the current connection is placed into the *dormant state*, and
- the server name is added to the set of connections, and
- the new connection is placed into both the *current state* and the *held state*.

If the server name is already in the set of existing connections of the application process and the application is precompiled with the option SQLRULES(STD), an error (SQLSTATE 08002) is raised.

- *Held and Release-pending States:* The RELEASE statement controls whether a connection is in the held or release-pending state. A *release-pending state* means that a disconnect is to occur for the connection at the next successful commit operation (a rollback has no effect on connections). A *held state* means that a connection is not to be disconnected at the next operation. All connections are initially in the held state and may be moved into the release-pending state using the RELEASE statement. Once in the release-pending state, a connection cannot be moved back to the held state. A connection will remain in a release-pending state across unit of work boundaries if a ROLLBACK statement is issued or if an unsuccessful commit operation results in a rollback operation.

  Even if a connection is not explicitly marked for release, it may still be disconnected by a commit operation if the commit operation satisfies the conditions of the DISCONNECT precompiler option.

- *Current and Dormant States:* Regardless of whether a connection is in the *held state* or the *release-pending state*, a connection can also be in the *current state* or *dormant state*. A *current state* means that the connection is the one used for SQL statements that are executed while in this state. A *dormant state* means that the connection is not current. The only SQL statements which can flow on a dormant connection are COMMIT and ROLLBACK; or DISCONNECT and RELEASE, which can specify either ALL (for all connections) or a specific database name. The SET CONNECTION and CONNECT statements change the connection for the named server into the *current state* while any existing connections are either placed or remain in the *dormant state*. At any point in time, only one connection can be in the *current state*. When a dormant connection becomes current in the same unit of work, the state of all locks, cursors, and prepared statements will remain the same and reflect their last use when the connection was current.

**When a Connection is Ended:**  When a connection is ended, all resources that were acquired by the application process through the connection and all resources that were used to create and maintain the connection are deallocated. For example, if the application process executes a RELEASE statement, any open cursors will be closed when the connection is ended during the next commit operation.

A connection can also be ended because of a communications failure. The application process is placed in the unconnected state if the connection ended was the current one.

All connections of an application process are ended when the process ends.

**Options that Govern Distributed Unit of Work Semantics**

The semantics of type 2 connection management are determined by a set of precompiler options. These are summarized briefly below with the defaults

indicated by bold and underlined text. For details refer to the *Command Reference* or *Administrative API Reference* manuals.

- CONNECT (**1** | 2)

  Specifies whether CONNECT statements are to be processed as type 1 or type 2.

- SQLRULES (**DB2** | STD)

  Specifies whether type 2 CONNECTs should be processed according to the DB2 rules which allow CONNECT to switch to a dormant connection, or the SQL92 Standard (STD) rules which do not allow this.

- DISCONNECT (**EXPLICIT** | CONDITIONAL | AUTOMATIC)

  Specifies what database connections are disconnected when a commit operation occurs. They are either:
  - those which had been explicitly marked for release by the SQL RELEASE statement (EXPLICIT), or
  - those that have no open WITH HOLD cursors as well as those marked for release (CONDITIONAL) [9] , or
  - all connections (AUTOMATIC).

- SYNCPOINT (**ONEPHASE** | TWOPHASE | NONE)

  Specifies how commits or rollbacks are to be coordinated among multiple database connections.

  ONEPHASE   Updates can only occur on one database in the unit of work, all other databases are read-only. Any update attempts to other databases raise an error (SQLSTATE 25000).

  TWOPHASE   A Transaction Manager (TM) will be used at run time to coordinate two phase commits among those databases that support this protocol.

  NONE       Does not use any TM to perform two phase commit and does not enforce single updater, multiple reader. When a COMMIT or ROLLBACK statement is executed, individual COMMITs or ROLLBACKs are posted to all databases. If one or more rollbacks fails an error (SQLSTATE 58005) is raised. If one or more commits fails an error (SQLSTATE 40003) is raised.

Any of the above options can be overridden at run time using a special SET CLIENT application programming interface (API). Their current settings can be obtained using the special QUERY CLIENT API. Note that these are not

---

9. The CONDITIONAL option will not work properly with downlevel servers prior to Version 2. A disconnection will occur in these cases regardless of the presence of WITH HOLD cursors

SQL statements; they are APIs defined in the various host languages and in the Command Line Processor. These are defined in the *Command Reference* and *Administrative API Reference* manuals.

## Data Representation Considerations

Different systems represent data in different ways. When data is moved from one system to another, data conversion sometimes must be performed. Products supporting DRDA will automatically perform any necessary conversions at the receiving system. With numeric data, the information needed to perform the conversion is the data type of the data and how that data type is represented by the sending system. With character data, additional information is needed to convert character strings. String conversion depends on both the code page of the data and the operation that is to be performed with that data. Character conversions are performed in accordance with the IBM Character Data Representation Architecture (CDRA). For more information on character conversion, refer to *Character Data Representation Architecture Reference* SC09-1390.

## DB2 Federated Systems

This section provides an overview of the elements of a DB2 federated system, an overview of the tasks that administrators and users of the system perform, and explanations of the concepts associated with these tasks.

### The Federated Server, Federated Database, and Data Sources

A *DB2 federated system* is a distributed computing system that consists of:
- A DB2 server, called a *federated server*.

  In a DB2 installation, any number of DB2 instances can be configured to function as federated servers.
- Multiple data sources to which the federated server sends queries.

  Each data source consists of an instance of a relational database management system plus the database or databases that the instance supports. The data sources in a DB2 federated system can include Oracle instances and instances of the members of the DB2 family.

  The data sources are semi-autonomous. For example, the federated server can send queries to Oracle data sources at the same time that Oracle applications can access these data sources. A DB2 federated system does not monopolize or restrict access to Oracle or other data sources (beyond integrity and locking constraints).

To end users and client applications, the data sources appear as a single collective database. In actuality, users and applications interface with a database, called the *federated database*, that is within the federated server. To

obtain data from data sources, they submit queries in DB2 SQL to the federated database. DB2 then distributes the queries to the appropriate data sources. DB2 also provides access plans for optimizing the queries (in some cases, these plans call for processing the queries at the federated server rather than at the data source). Finally, DB2 collects the requested data and passes it to the users and applications.

Queries submitted from the federated server to data sources must be read-only. To write to a data source (for example, to update a data source table), users and applications must use the data source's own SQL in a special mode called *pass-through*.

## Tasks to Perform in a DB2 Federated System

This section introduces concepts that are associated with tasks that users perform to establish and use a federated system. (In this section and those that follow, the term *users* refers to all types of personnel who work with federated systems; for example, database administrators [DBAs], application programmers, and end users).

The following list of tasks identifies the types of users who typically perform the tasks. Be aware that other types of users can also perform these tasks. For example, the list indicates that DBAs typically create mappings between authorizations to access the federated database and authorizations to access data sources. But application programmers and end users can also perform this task.

To establish and use a DB2 federated system:

1. The DBA designates a DB2 server as a federated server. Refer to the *Installation and Configuration Supplement* for information about how this is done.
2. Data sources are set up for access:
   a. The DBA connects to the federated database.
   b. The DBA creates a wrapper for each category of data source that is to be included in the federated system. (*Wrappers* are mechanisms by which the federated server interacts with data sources. Refer to "Wrappers and Wrapper Modules" on page 42 for details.)
   c. The DBA supplies the catalog with a description of each data source. (The description is called a *server definition*. Refer to "Server Definitions and Server Options" on page 42 for details.)
   d. If a user's authorization to access the federated database differs from the user's authorization to access a data source, the DBA defines an association between the two authorizations. (This association is called a *user mapping*. Refer to "User Mappings and User Options" on page 44 for details.)

e. If a default mapping between a DB2 data type and a data source data type does not meet user requirements, the DBA modifies the mapping as needed. (A *data type mapping* is a defined association between two compatible data types—one supported by the federated database and one supported by a data source. Refer to "Data Type Mappings" on page 45 for details.)

f. If a default mapping between a DB2 function and a data source function does not meet user requirements, the DBA modifies the mapping as needed. (A *function mapping* is a defined association between two compatible functions—one supported by the federated database and one supported by a data source. Refer to "Function Mappings, Function Templates, and Function Mapping Options" on page 46 for details.)

g. DBAs and application programmers create nicknames for the data source tables and views that are to be accessed. (A *nickname* is an identifier by which the federated system references a data source table or view. Refer to "Nicknames and Column Options" on page 47 for details.)

h. Optional: If a data source table has no index, the DBA can provide the federated server with the same sort of information that the definition of an actual index would contain. If a data source table has an index that the federated server is unaware of, the DBA can inform the server of the index's existence. In either case, the information that the DBA supplies helps DB2 to optimize queries of table data. (This information is called an *index specification*. Refer to "Index Specifications" on page 48 for details.)

3. Application programmers and end users retrieve information from data sources:

   • Using DB2 SQL, application programmers and end users query tables and views that are referenced by nicknames. (Queries directed to two or more data sources are called *distributed requests*. Refer to "Distributed Requests" on page 48 for details. )

   In processing a query, the federated server can perform operations that are supported by DB2 SQL but not by the data source's SQL. (This capability is called *compensation*. Refer to "Compensation" on page 49 for details.)

   • Application programmers and end users might occasionally submit queries, DML statements, and DDL statements to data sources in the data sources' own SQL. The programmers and users can do this in *pass-through* mode. (Refer to "Pass-Through" on page 50 for details.)

The following sections discuss the concepts mentioned in this task list in the same order in which they appear in the list. Some of these sections also introduce related concepts.

## Wrappers and Wrapper Modules

A wrapper is the mechanism by which the federated server communicates with, and retrieves data from, a data source. To implement a wrapper, the server uses routines stored in a library called a *wrapper module*. These routines allow the server to perform operations such as connecting to a data source and retrieving data from it iteratively.

There are three wrappers:
- A wrapper with the default name of DRDA is used for all DB2 family data sources.
- A wrapper with the default name of SQLNET is used for all Oracle data sources supported by Oracle's SQL*Net client software.
- A wrapper with the default name of NET8 is used for all Oracle data sources supported by Oracle's Net8 client software.

A wrapper is registered to the federated server with the CREATE WRAPPER statement. Refer to "CREATE WRAPPER" on page 729 for details.

## Server Definitions and Server Options

After the DBA registers a wrapper that allows the federated server to interact with data sources, the DBA defines those data sources to the federated database. This section:
- Describes the definition that the DBA provides, and explains why it is called a *server definition*
- Describes the SQL for specifying certain parameters, called *server options*, that contain portions of a server definition
- Distinguishes different meanings of the term "server".

### Introduction to Server Definitions

In defining a data source to the federated database, the DBA supplies a name for the data source as well as information that pertains to the data source. This information includes the type and version of the RDBMS of which the data source is an instance, and the RDBMS's name for the data source. It also includes metadata that is specific to the RDBMS. For example, a DB2 family data source can have multiple databases, and the definition of such a data source must specify which database the federated server can connect to. In contrast, an Oracle data source has one database, and the federated server can connect to the database without needing to know its name. Thus, the name is not included in the federated server's definition of the data source.

The name and information that the DBA supplies is collectively called a *server definition*. This term reflects the fact that data sources answer requests for data and are therefore servers in their own right. Other terms reflect this fact also. For example:

* Some of the information within a server definition is stored as values of parameters called *server options*. Thus, the name for a data source instance is stored as a value of a server option called NODE. For a DB2 family data source, the name of the database to which the federated server connects is stored as a value of a server option called DBNAME.
* The SQL statements for creating and modifying a server definition are called CREATE SERVER and ALTER SERVER, respectively.

**SQL Statements for Setting Server Options**

Values are assigned to server options through the CREATE SERVER, ALTER SERVER, and SET SERVER OPTION statements.

The CREATE SERVER and ALTER SERVER statements set server options to values that persist over successive connections to the data source. These values are stored in the catalog. Consider this scenario: A federated system DBA uses the CREATE SERVER statement to define a new Oracle data source to the federated system. This data source's database uses the same collating sequence that the federated database uses. The DBA wants the optimizer to know about this match, so that the optimizer can take advantage of it to expedite performance. Accordingly, in the CREATE SERVER statement, the DBA sets a server option called COLLATING_SEQUENCE to 'Y' (yes, the collating sequences at the data source and federated database are the same). The setting of 'Y' is recorded in the catalog, and it remains in force while users and applications access the data source.

Some months later, the Oracle DBA changes the Oracle data source's collating sequence. Therefore, the federated system DBA resets COLLATING_SEQUENCE to 'N' (no, the data source's collating sequence is not the same as the federated database's). The DBA uses the ALTER SERVER statement to make this update. The catalog is updated also, and the new setting remains in effect as users and applications continue to access the data source.

The SET SERVER OPTION statement overrides a server option value temporarily, for the duration of a single connection to the federated database. The overriding value does not get stored in the catalog.

To illustrate: A server option called PLAN_HINTS can be set to a value that enables DB2 to supply Oracle data sources with statement fragments, called plan hints, that help Oracle optimizers to do their job. For example, plan hints

can help an optimizer to decide what index to use in accessing a table, or
what table join sequence to use in retrieving data for a result set.

Consider another scenario: For data sources ORACLE1 and ORACLE2, the
PLAN_HINTS server option is set to its default, 'N' (no, do not furnish these
data sources with plan hints). Then a programmer writes a distributed request
for data from ORACLE1 and ORACLE2; and the programmer expects that
plan hints would help the optimizers at these data sources to improve their
strategies for accessing this data. Accordingly, the programmer uses the SET
SERVER OPTION statement to override the 'N' with 'Y' (yes, furnish plan
hints). The 'Y' stays in effect only while the application that contains the
request is connected to the federated database; it does not get stored in the
catalog.

Refer to "CREATE SERVER" on page 635, "ALTER SERVER" on page 445, and
"SET SERVER OPTION" on page 897 for more information. Refer to "Server
Options" on page 1049 for descriptions of all server options and their settings.

### Three Meanings for "Server"

In the terms "server definition" and "server option", and in the SQL
statements discussed in the preceding section, the word "server" refers to data
sources only. It does not refer to the federated server, or to DB2 application
servers.

The concepts of DB2 application servers and federated servers, however,
overlap. As indicated in "Distributed Relational Database" on page 28, an
application server is a database manager instance to which application
processes connect and submit requests. This is true of a federated server; thus,
a federated server is a type of application server. But two main things
distinguish it from other application servers:

- It is configured to receive requests that are ultimately intended for data
  sources; and it distributes these requests to the data sources.
- Like other application servers, a federated server uses drda communication
  protocols to communicate with DB2 family instances. Unlike other
  application servers, a federated server uses sqlnet and net8 communication
  protocols to communicate with Oracle instances.

## User Mappings and User Options

The federated server can send the distributed request of an authorized user or
application to a data source under either of these conditions:

- The user or application uses the same user ID for both the federated
  database and the data source. In addition, if the data source requires a
  password, the user or application uses the same password for the federated
  database and the data source.

- The user's or application's authorization to access the federated database differs in some way from the user's or application's authorization to access the data source. In addition, when the user or application requests access to the data source, the federated database authorization is changed to the data source authorization, so that the access can be granted. This change can occur only if a defined association, called a *user mapping*, exists between the two authorizations.

User mappings can be defined and modified with the CREATE USER MAPPING and ALTER USER MAPPING statements. These statements include parameters, called *user options*, to which values related to authorization are assigned. For example, suppose that a user has the same ID, but different passwords, for the federated database and a data source. For the user to access the data source, it is necessary to map the passwords to one another. This can be done with a CREATE USER MAPPING statement in which the password at the data source is assigned as a value to a user option called REMOTE_PASSWORD.

Refer to "CREATE USER MAPPING" on page 712, "ALTER USER MAPPING" on page 477, and *Administration Guide, Design and Implementation* for more information. Refer to "User Options" on page 1053 for descriptions of the user options and their settings.

## Data Type Mappings

For the federated server to retrieve data from columns of data source tables and views, the columns' data types at the data source must map to corresponding data types that are already defined to the federated database. DB2 supplies default mappings for most kinds of data types. For example, the Oracle type FLOAT maps by default to the DB2 type DOUBLE, and the DB2 Universal Database for OS/390 type DATE maps by default to the DB2 type DATE. There are no mappings for the data types that DB2 federated servers do not support: LONG VARCHAR, LONG VARGRAPHIC, DATALINK, large object (LOB) types, and user-defined types.

Refer to "Default Data Type Mappings" on page 1054 for listings of the default data type mappings.

When values from a data source column are returned, they conform fully to the DB2 type in the type mapping that applies to the column. If this mapping is a default, the values also conform fully to the data source type in the mapping. For example, when an Oracle table with a FLOAT column is defined to the federated database, the default mapping of Oracle FLOAT to DB2 DOUBLE will, unless it has been overridden, automatically apply to that column. Consequently, the values returned from the column will conform fully to both FLOAT and DOUBLE.

It is possible to change the format or length of values returned by changing the DB2 type that the values must conform to. For example, the Oracle type DATE is for time stamps. By default, it maps to the DB2 type TIMESTAMP. Suppose that several Oracle table columns have a data type of DATE, and that a user wants queries of these columns to yield times only. The user could then map the Oracle type DATE to the DB2 type TIME, overriding the default. That way, when the columns are queried, only the time portion of the time stamps would be returned.

The CREATE TYPE MAPPING statement can be used to create a modified data type mapping that applies to one or more data sources. The ALTER NICKNAME statement can be used to modify a data type mapping for a specific column of a specific table.

Refer to "CREATE TYPE MAPPING" on page 707, "ALTER NICKNAME" on page 437, and the *Application Development Guide* for more information.

## Function Mappings, Function Templates, and Function Mapping Options

For the federated server to recognize a data source function, there needs to be a mapping between this function and a corresponding DB2 function that already exists at the server. DB2 supplies default mappings between existing built-in data source functions and built-in DB2 functions. If a user wants to use a data source function that the federated server does not recognize—for example, a new built-in function or a user-defined function—then the user must create a mapping between this function and a counterpart at the federated database. If a counterpart does not exist, the user must create one that meets the following requirements:

- If the data source function has input parameters, the counterpart must have the same number of input parameters that the data source function has. If the data source function has no input parameters, the counterpart cannot have any.
- The counterpart's data types for input parameters (if any) and returned values must be compatible with the data source function's corresponding data types.

The counterpart can be either a complete function or a function template. A *function template* is a partial function that has no executable code. It cannot be invoked independently; its only purpose is to participate in a mapping with a data source function, so that the data source function can be invoked from the federated server.

Function mappings are created with the CREATE FUNCTION MAPPING statement. This statement includes parameters, called *function mapping options*, to which the user can assign values that pertain to the mapping being created

or to the data source function within the mapping. Such values, for example, can include estimated statistics on the overhead that would be consumed when the data source function is invoked. The optimizer uses these estimates in developing strategies for invoking the function.

Refer to "CREATE FUNCTION (Source or Template)" on page 593 and "CREATE FUNCTION MAPPING" on page 603 for details about creating function templates and function mappings. Refer to "Function Mapping Options" on page 1048 for descriptions of the function mapping options and their values. Refer to the *Application Development Guide* for guidelines on optimizing the invocation of data source functions.

## Nicknames and Column Options

When a client application submits a distributed request to the federated server, the server parcels out the request to the appropriate data sources. The request does not need to specify these data sources. Instead, it references data source tables and views by identifiers, called *nicknames*, that map to the tables' and views' names at the data source. The mappings obviate the need to qualify the nicknames by data source names. The locations of the tables and views are transparent to the client application.

Nicknames are not alternate names for tables and views in the same way that aliases are; they are pointers by which the federated server references these objects. Nicknames are defined with the CREATE NICKNAME statement. Refer to "CREATE NICKNAME" on page 614 for details.

When a nickname is created for a table or view, the catalog is populated with metadata that the optimizer can use to facilitate access to the table or view. For example, the catalog is supplied with the names of the DB2 data types to which the data types of the table's or view's columns map. If the nickname is for a table with an index, the catalog is supplied also with information related to the index; for example, the name of each column in the index key.

After a nickname is created, the user can supply the catalog with more metadata for the optimizer; for example, metadata that describes values in certain columns of the table or view that the nickname references. The user assigns this metadata to parameters called *column options*. To illustrate: If a table column contains numeric strings only, the user can indicate this by assigning the value 'Y' to a column option called NUMERIC_STRING. As a result, the optimizer can form strategies to have these strings sorted at the data source, thereby saving the overhead of porting them to the federated server and sorting them there. The savings is especially great when the database that contains the values has a collating sequence that differs from the federated database's collating sequence.

Column options are defined with the ALTER NICKNAME statement. Refer to "ALTER NICKNAME" on page 437 for more information about this statement. Refer to "Column Options" on page 1047 for descriptions of the column options and their settings.

## Index Specifications

When a nickname is created for a data source table, the federated server supplies the catalog with information about any indexes that a data source table has. The optimizer uses this information to facilitate retrieval of the table's data. If the table has no indexes, the user can nevertheless supply information that an index definition typically contains; for example, which column or columns in the table to search in order to find information quickly. The user would do this by running a CREATE INDEX statement that contains the information and references the table's nickname.

The user can supply the optimizer with similar information for tables that have indexes of which the federated server is unaware. For example, suppose that nickname NICK1 is created for a table that has no index but that acquires one later, or that nickname NICK2 is created for a view over a table that has an index. In these situations, the federated server would be unaware of the indexes. But the user could use CREATE INDEX statements to inform the server that the indexes exist. One statement would reference NICK1 and contain information about the index of the table that NICK1 identifies. The other would reference NICK2 and contain information about the index of the base table that underlies the view that NICK2 identifies.

In cases such as those just described, the information in the CREATE INDEX statement is cataloged as a set of metadata called an *index specification.* Be aware that when the statement references a nickname, it produces only an index specification, not an actual index. Refer to "CREATE INDEX" on page 608 and the *Administration Guide, Performance* for more information.

## Distributed Requests

A distributed request can use devices such as subqueries and join subselects to specify what table or view columns are be accessed, and what data is to be retrieved.

This section provides examples within the context of the following scenario: A federated server is configured to access a DB2 Universal Database for OS/390 data source, a DB2 Universal Database for AS/400 data source, and an Oracle data source. Stored in each data source is a table that contains employee information. The federated server references these tables by nicknames that refer to where the tables reside: UDB390_EMPLOYEES, AS400_EMPLOYEES,

and ORA_EMPLOYEES. In addition to its table of employee information, the Oracle data source has a table that contains information about the countries that the employees live in. The nickname for this second table is ORA_COUNTRIES.

### A Request with a Subquery

Table AS400_EMPLOYEES contains the phone numbers of employees who live in Asia. It also contains the country codes associated with these phone numbers, but it doesn't list the countries that the codes represent. Table ORA_COUNTRIES, however, does list both codes and countries. The following query uses a subquery to find out the country code for China; and it uses SELECT and WHERE clauses to list those employees in AS400_EMPLOYEES whose phone numbers require this particular code.

```
SELECT NAME, TELEPHONE
  FROM DJADMIN.AS400_EMPLOYEES
  WHERE COUNTRY_CODE IN
  (SELECT COUNTRY_CODE
     FROM DJADMIN.ORA_COUNTRIES
     WHERE COUNTRY_NAME = 'CHINA')
```

When a distributed request such as the one above is compiled, the compiler's query rewrite facility transforms it into a form that can be optimized more easily.

### A Request for a Join

A relational join produces a result set that contains a combination of columns retrieved from two or more tables. Conditions should always be specified to limit the size of the result set's rows.

The query below combines employee names and their corresponding country names by comparing the country codes listed in two tables. Each table resides in a different data source.

```
SELECT T1.NAME, T2.COUNTRY_NAME
  FROM DJADMIN.UDB390_EMPLOYEES T1, DJADMIN.ORA_COUNTRIES T2
  WHERE T1.COUNTRY_CODE = T2.COUNTRY_CODE
```

## Compensation

*Compensation* is the processing of SQL statements for RDBMSs that do not support those statements. Each type of RDBMS (DB2 Universal Database for AS/400, DB2 Universal Database for OS/390, Oracle, and so on) supports a subset of the international standard of SQL. In addition, some types support SQL constructs that exceed this standard. The totality of SQL that a type of RDBMS supports is called an *SQL dialect*. If an SQL construct is found in DB2's SQL dialect, but not in a data source's dialect, the federated server can implement this construct on behalf of the data source.

*Example 1:* DB2's SQL includes the clause, common-table-expression. In this clause, a name can be specified by which all FROM clauses in a fullselect can reference a result set. The federated server will process a common-table-expression for an Oracle database, even though Oracle's SQL dialect does not include common-table-expression.

*Example 2:* When connecting to a data source that does not support multiple open cursors within an application, the federated server can simulate this function by establishing separate, simultaneous connections to the data source. Similarly, the federated server can simulate CURSOR WITH HOLD capability for a data source that does not provide that function.

Compensation makes it possible to use DB2's SQL dialect to make all queries supported by the federated server. It is not necessary to use dialects specific to RDBMSs other than DB2.

## Pass-Through

Users can use the pass-through function to communicate with data sources in the data sources' own SQL dialect. In pass-through, users can submit not only queries, but also DML and DDL statements. Refer to "SQL Processing in Pass-Through Sessions" on page 1056 for information on how DB2 and data sources manage the processing of statements submitted in pass-through sessions.

The federated server provides the following SQL statements to manage pass-through sessions:

SET PASSTHRU
> Opens and terminates pass-through sessions.

GRANT (Server Privileges)
> Grants a user, group, list of authorization IDs, or PUBLIC the authority to initiate pass-through sessions to a specific data source.

REVOKE (Server Privileges)
> Revokes the authority to initiate pass-through sessions.

There are certain restrictions on using pass-through. For example, in a pass-through session, a cursor cannot be opened directly against a data source object. Refer to "Considerations and Restrictions" on page 1056 for a complete list of restrictions.

## Character Conversion

A *string* is a sequence of bytes that may represent characters. Within a string, all the characters are represented by a common coding representation. In some cases, it might be necessary to convert these characters to a different coding representation. The process of conversion is known as *character conversion*. [10]

Character conversion can occur when an SQL statement is executed remotely. Consider, for example, these two cases:
- The values of host variables sent from the application requester to the application server
- The values of result columns sent from the application server to the application requester.

In either case, the string could have a different representation at the sending and receiving systems. Conversion can also occur during string operations on the same system.

The following list defines some of the terms used when discussing character conversion.

character set
A defined set of characters. For example, the following character set appears in several code pages:
- 26 non-accented letters A through Z
- 26 non-accented letters a through z
- digits 0 through 9
- . , : ; ? ( ) ' " / - _ & + % * = < >

code page
A set of assignments of characters to code points. In the ASCII encoding scheme for code page 850, for example, "A" is assigned code point X'41' and "B" is assigned code point X'42'. Within a code page, each code point has only one specific meaning. A code page is an attribute of the database. When an application program connects to the database, the database manager determines the code page of the application.

---

10. Character conversion, when required, is automatic and is transparent to the application when it is successful. A knowledge of conversion is therefore unnecessary when all the strings involved in a statement's execution are represented in the same way. This is frequently the case for *stand-alone* installations and for networks within the same country. Thus, for many readers, character conversion may be irrelevant.

<table>
<tr><td>**code point**</td><td>A unique bit pattern that represents a character.</td></tr>
<tr><td>**encoding scheme**</td><td>A set of rules used to represent character data. For example:
<ul>
<li>Single-Byte ASCII</li>
<li>Single-Byte EBCDIC</li>
<li>Double-Byte ASCII</li>
<li>Mixed Single- and Double-Byte ASCII.</li>
</ul>
</td></tr>
</table>

## Character Sets and Code Pages

The following example shows how a typical character set might map to different code points in two different code pages.

code page: pp1 (ASCII)

|   | 0 | 1 | 2 | 3 | 4 | 5 |   | E | F |
|---|---|---|---|---|---|---|---|---|---|
| 0 |   |   |   | 0 | @ | P |   | Â |   |
| 1 |   |   |   | 1 | A | Q |   | À | α |
| 2 |   |   | " | 2 | B | R |   | Å | β |
| 3 |   |   |   | 3 | C | S |   | Á | γ |
| 4 |   |   |   | 4 | D | T |   | Ã | δ |
| 5 |   |   | % | 5 | E | U |   | Ä | ε |
| E |   |   | · | > | N |   |   | ⅝ | Ö |
| F |   |   | / | * | 0 |   |   |   | ® |

code point: 2F

character set ss1 (in code page pp1)

code page: pp2 (EBCDIC)

|   | 0 | 1 |   | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|
| 0 |   |   |   |   | # |   |   |   | 0 |
| 1 |   |   |   |   | $ | A | J |   | 1 |
| 2 |   |   |   | s | % | B | K | S | 2 |
| 3 |   |   |   | t | ¬ | C | L | T | 3 |
| 4 |   |   |   | u | * | D | M | U | 4 |
| 5 |   |   |   | v | ( | E | N | V | 5 |
| E |   |   |   |   | ! | : | Â | } |   |
| F |   |   |   | À | ¢ | ; | Á | { |   |

character set ss1 (in code page pp2)

Even with the same encoding scheme, there are many different code pages, and the same code point can represent a different character in different code pages. Furthermore, a byte in a character string does not necessarily represent a character from a single-byte character set (SBCS). Character strings are also used for mixed and bit data. *Mixed* data is a mixture of single-byte, double-byte, or multi-byte characters. *Bit* data (columns defined as FOR BIT DATA or BLOBs, or binary strings) is not associated with any character set.

## Code Page Attributes

The database manager determines code page attributes for all character strings when an application is bound to a database. The potential code page attributes are:

**The Database Code Page**    The database code page stored in the database configuration files. This code page value is determined when the database is created and cannot be altered.

**The Application Code Page**    The code page under which the application is executed. Note that this is not necessarily the same code page under which the application was bound. (See the *Application Development Guide* for further information on binding and executing application programs.)

**Code Page 0**    This represents a string that is derived from an expression that contains a FOR BIT DATA or BLOB value.

### String Code Page Attributes

Character string code page attributes are as follows:
- Columns may be in the database code page or code page 0 (if defined as character FOR BIT DATA or BLOB).
- Constants and special registers (for example, USER, CURRENT SERVER) are in the database code page. Note that constants are converted to the database code page when an SQL statement is bound to the database.
- Input host variables are in the application code page.

A set of rules is used to determine the code page attributes for operations that combine string objects, such as the results of scalar operations, concatenation, or set operations. At execution time, code page attributes are used to determine any requirements for code page conversions of strings.

For more details on character conversion, see:

- "Conversion Rules for String Assignments" on page 94 for rules on string assignments
- "Rules for String Conversions" on page 108 for rules on conversions when comparing or combining character strings.

## Authorization and Privileges

An *authorization* allows a user or group to perform a general task such as connecting to a database, creating tables, or administering a system. A *privilege* gives a user or group the right to access one specific database object in a specified way.

The database manager requires that a user be specifically authorized, either implicitly or explicitly, [11] to use each database function needed by that user to perform a specific task. Thus to create a table, a user must be authorized to create tables; to alter a table, a user must be authorized to alter the table; and so on.

```
                    ┌─────────────────────────────┐
                    │          SYSADM             │
                    │   (System Administrator)     │
                    └─────────────────────────────┘
                          │                 │
        ┌─────────────────────────┐   ┌──────────────────────────────────┐
        │         DBADM           │   │           SYSCTRL                 │
        │ (Database Administrator) │   │ (System Resource Administrator)   │
        └─────────────────────────┘   └──────────────────────────────────┘
                    │                              │
                    │               ┌──────────────────────────────────────┐
                    │               │             SYSMAINT                  │
                    │               │ (System Maintenance Administrator)     │
                    │               └──────────────────────────────────────┘
    ┌┬┬┬┬┬┬┬┬┬┬┬┬┬┬┬┬┬┐
    └┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┘
    Database Users with Privileges
```
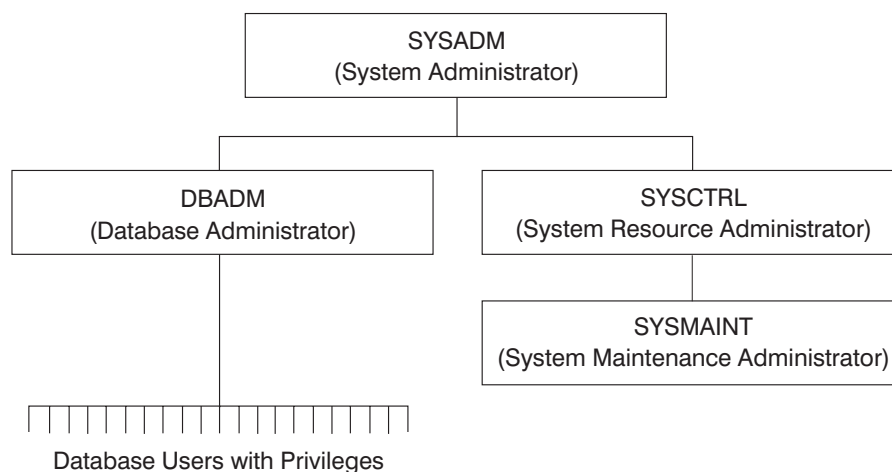
*Figure 7. Hierarchy of Authorities and Privileges*

The person or persons with administrative authority have the task of controlling the database manager and are responsible for the safety and integrity of the data. They control who will have access to the database manager and to what extent each user has access.

---

11. Explicit authorities or privileges are granted to the user (GRANTEETYPE of U). Implicit authorities or privileges are granted to a group to which the user belongs (GRANTEETYPE of G).

The database manager provides two administrative authorities:

**SYSADM**
> System administrator authority

**DBADM**
> Database administrator authority

and two system control authorities:

**SYSCTRL**
> System control authority

**SYSMAINT**
> System maintenance authority

**SYSCTRL**
> System control authority

**SYSMAINT**
> System maintenance authority

SYSADM authority is the highest level of authority and has control over all the resources created and maintained by the database manager. SYSADM authority includes all the privileges of DBADM, SYSCTRL, and SYSMAINT, and the authority to grant or revoke DBADM authorities.

DBADM authority is the administrative authority specific to a single database. This authority includes privileges to create objects, issue database commands, and access the data in any of its tables through SQL statements. DBADM authority also includes the authority to grant or revoke CONTROL and individual privileges.

SYSCTRL authority is the higher level of system control authority and applies only to operations affecting system resources. It does not allow direct access to data. This authority includes privileges to create, update, or drop a database; quiesce an instance or database; and drop or create a table space.

SYSMAINT authority is the second level of system control authority. A user with SYSMAINT authority can perform maintenance operations on all databases associated with an instance. It does not allow direct access to data. This authority includes privileges to update database configuration files, backup a database or table space, restore an existing database, and monitor a database.

Database authorities apply to those activities that an administrator has allowed a user to perform within the database that do not apply to a specific instance of a database object. For example, a user may be granted the authority to create packages but not create tables.

Privileges apply to those activities that an administrator or object owner has allowed a user to perform on database objects. Users with privileges can create objects, though they face some constraints, unlike a user with an authority like SYSADM or DBADM. For example, a user may have the privilege to create a view on a table but not a trigger on the same table. Users with privileges have access to the objects they own, and can pass on privileges on their own objects to other users by using the GRANT statement.

CONTROL privilege allows the user to access a specific database object as desired and to GRANT and REVOKE privileges to and from other users on that object. DBADM authority is required to grant CONTROL privilege.

Individual privileges and database authorities allow a specific function but do not include the right to grant the same privileges or authorities to other users. The right to grant table, view or schema privileges to others can be extended to other users using the WITH GRANT OPTION on the GRANT statement.

## Table Spaces and Other Storage Structures

Storage structures contain the objects of the database. The basic storage structures managed by the database manager are table spaces. A *table space* is a storage structure containing tables, indexes, large objects, and data defined with a LONG data type. There are two types of table spaces:

**Database Managed Space (DMS) Table Space**
> A table space which has its space managed by the database manager.

**System Managed Space (SMS) Table Space**
> A table space which has its space managed by the operating system.

All table spaces consist of containers. A *container* describes where objects, such as some tables, are stored. For example, a subdirectory in a file system could be a container.

For more information on table spaces and containers, see "CREATE TABLESPACE" on page 681 or the *Administration Guide*.

Data that is read from table space containers is placed in an area of memory called a *buffer pool*. A buffer pool is associated with a table space allowing control over which data shares the same memory areas for data buffering. For more information on buffer pools, see "CREATE BUFFERPOOL" on page 531 or the *Administration Guide*.

A partitioned database allows data to be spread across different database partitions. The partitions included are determined by the *nodegroup* assigned to the table space. A nodegroup is a group of one or more partitions that are defined as part of the database. A table space includes one or more containers for each partition in the nodegroup. A *partitioning map* is associated with each nodegroup. The partitioning map is used by the database manager to determine which partition from the nodegroup will store a given row of data. For more information on nodegroups and data partitioning, see "Data Partitioning Across Multiple Partitions", "CREATE NODEGROUP" on page 617 or the *Administration Guide*.

A table can also include columns that register links to data stored in external files. The mechanism for this is the DATALINK data type. A DATALINK value which is recorded in a regular table points to a file stored in an external file server.

The DB2 Data Links Manager, which is installed on a fileserver, works in conjunction with DB2 to provide the following optional functionality:

- Referential integrity to insure that files currently linked to DB2 are not deleted or renamed.
- Security to insure that only those with suitable SQL privileges on the DATALINK column can read the files linked to that column.
- Coordinated backup and recovery of the file.

The DB2 Data Links Manager product comprises the following facilities:

**Data Links File Manager**
Registers all the files in a particular file server that are linked to DB2.

**Data Links Filesystem Filter**
Filters file system commands to insure that registered files are not deleted or renamed. Optionally also filters commands to insure that proper access authority exists.

For more information on DB2 Data Links Manager refer to *Administration Guide, Design and Implementation*.

## Data Partitioning Across Multiple Partitions

DB2 allows great flexibility in spreading data across multiple partitions (nodes) of a partitioned database. Users can choose how to partition their data by declaring partitioning keys and can determine which and how many partitions their table data can be spread across by selecting the nodegroup and table space in which the data should be stored. In addition, a partitioning map (which can be user-updatable) specifies the mapping of partitioning key

## Data Partitioning Across Multiple Partitions

values to partitions. This makes it possible for flexible workload parallelization across a partitioned database for large tables, while allowing smaller tables to be stored on one or a small number of partitions if the application designer chooses. Each local partition may have local indexes on the data it stores in order to provide high performance local data access.

A partitioned database supports a partitioned storage model, in which the partitioning key is used to partition table data across a set of database partitions. Index data is also partitioned with its corresponding tables, and stored locally at each partition.

Before partitions can be used to store database data, they must be defined to the database manager. Partitions are defined in a file called db2nodes.cfg. See the *Administration Guide* for more details about defining partitions.

The partitioning key for a table in a table space on a partitioned nodegroup is specified in the CREATE TABLE statement (or ALTER TABLE statement). If not specified, a partitioning key for a table is created by default from the first column of the primary key. If no primary key is specified, the default partitioning key is the first column defined in that table that has a data type other than a LONG or LOB data type. Partitioned tables must have at least one column that is neither a LONG nor a LOB data type. A table in a table space on a single-partition nodegroup will only have a partitioning key if it is explicitly specified.

*Hash partitioning* is used to place a row on a partition as follows.

1. A hashing algorithm (partitioning function) is applied to the partitioning key (all the columns), which results in a partitioning map index being generated.
2. This partitioning map index is used as an index into the partitioning map. The partition number at that index in the partitioning map is the partition where the row is stored.
3. Partitioning maps are associated with nodegroups, and tables are created in table spaces which are on nodegroups.

DB2 supports *partial declustering*, which means that the table can be partitioned across a subset of partitions in the system (that is, a nodegroup). Tables do not have to be partitioned across all the partitions in the system.

### Partitioning Maps

Each nodegroup is associated with a *partitioning map*, which is an array of 4 096 partition numbers. The partitioning map index produced by the partitioning function for each row of a table is used as an index into the partitioning map to determine partition on which a row is stored.

## Data Partitioning Across Multiple Partitions

Figure 8 shows how the row with the partitioning key value (c1, c2, c3) is mapped to partitioning map index 2, which, in turn, references partition p5.
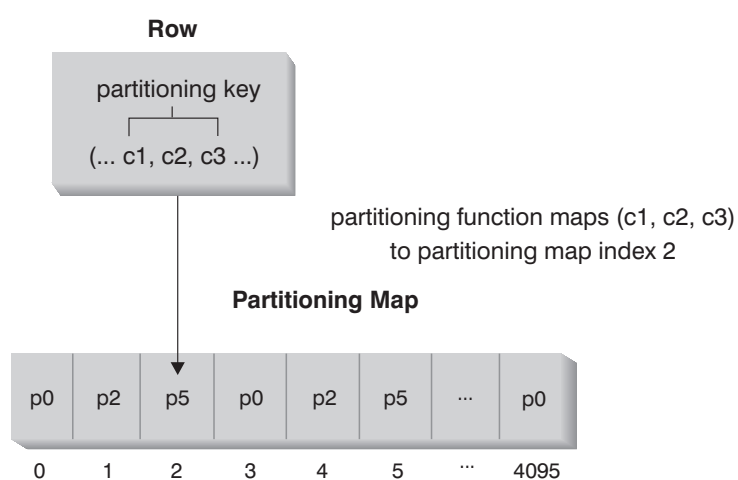


*Figure 8. Data Distribution*

The partitioning map can be changed, allowing the data distribution to be changed without modifying the partitioning key or the actual data. The new partitioning map is specified as part of the REDISTRIBUTE NODEGROUP command or API which uses it to redistribute the tables in the nodegroup. See *Command Reference* or *Administrative API Reference* for further information.

## Table Collocation

DB2 has the capability of recognizing when the data accessed for a join or subquery is located at the same partition in the same nodegroup. When this happens DB2 can choose to perform the join or subquery processing at the partition where the data is stored, which often has significant performance advantages. This situation is called table collocation. To be considered collocated tables, the tables must:

* be in the same nodegroup (that is not being redistributed [12])
* have partitioning keys with the same number of columns

---

12. While redistributing a nodegroup, tables in the nodegroup may be using different partitioning maps - they are not collocated.

## Data Partitioning Across Multiple Partitions

- have the corresponding columns of the partitioning key be partition compatible (see "Partition Compatibility" on page 110).

OR

- be in a single partition nodegroup defined on the same partition.

Rows in collocated tables with the same partitioning key values will be located on the same partition.

# Chapter 3. Language Elements

This chapter defines the basic syntax of SQL and language elements that are common to many SQL statements.

## 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) and 26 lowercase (a through z) letters plus the three characters ($, #, and @), which are included for compatibility with host database products (for example, in code page 850, $ is

## Characters

at X'24' # is at X'23', and @ is at X'40'). Letters also include the alphabetics from the extended character sets. Extended character sets contain additional alphabetic characters; for example, those with diacritical (eg., ´) marks. 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 below:

|   | blank | – | minus sign |
|---|---|---|---|
| " | quotation mark or double-quote | . | period |
| % | percent | / | slash |
| & | ampersand | : | colon |
| ' | apostrophe or single quote | ; | semicolon |
| ( | left parenthesis | < | less than |
| ) | right parenthesis | = | equals |
| * | asterisk | > | greater than |
| + | plus sign | ? | question mark |
| , | comma | _ | underline or underscore |
| \| | vertical bar | ˆ | caret |
| ! | exclamation mark | | |

### MBCS Considerations

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

---

## Tokens

The basic syntactical units of the language are called *tokens*. A token is a sequence of one or more characters. A token cannot contain blank characters, unless it is a string constant or delimited identifier, which may contain blanks. (These terms are defined later.)

Tokens are classified as *ordinary* or *delimiter* tokens:

- 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, as explained under "PREPARE" on page 824.

*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**: Static SQL statements may include host language comments or SQL comments. Either type of comment may be specified wherever a space may be specified, except within a delimiter token or between the keywords EXEC and SQL. SQL comments are introduced by two consecutive hyphens (--) and ended by the end of the line. For more information, see "SQL Comments" on page 434.

**Uppercase and Lowercase**: 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';
```

## MBCS Considerations

Multi-byte alphabetic letters are not folded to uppercase. Single-byte characters, a to z, are folded to uppercase.

## 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 a letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character. An ordinary identifier should not be identical to a reserved word (see "Appendix G. Reserved Schema Names and Reserved Words" on page 1079 for information on reserved words).

## Identifiers

- A *delimited identifier* is a sequence of one or more characters enclosed within quotation marks ("). Two consecutive quotation marks are used to represent one quotation mark within the delimited identifier. In this way an identifier can include lowercase letters.

*Examples of ordinary and delimited identifiers are:*

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

Character conversions between 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 to multi-byte, it is possible that such identifiers may exceed the length limit for an identifier (see "Appendix O. Japanese and Traditional-Chinese EUC Considerations" on page 1181 for details).

### 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 characters and should not begin with upper or lower case spelling of 'SQL' or 'DB2'.

## Naming Conventions and Implicit Object Name Qualifications

The rules for forming a name depend on the type of the object designated by the name. Database object names may be made up of a single identifier or they may be schema qualified objects made up of two identifiers. Schema qualified object names may be specified without the schema name. In such cases, a schema name is implicit.

In dynamic SQL statements, a schema qualified object name implicitly uses the CURRENT SCHEMA special register value as the qualifier for unqualified object name references. By default it is set to the current authorization ID. See "SET SCHEMA" on page 895 for details. If the dynamic SQL statement is from a package bound with the DYNAMICRULES BIND option, the CURRENT SCHEMA special register is not used in qualification. In a DYNAMICRULES BIND package, the package default qualifier is used as the value for qualification of unqualified object references within dynamic SQL statements.

In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified database object names. By default, it is set to the package authorization ID. See the *Command Reference* for details.

The syntax diagrams use different terms for different types of names. The following list defines these terms. For maximum length of various identifiers refer to "Appendix A. SQL Limits" on page 919.

| | |
|---|---|
| **alias-name** | A schema qualified name that designates an alias. |
| **attribute-name** | An identifier that designates an attribute of a structured data type. |
| **authorization-name** | An identifier that designates a user or group. Note the following restrictions on the characters that can be used: |

- Valid characters are A through Z, a through z, 0 through 9, #, @, $ 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 bufferpool. |
| **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. |
| **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. |
| **data-source-name** | A identifier that designates a data source. This identifier is the first of the three parts of remote-object-name. |
| **descriptor-name** | A colon followed by a host identifier that designates an SQL descriptor area (SQLDA). See "References to Host Variables" on page 131 for a description of a host identifier. Note that a descriptor-name never includes an indicator variable. |
| **distinct-type-name** | A qualified or unqualified name that |

|  | designates a distinct type-name. An unqualified distinct-type-name in an SQL statement is implicitly qualified by the database manager, depending on context. |
|---|---|
| **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. A unqualified function-name in an SQL statement is implicitly qualified by the database manager, depending on context. |
| **host-variable** | A sequence of tokens that designates a host variable. A host variable includes at least one host identifier, as explained in "References to Host Variables" on page 131. |
| **index-name** | A schema qualified name that designates an index or an index specification. |
| **nickname** | A schema qualified name that designates a federated server reference to a table or view. |
| **nodegroup-name** | An identifier that designates a nodegroup. |
| **package-name** | A schema qualified name that designates a package. |
| **procedure-name** | A qualified or unqualified name that designates a procedure. An unqualified procedure-name in an SQL statement is implicitly qualified by the database manager, depending on context. |
| **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 where the table or view resides. The parts in this name are data-source-name, remote-schema-name, and remote-table-name. |
| **remote-schema-name** | A name that designates the schema that a data |

|  | source table or view belongs. This name is the second of the three parts of remote-object-name. |
|---|---|
| **remote-table-name** | A name that designates a table or view at a data source. This name is the third of the three parts of remote-object-name. |
| **remote-type-name** | A data type supported by a data source database. Do not use the long form for system built-in types (for example, use CHAR instead of CHARACTER). |
| **schema-name** | An identifier that provides a logical grouping for SQL objects. A schema-name used as a qualifier of the name of an object may be implicitly determined:<br>• from the value of the CURRENT SCHEMA special register<br>• from the value of the QUALIFIER precompile/bind option<br>• based on a resolution algorithm that uses the CURRENT PATH special register<br>• based on the schema name of another object in the same SQL statement. |
| **server-name** | An identifier that designates an application server. In a federated system, server-name also designates the local name of a data source. |
| **specific-name** | A qualified or unqualified name that designates specific-name. A specific-name can be used to source a function, to drop and to comment on a procedure or a function. It can never be used to invoke a function or procedure. An unqualified specific-name in an SQL statement is implicitly qualified by the database manager, depending on context. |
| **statement-name** | An identifier that designates a prepared SQL statement. |
| **supertype-name** | A qualified or unqualified name that designates the supertype of a type-name. An unqualified supertype-name in an SQL statement is implicitly qualified by the database manager, depending on context. |

## Naming Conventions

| | |
|---|---|
| **table-name** | A schema qualified name that designates a table. |
| **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-name. An unqualified type-name in an SQL statement is implicitly qualified by the database manager, 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. |
| **view-name** | A schema qualified name that designates a view. |
| **wrapper-name** | An identifier that designates a wrapper. |

## Aliases

A table alias can be thought of as an alternative name for a table or view. A table or view, therefore, can be referred to in an SQL statement by its name or by a table alias.

An alias can be used wherever a table or view name can be used. An alias can be created even though the object does not exist (though 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 table, view, or alias within the same database. An alias name cannot be used where a new table or view name is expected, such as in the CREATE TABLE or CREATE VIEW statements; for example, if an alias name of PERSONNEL is created then a subsequent statement such as CREATE TABLE PERSONNEL... will cause an error.

The option of referring to a table or view by an alias is not explicitly shown in the syntax diagrams or mentioned in the description of the SQL statement.

A new unqualified alias cannot have the same fully-qualified name as an existing table, view, or alias.

The effect of using an alias in an SQL statement is similar to that of text substitution. The alias, which must be defined when the SQL statement is compiled, is replaced at statement compilation time by the qualified base table or view 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
```

In a federated system, the aforementioned uses and restrictions apply not only to table aliases but also to aliases for nicknames. Thus, a nickname's alias can be used in lieu of the nickname in an SQL statement; an alias can be created for a nickname that does not yet exist, provided that the nickname is created before statements that reference the alias are compiled; an alias for a nickname can refer to another alias for that nickname; and so on.

For syntax toleration of other relational database management system applications, SYNONYM can be used in place of ALIAS in the CREATE ALIAS and DROP ALIAS statements.

Refer to "CREATE ALIAS" on page 528 for more information about aliases.

## 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
- Default value for the QUALIFIER precompile/bind option and the CURRENT SCHEMA special register. Also, the authorization ID is included in the default CURRENT PATH special register and 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 for the package issuing the dynamic SQL statement. For a package bound with DYNAMICRULES RUN, the authorization ID used is the authorization ID of

## Authorization IDs and authorization-names

the user executing the package. For a package bound with DYNAMICRULES BIND, the authorization ID used is the authorization ID of the package. This is called the *run-time authorization ID*.

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 statement. An authorization-name is used in a CREATE SCHEMA statement to designate the owner of the schema. An authorization-name is used in GRANT and REVOKE statements to designate a target of the grant or revoke. Note that the premise of a grant of privileges to *X* is that *X* or a member of the group *X* will subsequently be the authorization ID of statements which require those privileges.

*Examples:*
- Assume SMITH is the userid and the authorization ID that the database manager obtained when the 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. Hence, in a dynamic SQL statement the default value of the CURRENT SCHEMA special register and in static SQL the default QUALIFIER precompile/bind option is SMITH. Thus, 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 64.

  KEENE is an authorization-name specified in the statement. KEENE is given the SELECT privilege on SMITH.TDEPT.
- Assume 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 which 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 and PRECOMPILE command option OWNER defines the authorization ID of the package.

The BIND and PRECOMPILE command option QUALIFIER defines implicit qualifier for unqualified objects contained in the package.

The BIND and PRECOMPILE command option DYNAMICRULES determines whether dynamic SQL statements are processed at run time with run time rules, DYNAMICRULES RUN, or with bind time rules, DYNAMICRULES BIND. These rules indicate the setting of the value used as authorization ID and for the setting of the value used for implicit qualification of unqualified object references within dynamic SQL statements. The DYNAMICRULES options have the effects described in the following tables:

*Table 1. Static SQL Characteristics affected by OWNER and QUALIFIER*

| Feature | Only OWNER Specified | Only QUALIFIER Specified | QUALIFIER and OWNER specified |
| --- | --- | --- | --- |
| Authorization ID Used | ID of User specified in the OWNER option on the BIND command | ID of User Binding the Package | ID of User specified in the OWNER option on the BIND command |
| Unqualified Object Qualification Value Used | ID of User specified in the OWNER option on the BIND command | ID of User specified in the QUALIFIER option on the BIND command | ID of User specified in the QUALIFIER option on the BIND command |

*Table 2. Dynamic SQL Characteristics affected by DYNAMICRULES, OWNER and QUALIFIER*

| Feature | RUN | BIND |
| --- | --- | --- |
| Authorization ID Used | ID of User Executing Package | Authorization ID for the package |
| Unqualified Object Qualification Value Used | CURRENT SCHEMA Special Register | Authorization ID for the package |

Considerations regarding the DYNAMICRULES option:
- The CURRENT SCHEMA special register will not be used to qualify unqualified object references within dynamic SQL statements executed from a package bound with DYNAMICRULES BIND. Instead, DB2 will use the package default qualifier as is shown in the table. This is true even after

you issue the statement SET CURRENT SCHEMA in order to change the
CURRENT SCHEMA special register; the register value will be changed but
not used.

- The following dynamically prepared SQL statements can not be used within
  a package bound with DYNAMICRULES BIND option: GRANT, REVOKE,
  ALTER, CREATE, DROP, COMMENT ON, RENAME, SET INTEGRITY, SET
  EVENT MONITOR STATE and queries that reference a nickname.

- If multiple packages are referenced during a single connection, dynamic
  SQL will behave according to the BIND options for the package in which a
  statement is bound.

- It is important to keep in mind that when binding a package with
  DYNAMICRULES BIND, the binder of the package should not have any
  authorities granted to them that you would not want the user of the
  package to have since a dynamic statement will be using the authorization
  ID of the package owner.

## Data Types

For information about specifying the data types of columns, see "CREATE
TABLE" on page 639.

The smallest unit of data that can be manipulated in SQL is called a *value.*
How values are interpreted depends on the data type of their source. The
sources of values are:

Constants
Columns
Host variables
Functions
Expressions
Special registers.

DB2 supports a number of built-in datatypes, which are described in this
section. It also provides support for user-defined data types. See "User
Defined Types" on page 84 for a description of user-defined data types.

Figure 9 on page 73 illustrates the supported built-in data types.

```
                          ┌──────────┐
                          │ built-in │
                          │  data    │
                          │  types   │
                          └──────────┘
```
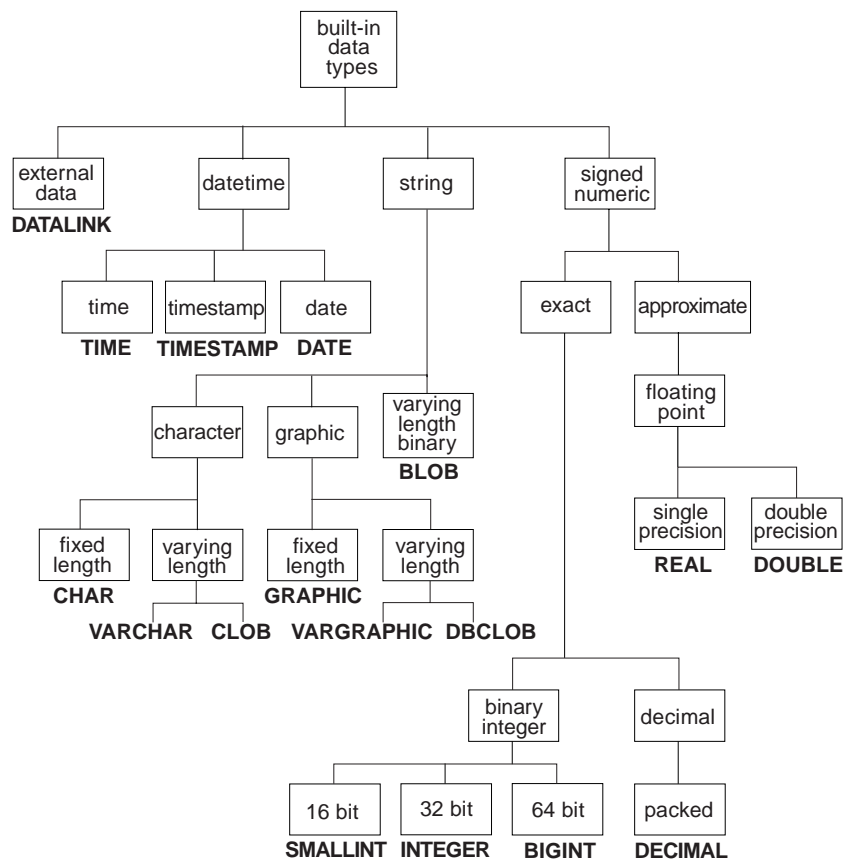
Figure 9. Supported Built-in Data Types

## Nulls

All data types include the null 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, columns defined as NOT NULL cannot contain null values.

## Large Objects (LOBs)

The term *large object* and the generic acronym *LOB* are used to refer to any BLOB, CLOB, or DBCLOB data type. LOB values are subject to the restrictions that apply to LONG VARCHAR values as specified in "Restrictions Using Varying-Length Character Strings" on page 76. For LOB strings, these restrictions apply even when the length attribute of the string is 254 bytes or less.

**Data Types**

### Character Large Object (CLOB) Strings

A *Character Large OBject (CLOB)* is a varying-length string measured in bytes that can be up to 2 gigabytes (2 147 483 647 bytes) long. 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 associated with it). Note that a CLOB is considered to be a character string.

### Double-Byte Character Large Object (DBCLOB) Strings

A *Double-Byte Character Large OBject (DBCLOB)* is a varying-length string of double-byte characters that can be up to 1 073 741 823 characters long. 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 CCSID associated with it). Note that a DBCLOB is considered to be a graphic string.

### Binary Large Objects (BLOBs)

A *Binary Large OBject (BLOB)* is a varying-length string measured in bytes that can be up to 2 gigabytes (2 147 483 647 bytes) long. A *BLOB* is primarily intended to hold non-traditional data such as pictures, voice, and mixed media. Another use is to hold structured data for exploitation by user-defined types and user-defined functions. As with FOR BIT DATA character strings, BLOB strings are not associated with a character set.

### Manipulating Large Objects (LOBs) with Locators

Since LOB values can be very large, the transfer of these values from the database server to client application program host variables can be time consuming. However, it is also true that application programs typically process LOB values a piece at a time, rather than as a whole. For those cases where an application does not need (or want) the entire LOB value to be stored in application memory, the application can reference a LOB value via a large object locator (LOB locator).

A *large object locator* or LOB locator is a host variable with a value that represents a single LOB value in the database server. LOB locators were developed to provide users with a mechanism by which they could easily manipulate very large objects in application programs without requiring them to store the entire LOB value on the client machine where the application program may be running.

For example, when selecting a LOB value, an application program could select the entire LOB value and place it into an equally large host variable (which is acceptable if the application program is going to process the entire LOB value at once), or it could instead select the LOB value into a LOB locator. Then,

using the LOB locator, the application program can issue subsequent database operations on the LOB value (such as applying the scalar functions SUBSTR, CONCAT, VALUE, LENGTH, doing an assignment, searching the LOB with LIKE or POSSTR, or applying UDFs against the LOB) by supplying the locator value as input. The resulting output of the locator operation, for example the amount of data assigned to a client host variable, would then typically be a small subset of the input LOB value.

LOB locators may also 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> )
```

For normal host variables in an application program, when a null value is selected into that 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. Since a locator host variable itself can 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 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 in order 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 above) is deferred until it is actually assigned to some location — either into a user buffer in the form of a host variable or into another record's field value 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. Also, 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, since a 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 that is used by FETCH, OPEN and EXECUTE statements.

**Data Types**

## Character Strings

A *character string* is a sequence of bytes. The length of the string is the number of bytes 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.

### Fixed-Length Character Strings

All values of a fixed-length string column have the same length, which is determined by the length attribute of the column. The length attribute must be between 1 and 254, inclusive.

### Varying-Length Character Strings

Varying-length character strings are of three types: VARCHAR, LONG VARCHAR, and CLOB.
- VARCHAR types are varying-length strings of up to 32 672 bytes.
- LONG VARCHAR types are varying-length strings of up to 32 700 bytes.
- CLOB types are varying-length strings of up to 2 gigabytes.

**Restrictions Using Varying-Length Character Strings:** Special restrictions apply to an expression resulting in a varying-length string data type whose maximum length is greater than 254 bytes; such expressions are not permitted in:
- A SELECT DISTINCT statement's SELECT list
- A GROUP BY clause
- An ORDER BY clause
- A column function with DISTINCT
- A subselect of a set operator other than UNION ALL.

In addition to the restrictions listed above, expressions resulting in LONG VARCHAR or CLOB data types are not permitted in:
- A Basic, Quantified, BETWEEN, or IN predicate
- A column 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 may explictly cast the greater than 4 000 VARCHAR strings into CLOBs and then recast the result back into VARCHARs of desired length.

### NUL-Terminated Character Strings

NUL-terminated character strings found in C are handled differently, depending on the standards level of the precompile option. See the C language specific section in the *Application Development Guide* for more information on the treatment of NUL-terminated character strings.

This data type cannot be created in a table. It can only be used to insert data into and retrieve data from the database.

### Character Subtypes

Each character string is further defined as one of:

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

**SBCS data**   Data in which every character is represented by a single byte.

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

**SBCS and MBCS Considerations:**  SBCS data is supported only in a SBCS database. Mixed data is only supported in an MBCS database.

## Graphic Strings

A *graphic string* is a sequence of bytes which represents double-byte character data. The length of the string is the number of double-byte characters 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 validated to ensure that their values contain only double-byte character code points.[13] Rather, the database manager assumes that double-byte character data is contained within graphic data fields. The database manager checks that a graphic string value is an even number of bytes in length.

A graphic string data type may be fixed length or varying length; the semantics of fixed length and varying length are analogous to those defined for character string data types.

---

13. The exception to this rule is an application precompiled with the WCHARTYPE CONVERT option. In this case, validation does occur. See "Programming in C and C++" in the *Application Development Guide* for details.

### Fixed-Length Graphic Strings

All values of 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 between 1 and 127, inclusive.

### Varying-Length Graphic Strings

Varying-length graphic strings are of three types: VARGRAPHIC, LONG VARGRAPHIC, and DBCLOB.
- VARGRAPHIC types are varying-length strings of up to 16 336 double-byte characters.
- LONG VARGRAPHIC types are varying-length strings of up to 16 350 double-byte characters.
- DBCLOB types are varying-length strings of up to 1 073 741 823 double-byte characters.

Special restrictions apply to an expression resulting in a varying-length graphic string data type whose maximum length is greater than 127. Those restrictions are the same as specified in "Restrictions Using Varying-Length Character Strings" on page 76.

### NUL-Terminated Graphic Strings

NUL-terminated graphic strings found in C are handled differently, depending on the standards level of the precompile option. See the C language specific section in the *Application Development Guide* for more information on the treatment of NUL-terminated graphic strings.

This data type cannot be created in a table. It can only be used to insert data into and retrieve data from the database.

## Binary String

A *binary string* is a sequence of bytes. Unlike a character string which usually contains text data, a binary string is used to hold non-traditional data such as pictures. Note that character strings of the 'bit data' subtype may be used for similar purposes, but the two data types are not compatible. The BLOB scalar function can be used to cast a character for bit string to a binary string. The length of a binary string is the number of bytes. It is not associated with a code page. Binary strings have the same restrictions as character strings (see "Restrictions Using Varying-Length Character Strings" on page 76 for details).

## Numbers

All numbers have a sign and a precision. The *precision* is the number of bits or digits excluding the sign. The sign is considered positive if the value of a number is zero.

### 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.

### 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 -3.402E+38 to -1.175E-37, or from 1.175E-37 to 3.402E+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 -1.79769E+308 to -2.225E-307, or from 2.225E-307 to 1.79769E+308.

### 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. For information on packed decimal representation, see "Packed Decimal Numbers" on page 943.

All values of 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.

**Data Types**

## Datetime Values

The datetime data types are described below. Although datetime values can be used in certain arithmetic and string operations and are compatible with certain strings, they are neither strings nor numbers.

### Date

A *date* is a three-part value (year, month, and day). The range of the year part is 0001 to 9999. The range of the month part is 1 to 12. The range of the day part is 1 to $x$, where $x$ depends on the month.

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 to 24; while the range of the other parts is 0 to 59. If the hour is 24, the minute and second specifications will be zero.

The internal representation of a time is a string of 3 bytes. Each byte is 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 seven-part value (year, month, day, hour, minute, second, and microsecond) that designates a date and time as defined above, except that the time includes a fractional specification of microseconds.

The internal representation of a timestamp is a string of 10 bytes, each of which consists of 2 packed decimal digits. The first 4 bytes represent the date, the next 3 bytes the time, and the last 3 bytes the microseconds.

The length of a TIMESTAMP column, as described in the SQLDA, is 26 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 SQL user. Dates, times, and timestamps can, however, also be represented by character strings, and these representations directly concern the SQL user since there are no constants or variables whose data types are DATE, TIME, or TIMESTAMP. Thus, to be retrieved, a datetime value must be assigned to a character string variable. Note that the CHAR function can be used to change a datetime value to a string representation. The character string representation is normally the default format of datetime values associated with the country code of the database, 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 or LONG VARCHAR cannot be used as the string that represents a datetime value; otherwise an error is raised (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 before the operation is performed. The following sections define the valid string representations of datetime values.

## Date Strings

A string representation of a date is a character string that starts with a digit and has a length of at least 8 characters. Trailing blanks may be included; leading zeros may be omitted from the month and day portions.

Valid string formats for dates are listed in Table 1. Each format is identified by name and includes an associated abbreviation and an example of its use.

*Table 3. Formats for String Representations of Dates*

| Format Name | Abbreviation | Date Format | Example |
|---|---|---|---|
| International Standards Organization | ISO | yyyy-mm-dd | 1991-10-27 |
| IBM USA standard | USA | mm/dd/yyyy | 10/27/1991 |
| IBM European standard | EUR | dd.mm.yyyy | 27.10.1991 |
| Japanese Industrial Standard Christian era | JIS | yyyy-mm-dd | 1991-10-27 |
| Site-defined (see *DB2 Data Links Manager for AIX Quick Beginnings*) | LOC | Depends on database country code | — |

## Data Types

### Time Strings

A string representation of a time is a character string that starts with a digit and has a length of at least 4 characters. Trailing blanks may be included; a leading zero may be omitted from the hour part of the time and seconds may be omitted entirely. If seconds are omitted, an implicit specification of 0 seconds is assumed. Thus, 13.30 is equivalent to 13.30.00.

Valid string formats for times are listed in Table 4. Each format is identified by name and includes an associated abbreviation and an example of its use.

*Table 4. Formats for String Representations of Times*

| Format Name | Abbreviation | Time Format | Example |
|---|---|---|---|
| International Standards Organization[2] | ISO | hh.mm.ss | 13.30.05 |
| IBM USA standard | USA | hh:mm AM or PM | 1:30 PM |
| IBM European standard | EUR | hh.mm.ss | 13.30.05 |
| Japanese Industrial Standard Christian Era | JIS | hh:mm:ss | 13:30:05 |
| Site-defined (see *DB2 Data Links Manager for AIX Quick Beginnings*) | LOC | Depends on database country code | — |

**Notes:**

1. In ISO, EUR and JIS format, .ss (or :ss) is optional.

2. The International Standards Organization recently changed the time format so that it is identical with the Japanese Industrial Standard Christian Era. Therefore, use JIS format if an application requires the current International Standards Organization format.

3. In the case of the USA time string format, the minutes specification may be omitted, indicating an implicit specification of 00 minutes. Thus 1 PM is equivalent to 1:00 PM.

4. In the USA time format, the hour must not be greater than 12 and cannot be 0 except for the special case of 00:00 AM. There is a single space before the AM and PM. Using the ISO format of the 24-hour clock, the correspondence between the USA format and the 24-hour clock is as follows:

   12:01 AM through 12:59 AM corresponds to 00.01.00 through 00.59.00.
   01:00 AM through 11:59 AM corresponds to 01.00.00 through 11.59.00.
   12:00 PM (noon) through 11:59 PM corresponds to 12.00.00 through 23.59.00.
   12:00 AM (midnight) corresponds to 24.00.00 and 00:00 AM (midnight) corresponds to 00.00.00.

### Timestamp Strings

A string representation of a timestamp is a character string that starts with a digit and has a length of at least 16 characters. The complete string representation of a timestamp has the form *yyyy-mm-dd-hh.mm.ss.nnnnnn*. Trailing blanks may be included. Leading zeros may be omitted from the month, day, and hour part of the timestamp, and microseconds may be truncated or entirely omitted. If any trailing zero digits are omitted in the microseconds portion, an implicit specification of 0 is assumed for the missing digits. Thus, 1991-3-2-8.30.00 is equivalent to 1991-03-02-08.30.00.000000.

SQL statements also support the ODBC string representation of a timestamp as an input value only. The ODBC string representation of a timestamp has the form *yyyy-mm-dd  hh:mm:ss.nnnnnn*. See the *CLI Guide and Reference* for more information on ODBC.

### MBCS Considerations

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

## DATALINK Values

A DATALINK value is an encapsulated value that contains a logical reference from the database to a file stored outside the database. The attributes of this encapsulated value are as follows:

**link type**
    The currently supported type of link is a URL (Uniform Resource Locator).

**scheme**
    For URLs this is a value such as HTTP or FILE. The value, no matter what case it is entered in, is stored in the database in upper case. If a value is not specified, FILE is included in the DATALINK value.

**file server name**
    The complete address of the file server. The value, no matter what case it is entered in, is stored in the database in upper case. If a value is not specified, the file server name of the database server is selected as default and included in the DATALINK value.

**file path**
    The identity of the file within the server. The value is case sensitive and therefore it is not converted to upper case when stored in the database.

## Data Types

**access control token**
> When appropriate, the access token is embedded within the file path. It is generated dynamically and is not a permanent part of the DATALINK value that is stored in the database.

**comment**
> Up to 254 bytes of descriptive information. This is intended for application specific uses such as further or alternative identification of the location of the data.

The characters used in a DATALINK value are limited to the set defined for a URL. These characters include the uppercase (A through Z) and lower case (a through z) letters, the digits (0 through 9) and a subset of special characters ($, -, _, @, ., &, +, !, *, ", ', (, ), =, ;, /, #, ?, :, space, and comma).

The first four attributes are collectively known as the *linkage attributes*. It is possible for a DATALINK value to have only a comment attribute and no linkage attributes. Such a value may even be stored in a column but, of course, no file will be linked to such a column.

It should be noted that DATALINKs cannot be exchanged with a DRDA server.

It is important to distinguish between these DATALINK references to files and the LOB file reference variables described in the section entitled "References to BLOB, CLOB, and DBCLOB Host Variables" on page 133. The similarity is that they both contain a representation of a file. However:

- DATALINKs are retained in the database and both the links and the data in the linked files can be considered as a natural extension of data in the database.
- File reference variables exist temporarily on the client and they can be considered as an alternative to a host program buffer.

Built-in scalar functions are provided to build a DATALINK value (DLVALUE) and to extract the encapsulated values from a DATALINK value (DLCOMMENT, DLLINKTYPE, DLURLCOMPLETE, DLURLPATH, DLURLPATHONLY, DLURLSCHEME, DLURLSERVER).

## User Defined Types

### Distinct Types

A *distinct type* is a user-defined data type that shares its internal representation with an existing type (its "source" type), but is considered to be a separate and incompatible type for most operations. For example, one might want to

define a picture type, a text type, and an audio type, all of which have quite different semantics, but which use the built-in data type BLOB for their internal representation.

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

```
CREATE DISTINCT 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 that is not comparable to a BLOB or to any other type. This allows the creation of functions written specifically for AUDIO and assures that these functions will not be applied to any other type (pictures, text, etc.).

Distinct types are identified by qualified identifiers. If the schema name is not used to qualify the distinct type name when used in other than the CREATE DISTINCT TYPE, DROP DISTINCT TYPE, or COMMENT ON DISTINCT TYPE statements, the *SQL path* is searched in sequence for the first schema with a distinct type that matches. The SQL path is described in "CURRENT PATH" on page 118.

Distinct types support strong typing by ensuring that only those functions and operators explicitly defined on a distinct type can be applied to its instances. For this reason, a distinct type does not automatically acquire the functions and operators of its source type, since these may not be meaningful. (For example, the LENGTH function of the AUDIO type might return the length of its object in seconds rather than in bytes.)

Distinct types sourced on LONG VARCHAR, LONG VARGRAPHIC, LOB types, or DATALINK are subject to the same restrictions as their source type.

However, certain functions and operators of the source type can be explicitly specified to apply to the distinct type by defining user-defined functions that are sourced on functions defined on the source type of the distinct type (see "User-defined Type Comparisons" on page 104 for examples). The comparison operators are automatically generated for user-defined distinct types, except those using LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB, or DATALINK as the source type. In addition, functions are generated to support casting from the source type to the distinct type and from the distinct type to the source type.

## Structured Types

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 may be defined as a *subtype* of another

structured type, called its *supertype*. A subtype inherits all the attributes of its supertype and may have additional attributes defined. The set of structured types that are 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.

A structured type may be used as the type of a table or a view. 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.

A structured type cannot be used as the data type of a column of a table or a view. There is also no support for retrieving a structured type into a host variable in an application program.

### Reference (REF) Types

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 view. When a reference type is used, it may 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*.

## 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 (see "Function Resolution" on page 139)
- casting user-defined types (see "Casting Between Data Types" on page 88)
- assigning user-defined types to built-in data types (see "User-defined Type Assignments" on page 98).

Table 5 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 5. Data Type Precedence Table*

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

## Promotion of Data Types

*Table 5. Data Type Precedence Table  (continued)*

| Data Type | Data Type Precedence List (in best-to-worst order) |
|-----------|-----------------------------------------------------|
| DATE | DATE |
| TIME | TIME |
| TIMESTAMP | TIMESTAMP |
| DATALINK | DATALINK |
| udt | udt (same name) |
| REF(T) | REF(S) (provided that S is a supertype of T) |

**Note:**

The lower case types above 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

Shorter and longer form synonyms of the data types listed are considered to be the same as the synonym listed.

## 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 (as defined in "Promotion of Data Types" on page 87) is one example where the promotion of one data type to another data type requires that the value is 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.

Casting between data types can be done explicitly using the CAST specification (see "CAST Specifications" on page 162) but may also occur implicitly during assignments involving a user-defined types (see "User-defined Type Assignments" on page 98). Also, when creating sourced user-defined functions (see "CREATE FUNCTION" on page 551), 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 6 on page 90.

## Casting Between Data Types

The following casts involving distinct types are supported:
- 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* (see "Promotion of Data Types" on page 87)
- 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 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.

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 SQL path is described further in "CURRENT PATH" on page 118.

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* (see "Promotion of Data Types" on page 87).

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. The SQL path is described further in "CURRENT PATH" on page 118.

# Casting Between Data Types

*Table 6. Supported Casts between Built-in Data Types*

| Source Data Type ↓ / Target Data Type → | SMALLINT | INTEGER | BIGINT | DECIMAL | REAL | DOUBLE | CHAR | VARCHAR | LONG VARCHAR | CLOB | GRAPHIC | VARGRAPHIC | LONG VARG | DBCLOB | DATE | TIME | TIMESTAMP | BLOB |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SMALLINT | Y | Y | Y | Y | Y | Y | Y | - | - | - | - | - | - | - | - | - | - | - |
| INTEGER | Y | Y | Y | Y | Y | Y | Y | - | - | - | - | - | - | - | - | - | - | - |
| BIGINT | Y | Y | Y | Y | Y | Y | Y | - | - | - | - | - | - | - | - | - | - | - |
| DECIMAL | Y | Y | Y | Y | Y | Y | Y | - | - | - | - | - | - | - | - | - | - | - |
| REAL | Y | Y | Y | Y | Y | Y | - | - | - | - | - | - | - | - | - | - | - | - |
| DOUBLE | Y | Y | Y | Y | Y | Y | - | - | - | - | - | - | - | - | - | - | - | - |
| CHAR | Y | Y | Y | Y | - | - | Y | Y | Y | Y | - | Y | - | - | Y | Y | Y | Y |
| VARCHAR | Y | Y | Y | Y | - | - | Y | Y | Y | Y | - | Y | - | - | Y | Y | Y | Y |
| LONG VARCHAR | - | - | - | - | - | - | Y | Y | Y | Y | - | - | - | - | - | - | - | Y |
| CLOB | - | - | - | - | - | - | Y | Y | Y | Y | - | - | - | - | - | - | - | Y |
| GRAPHIC | - | - | - | - | - | - | - | - | - | - | Y | Y | Y | Y | - | - | - | Y |
| VARGRAPHIC | - | - | - | - | - | - | - | - | - | - | Y | Y | Y | Y | - | - | - | Y |
| LONG VARG | - | - | - | - | - | - | - | - | - | - | Y | Y | Y | Y | - | - | - | Y |
| DBCLOB | - | - | - | - | - | - | - | - | - | - | Y | Y | Y | Y | - | - | - | Y |
| DATE | - | - | - | - | - | - | Y | Y | - | - | - | - | - | - | Y | - | - | - |
| TIME | - | - | - | - | - | - | Y | Y | - | - | - | - | - | - | - | Y | - | - |
| TIMESTAMP | - | - | - | - | - | - | Y | Y | - | - | - | - | - | - | Y | Y | Y | - |
| BLOB | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | Y |

## Notes

- See the description preceding the table for information on supported casts involving user-defined types and reference types.
- Only a DATALINK type can be cast to a DATALINK type.

## 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.

The 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 (see "Rules for Result Data Types" on page 105). The compatibility matrix is as follows.

*Table 7. Data Type Compatibility for Assignments and Comparisons*

| Operands | Binary Integer | Decimal Number | Floating Point | Character String | Graphic String | Date | Time | Time-stamp | Binary String | UDT |
|---|---|---|---|---|---|---|---|---|---|---|
| Binary Integer | Yes | Yes | Yes | No | No | No | No | No | No | [2] |
| Decimal Number | Yes | Yes | Yes | No | No | No | No | No | No | [2] |
| Floating Point | Yes | Yes | Yes | No | No | No | No | No | No | [2] |
| Character String | No | No | No | Yes | No | [1] | [1] | [1] | No [3] | [2] |
| Graphic String | No | No | No | No | Yes | No | No | No | No | [2] |
| Date | No | No | No | [1] | No | Yes | No | No | No | [2] |
| Time | No | No | No | [1] | No | No | Yes | No | No | [2] |
| Timestamp | No | No | No | [1] | No | No | No | Yes | No | [2] |
| Binary String | No | No | No | No [3] | No | No | No | No | Yes | [2] |
| UDT | [2] | [2] | [2] | [2] | [2] | [2] | [2] | [2] | [2] | Yes |

## Assignments and Comparisons

*Table 7. Data Type Compatibility for Assignments and Comparisons  (continued)*

| Operands | Binary Integer | Decimal Number | Floating Point | Character String | Graphic String | Date | Time | Time- stamp | Binary String | UDT |
|---|---|---|---|---|---|---|---|---|---|---|

**Note:**

[1]
The compatibility of datetime values and character strings is limited to assignment and comparison:

- Datetime values can be assigned to character string columns and to character string variables as explained in "Datetime Assignments" on page 96.

- A valid string representation of a date can be assigned to a date column or compared with a date.
- A valid string representation of a time can be assigned to a time column or compared with a time.
- A valid string representation of a timestamp can be assigned to a timestamp column or compared with a timestamp.

[2]
A user-defined type (UDT) value is only comparable to a value defined with the same UDT. In general, assignments are supported between a distinct type value and its source data type. For additional information see "User-defined Type Assignments" on page 98.

[3]
Note that this means that character strings defined with the FOR BIT DATA attribute are also not compatible with binary strings.

[4]
A DATALINK operand can only be assigned to another DATALINK operand. The DATALINK value can only be assigned to a column if the column is defined with NO LINK CONTROL or the file exists and is not already under file link control.

[5]
For information on assignment and comparison of reference types see "Reference Type Assignments" on page 99 and "Reference Type Comparisons" on page 104.

A 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. (See "References to Host Variables" on page 131 for a discussion of indicator variables.)

### Numeric Assignments

The basic rule for numeric assignments is that the whole part of a decimal or integer number is never truncated. If the scale of the target number is less than the scale of the assigned number the excess digits in the fractional part of a decimal number are truncated.

#### Decimal or Integer to Floating-Point

Floating-point numbers are approximations of real numbers. Hence, when a decimal or integer number is assigned to a floating-point column or variable, the result may not be identical to the original number.

### Floating-Point or Decimal to Integer

When a floating-point or decimal number is assigned to an integer column or variable, the fractional part of the number is lost.

### Decimal to Decimal

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 appended or eliminated, and, in the fractional part of the number, the necessary number of trailing zeros is appended, or the necessary number of trailing digits is eliminated.

### Integer to Decimal

When an integer is assigned to a decimal column or variable, the number is converted first 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, or 11,0 for a large integer, or 19,0 for a big integer.

### Floating-Point to Decimal

When a floating-point number is converted to decimal, 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 less than $0.5*10^{-31}$ 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.

## String Assignments

There are two types of assignments:
- *storage assignment* is when a value is assigned to a column or parameter of a function
- *retrieval assignment* is when a value is assigned to a host variable.

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

### Storage Assignment

The basic rule is that the length of the string assigned to a column or function parameter must not be greater than the length attribute of the column or the function parameter. When the length of the string is greater than the length attribute of the column or the function parameter, the following actions may occur:

## Assignments and Comparisons

- the string is assigned with trailing blanks truncated (from all string types except long strings) to fit the length attribute of the target column or function parameter
- an error is returned (SQLSTATE 22001) when:
  - non-blank characters would be truncated from other than a long string
  - any character (or byte) would be truncated from a long string.

When a string is assigned to a fixed-length column 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 blanks. The pad character is always a blank even for columns defined with the FOR BIT DATA attribute.

### Retrieval Assignment

The length of a string assigned to a host variable may be longer than the length attribute of the host variable. When a string is assigned to a host variable and the length of the string is longer than the length attribute of the variable, the string is truncated on the right by the necessary number of characters (or bytes). When this occurs, a warning is returned (SQLSTATE 01004) and the value 'W' is assigned to the SQLWARN1 field of the SQLCA.

Furthermore, 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.

When a character string is assigned to a fixed-length variable 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 blanks. The pad character is always a blank even for strings defined with the FOR BIT DATA attribute.

Retrieval assignment of C NUL-terminated host variables is handled based on options specified with the PREP or BIND command. See the section on programming in C and C++ in the *Application Development Guide* for details.

### Conversion Rules for String Assignments

A character string or graphic string assigned to a column or host variable 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). [14]

---

14. When acting as a DRDA application server, input host variables are converted to the code page of the application server, even if being assigned, compared or combined with a FOR BIT DATA column. If the SQLDA has been modified to identify the input host variable as FOR BIT DATA, conversion is not performed.

## 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 as any other character with respect to truncation.
- Assignment of a character string to a host variable may 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. Graphic string data types are compatible only with other graphic string data types, and never with numeric, character string, or datetime data types.

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 or host variable), 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

## Assignments and Comparisons

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. See the section on programming in C and C++ in the *Application Development Guide* for details.

### Datetime Assignments

The basic rule for datetime assignments is that a DATE, TIME, or TIMESTAMP value may only be assigned to a column with a matching data type (whether DATE, TIME, or TIMESTAMP) or to a fixed– or varying–length character string variable or string column. The assignment must not be to a LONG VARCHAR, BLOB, or CLOB variable or column.

When a datetime value is assigned to a character 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 a host variable, the following rules apply:
**For a DATE:** If the variable length is less than 10 bytes, an error occurs.
**For a TIME:** If the USA format is used, the length of the variable must not be less than 8; in other formats the length must not be less than 5.

If ISO or JIS formats are used, and if the length of the host variable is less than 8, 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 host variable is less than 19 bytes, an error occurs. If the length is less than 26, but greater than or equal to 19 bytes, trailing digits of the microseconds part of the value are omitted. The SQLWARN1 field of the SQLCA is set to indicate the omission.

For further information on string lengths for datetime values, see "Datetime Values" on page 80.

### DATALINK Assignments

The assignment of a value to a DATALINK column results in the establishment of a link to a file unless the linkage attributes of the value are

empty or the column is defined with NO LINK CONTROL. In cases where a linked value already exists in the column, that file is unlinked. Assigning a null value where a linked value already exists also unlinks the file associated with the old value.

If the application provides the same data location as already exists in the column, the link is retained. There are two reasons that this might be done:

- the comment is being changed
- if the table is placed in Datalink Reconcile Not Possible (DRNP) state, the links in the table can be reinstated by providing linkage attributes identical to the ones in the column.

A DATALINK value may be assigned to a column in any of the following ways:

- The DLVALUE scalar function can be used to create a new DATALINK value and assign it to a column. Unless the value contains only a comment or the URL is exactly the same, the act of assignment will link the file.
- A DATALINK value can be constructed in a CLI parameter using the CLI function SQLBuildDataLink. This value can then be assigned to a column. Unless the value contains only a comment or the URL is exactly the same, the act of assignment will link the file.

When assigning a value to a DATALINK column, the following error conditions return SQLSTATE 428D1:

- Data Location (URL) format is invalid (reason code 21).
- File server is not registered with this database (reason code 22).
- Invalid link type specified (reason code 23).
- Invalid length of comment or URL (reason code 27).

  Note that the size of a URL parameter or function result is the same on both input or output and is bound by the length of the DATALINK column. However, in some cases the URL value returned has an access token attached. In situations where this is possible, the output location must have sufficient storage space for the access token and the length of the DATALINK column. Hence, the actual length of the comment and URL in its fully expanded form, including any default URL scheme or default hostname, provided on input should be restricted to accomodate the output storage space. If the restricted length is exceeded, this error is raised.

When the assignment is also creating a link, the following errors can occur:

- File server not currently available (SQLSTATE 57050).
- File does not exist (SQLSTATE 428D1, reason code 24).
- Referenced file cannot be accessed for linking (reason code 26).
- File already linked to another column (SQLSTATE 428D1, reason code 25).

## Assignments and Comparisons

Note that this error will be raised even if the link is to a different database.

In addition, when the assignment removes an existing link, the following errors can occur:

- File server not currently available (SQLSTATE 57050).
- File with referential integrity control is not in a correct state according to the Data Links File Manager (SQLSTATE 58004).

A DATALINK value may be retrieved from the database in either of the following ways:

- Portions of a DATALINK value can be assigned to host variables by use of scalar functions (such as DLLINKTYPE or DLURLPATH).

Note that usually no attempt is made to access the file server at retrieval time.[15] It is therefore possible that subsequent attempts to access the file server through file system commands might fail.

When retrieving a DATALINK, the registry of file servers at the database server is checked to confirm that the file server is still registered with the database server (SQLSTATE 55022). In addition, a warning may be returned when retrieving a DATALINK value because the table is in reconcile pending or reconcile not possible state (SQLSTATE 01627).

### User-defined Type Assignments

With user-defined types, different rules are applied for assignments to host variables than are used for all other assignments.

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

> A value of a distinct type on the right hand side of an assignment is assignable to a host variable on the left hand side if and only if the source type of this distinct type is assignable to this host variable.

If the target of the assignment is a column, the source data type must be castable to the target data type as described in "Casting Between Data Types" on page 88 for user-defined types.

---

15. It may be necessary to access the file server to determine the prefix name associated with a path. This can be changed at the file server when the mount point of a file system is moved. First access of a file on a server will cause the required values to be retrieved from the file server and cached at the database server for the subsequent retrieval of DATALINK values for that file server. An error is returned if the file server cannot be accessed (SQLSTATE 57050).

## 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 hand 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 hand 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.

## String Comparisons

Character strings are compared according to the collating sequence specified when the database was created, except those with a FOR BIT DATA attribute which are always compared according to their bit values.

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

## Assignments and Comparisons

with single-byte 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 according to the collating sequence specified when the database was created (see the *Administration Guide* for more information on collating sequences specified at database creation time). 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.

Long strings and LOB strings are not supported in any comparison operations that use the basic comparison operators (=, <>, <, >, <=, and >=). They are supported in comparisons using the LIKE predicate and the POSSTR function. See "LIKE Predicate" on page 177 and see "POSSTR" on page 312 for details.

Portions of long strings and LOB strings of up to 4 000 bytes can be compared using the SUBSTR and VARCHAR scalar functions. For example, given the columns:

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

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 code page set of the other string. For details, see "Rules for String Conversions" on page 108.

## Assignments and Comparisons

### Ordering of Results

Results that require sorting are ordered based on the string comparison rules discussed in "String Comparisons" on page 99. 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. This was the default collation table in DB2 Version 2.

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 LONG VARGRAPHIC. LONG VARGRAPHIC and DBCLOB data types are not allowed in a comparison operation.

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 may be compared either with another value of the same data type or with a string representation of that data type. All comparisons are chronological, which means the farther a point in time is from January 1, 0001, the greater the value of that point in time.

Comparisons involving TIME values and string representations of time values always include seconds. If the string representation omits seconds, zero seconds is implied.

Comparisons involving TIMESTAMP values are chronological without regard to representations that might be considered equivalent.

Example:

```
TIMESTAMP('1990-02-23-00.00.00') > '1990-02-22-24.00.00'
```

## Assignments and Comparisons

### User-defined Type Comparisons

Values with a user-defined type can only be compared with values of exactly the same user-defined type. The user-defined type must have been defined using the WITH COMPARISONS clause.

Example:

Given the following YOUTH distinct type and CAMP_DB2_ROSTER table:

```
CREATE DISTINCT TYPE YOUTH AS INTEGER WITH COMPARISONS

CREATE TABLE CAMP_DB2_ROSTER
   ( NAME                 VARCHAR(20),
     ATTENDEE_NUMBER      INTEGER NOT NULL,
     AGE                  YOUTH,
     HIGH_SCHOOL_LEVEL    YOUTH)
```

The following comparison is valid:

```
SELECT * FROM CAMP_DB2_ROSTER
   WHERE AGE > HIGH_SCHOOL_LEVEL
```

The following comparison is not valid:

```
SELECT * FROM CAMP_DB2_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_DB2_ROSTER
   WHERE INTEGER(AGE) > ATTENDEE_NUMBER

SELECT * FROM CAMP_DB2_ROSTER
   WHERE CAST( AGE AS INTEGER) > ATTENDEE_NUMBER

SELECT * FROM CAMP_DB2_ROSTER
   WHERE AGE > YOUTH(ATTENDEE_NUMBER)

SELECT * FROM CAMP_DB2_ROSTER
   WHERE AGE > CAST(ATTENDEE_NUMBER AS YOUTH)
```

### 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 function path. The comparison is performed using the representation type of the reference types. The scope of the reference is not considered in the comparison.

## 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 section explains those rules.

These rules apply to:
- Corresponding columns in fullselects of set operations (UNION, INTERSECT and EXCEPT)
- Result expressions of a CASE expression
- Arguments of the scalar function COALESCE (or VALUE)
- Expression values of the in list of an IN predicate
- Corresponding expressions of a multiple row VALUES clause.

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.

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 characters. 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

Character strings are compatible with other character strings. Character strings include data types CHAR, VARCHAR, LONG 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) |

The code page of the result character string will be derived based on the "Rules for String Conversions" on page 108.

## Rules for Result Data Types

### Graphic Strings

Graphic strings are compatible with other graphic strings. Graphic strings include data types GRAPHIC, VARGRAPHIC, LONG 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) |

The code page of the result graphic string will be derived based on the "Rules for String Conversions" on page 108.

### Binary Large Object (BLOB)

A BLOB is compatible only with another BLOB and the result is a BLOB. The BLOB scalar function should be used to cast from other types if they should be treated as BLOB types (see "BLOB" on page 235). The length of the result BLOB is the largest length of all the data types.

### Numeric

Numeric types are compatible with other numeric types. Numeric types include SMALLINT, INTEGER, BIGINT, DECIMAL, REAL and DOUBLE.

| If one operand is... | And the other operand is... | The data type of the result is... |
| --- | --- | --- |
| SMALLINT | SMALLINT | SMALLINT |
| INTEGER | INTEGER | INTEGER |
| INTEGER | SMALLINT | INTEGER |
| BIGINT | BIGINT | BIGINT |
| BIGINT | INTEGER | BIGINT |
| BIGINT | SMALLINT | BIGINT |
| DECIMAL(w,x) | SMALLINT | DECIMAL(p,x) where $p = x+max(w-x,5)$[1] |
| 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] $s = max(x,z)$ |

| If one operand is... | And the other operand is... | The data type of the result is... |
|---|---|---|
| REAL | REAL | REAL |
| REAL | DECIMAL, BIGINT, INTEGER, or SMALLINT | DOUBLE |
| DOUBLE | any numeric | DOUBLE |

**Note:**

1. Precision cannot exceed 31.

## DATE

A date is compatible with another date, or any CHAR or VARCHAR expression that contains a valid string representation of a date. The data type of the result is DATE.

## TIME

A time is compatible with another time, or any CHAR or VARCHAR expression that contains a valid string representation of a time. The data type of the result is TIME.

## TIMESTAMP

A timestamp is compatible with another timestamp, or any CHAR or VARCHAR expression that contains a valid string representation of a timestamp. The data type of the result is TIMESTAMP.

## DATALINK

A datalink is compatible with another datalink. The data type of the result is DATALINK. The length of the result DATALINK is the largest length of all the data types.

## User-defined Types

### Distinct Types

A user-defined distinct type is compatible only with the same user-defined distinct type. The data type of the result is the user-defined distinct type.

### Reference Types

A reference type is compatible with another reference type provided that their target types have a common supertype. The data type of the result is a

## Rules for Result Data Types

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.

### 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 section 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
- Arguments of the scalar function COALESCE (and VALUE)
- 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. [14]
- Otherwise, the result code page is determined by Table 8. An entry of 'first' in the table means the code page from the first operand is selected and an entry of 'second' means the code page from the second operand is selected.

*Table 8. Selecting the Code Page of the Intermediate Result*

| First Operand | Second Operand | | | | |
| --- | --- | --- | --- | --- | --- |
| | Column Value | Derived Value | Constant | Special Register | Host Variable |
| Column Value | first | first | first | first | first |
| Derived Value | second | first | first | first | first |
| Constant | second | second | first | first | first |
| Special Register | second | second | first | first | first |
| Host Variable | second | second | second | second | first |

An intermediate result is considered to be a derived value operand. An expression that is not a single column value, constant, special register, or host variable is also considered a derived value operand. There is an exception to this if the expression is a CAST specification (or a call to a function that is equivalent). In this case, the *kind* for the first operand is based on the first argument of the CAST specification.

View columns are considered to have the operand type of the object on which they are ultimately based. For example, a view column defined on a table column is considered to be a column value, whereas a view column based on a string expression (for example, A CONCAT B) is considered to be a derived value.

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 (or VALUE) scalar function
- The selected result expression of the CASE expression
- 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
- The code page conversion selection table indicates that conversion is necessary.

## Rules for String Conversions

### Examples

*Example 1:* Given the following:

| 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, since the dominant operand is the column COL_1.

*Example 2:* Using the 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 the code page 850.

## Partition Compatibility

*Partition compatibility* is defined between the base data types of corresponding columns of partitioning keys. Partition compatible data types have the property that two variables, one of each type, with the same value, are mapped to the same partitioning map index by the same partitioning function.

Table 9 on page 111 shows the compatibility of data types in partitions.

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 CHAR.
- Partition compatibility is not affected by columns with NOT NULL or FOR BIT DATA definitions.
- NULL values of compatible data types are treated identically. Different results might be produced for NULL values of non-compatible data types.
- Base datatype of the UDT is used to analyze partition compatibility.
- Decimals of the same value in the partitioning 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.

- CHAR or VARCHAR of different lengths are compatible data types.
- REAL or DOUBLE values that are equal are treated identically even though their precision differs.

*Table 9. Partition Compatibilities*

| Operands | Binary Integer | Decimal Number | Floating Point | Character String | Graphic String | Date | Time | Time-stamp | UDT |
|---|---|---|---|---|---|---|---|---|---|
| Binary Integer | Yes | No | No | No | No | No | No | No | [1] |
| Decimal Number | No | Yes | No | No | No | No | No | No | [1] |
| Floating Point | No | No | Yes | No | No | No | No | No | [1] |
| Character String[3] | No | No | No | Yes[2] | No | No | No | No | [1] |
| Graphic String[3] | No | No | No | No | Yes | No | No | No | [1] |
| Date | No | No | No | No | No | Yes | No | No | [1] |
| Time | No | No | No | No | No | No | Yes | No | [1] |
| Timestamp | No | No | No | No | No | No | No | Yes | [1] |
| UDT | [1] | [1] | [1] | [1] | [1] | [1] | [1] | [1] | [1] |

**Note:**

[1]　A user-defined type (UDT) value is partition compatible with the source type of the UDT or any other UDT with a partition compatible source type.

[2]　The FOR BIT DATA attribute does not affect the partition compatibility.

[3]　Note that data types LONG VARCHAR, LONG VARGRAPHIC, CLOB, DBCLOB, and BLOB are not applicable for partition compatibility since they are not supported in partitioning keys.

## 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 attribute NOT NULL.

A negative zero value in a numeric constant (-0) is the same value as a zero without the sign (0).

# Constants

## 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 a 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 characters 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 in 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
```

## Character String Constants

A *character string constant* specifies a varying-length character string and consists of a sequence of characters that starts and ends with an apostrophe ('). This form of string constant specifies the character string contained between the string delimiters. The length of the character string must not be greater than 32 672 bytes. Two consecutive string delimiters are used to represent one string delimiter within the character string.

*Examples*
```
'12/14/1985'
'32'
'DON''T CHANGE'
```

### Unequal Code Page Considerations

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, of which the result is FOR BIT DATA, the constant value will *not* be converted from its database code page representation when used.

## Hexadecimal Constants

A *hexadecimal constant* specifies a varying-length character string with the code page of the application server.

The format of a hexadecimal string constant is an X followed by a sequence of characters that starts and ends with an apostrophe (single quote). The characters between the apostrophes must be an even number of hexadecimal digits. The number of hexadecimal digits must not exceed 16 336, otherwise an error is raised (SQLSTATE -54002). A hexadecimal digit represents 4 bits. It is specified as a digit or any of the letters A through F (uppercase or lowercase) where A represents the bit pattern '1010', B the bit pattern '1011', etc. If a hexadecimal constant is improperly formatted (e.g. it contains an invalid hexadecimal digit or an odd number of hexadecimal digits), an error is raised (SQLSTATE 42606).

*Examples*
```
X'FFFF'      representing the bit pattern '1111111111111111'

X'4672616E6B' representing the VARCHAR pattern of the ASCII string 'Frank'
```

## Graphic String Constants

A *graphic string constant* specifies a varying-length graphic string and consists of a sequence of double-byte characters that starts and ends with a single-byte

apostrophe (') and is preceded by a single-byte G or N. This form of string constant specifies the graphic string contained between the string delimiters. The length of the graphic string must be an even number of bytes and must not be greater than 16 336 bytes.

Examples:

```
G'double-byte character string'
N'double-byte character string'
```

### MBCS Considerations

The apostrophe must not appear as part of an MBCS character to be considered a delimiter.

## Using Constants with User-defined Types

User-defined types have strong typing. This means that a user-defined type is only compatible with its own type. A constant, however, has a built-in type. Therefore, an operation involving a 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 the constant has been cast to the user-defined type (see "CAST Specifications" on page 162 for information on casting). For example, using the table and distinct type in "User-defined Type Comparisons" on page 104, the following comparisons with the constant 14 are valid:

```
SELECT * FROM CAMP_DB2_ROSTER
    WHERE AGE > CAST(14 AS YOUTH)
```

```
SELECT * FROM CAMP_DB2_ROSTER
    WHERE CAST(AGE AS INTEGER) > 14
```

The following comparison is not valid:

```
SELECT * FROM CAMP_DB2_ROSTER
    WHERE AGE > 14
```

## Special Registers

A *special register* is a storage area that is defined for an application process by the database manager and is used to store information that can be referenced in SQL statements. Special registers are in the database code page.

## CURRENT DATE

The 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.

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.

### Example

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 DEGREE

The CURRENT DEGREE special register specifies the degree of intra-partition
parallelism for the execution of dynamic SQL statements. [16] 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 intra-partition 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 intra-partition
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
intra-partition parallelism using a degree determined by the database
manager.

The actual runtime degree of parallelism will be the lower of:
• Maximum query degree (max_querydegree) configuration parameter
• Application runtime degree
• SQL statement compilation degree

---

16. For static SQL, the DEGREE bind option provides the same control.

## Special Registers

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 intra-partition parallelism.

See the *Administration Guide* for a description of parallelism and a list of restrictions.

The value can be changed by executing the SET CURRENT DEGREE statement (see "SET CURRENT DEGREE" on page 865 for information on this statement).

The initial value of CURRENT DEGREE is determined by the dft_degree database configuration parameter. See the *Administration Guide* for a description of this configuration parameter.

## CURRENT EXPLAIN MODE

The CURRENT EXPLAIN MODE special register holds a VARCHAR(254) value which controls the behaviour of the Explain facility with respect to eligible dynamic SQL statements. This facility generates and inserts Explain information into the Explain tables (for more information see the *Administration Guide*). This information does not include the Explain snapshot.

The possible values are YES, NO, EXPLAIN, RECOMMEND INDEXES and EVALUATE INDEXES. [17]

**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 like YES, however, the dynamic statements are not executed.

**NO**     Disables the Explain facility.

**RECOMMEND INDEXES**
For each dynamic query, a set of indexes is recommended. ADVISE_INDEX table is populated with the set of indexes.

**EVALUATE INDEXES**
Dynamic queries are explained as if the recommended indexes existed. The indexes are picked up from ADVISE_INDEX table.

---

17. 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.

The initial value is NO.

Its value can be changed by the SET CURRENT EXPLAIN MODE statement (see "SET CURRENT EXPLAIN MODE" on page 867 for information on this statement).

The CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values interact when the Explain facility is invoked (see Table 126 on page 1165 for details). The CURRENT EXPLAIN MODE special register also interacts with the EXPLAIN bind option (see Table 127 on page 1166 for details). The values 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 which controls the behavior of the Explain snapshot facility. This facility generates compressed information including access plan information, operator costs, and bind-time statistics (for more information see the *Administration Guide* ).

Only the following statements consider the value of this register: DELETE, INSERT, SELECT, SELECT INTO, UPDATE, VALUES or VALUES INTO.

The possible values are YES, NO, and EXPLAIN. [18]
**YES**     Enables the snapshot facility and takes a snapshot of the internal representation of a dynamic SQL statement as the statement is compiled.
**EXPLAIN**
    Enables the facility like YES, however, the dynamic statements are not executed.
**NO**     Disables the Explain snapshot facility.

The initial value is NO.

---

18. 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.

## Special Registers

Its value can be changed by the SET CURRENT EXPLAIN SNAPSHOT statement (see "SET CURRENT EXPLAIN SNAPSHOT" on page 870).

The CURRENT EXPLAIN SNAPSHOT and CURRENT EXPLAIN MODE special register values interact when the Explain facility is invoked (see Table 126 on page 1165 for details). The CURRENT EXPLAIN SNAPSHOT special register also interacts with the EXPLSNAP bind option (see Table 128 on page 1167 for details).

### 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 NODE

The CURRENT NODE special register specifies an INTEGER value that identifies the coordinator node number (the partition to which an application connects).

CURRENT NODE returns 0 if the database instance is not defined to support partitioning (no db2nodes.cfg file[19]).

The CURRENT NODE can be changed by the CONNECT statement, but only under certain conditions (see "CONNECT (Type 1)" on page 512).

### Example

Set the host variable APPL_NODE (integer) to the number of the partition to which the application is connected.

```
VALUES CURRENT NODE
   INTO :APPL_NODE
```

## CURRENT PATH

The CURRENT PATH special register specifies a VARCHAR(254) value that identifies the *SQL path* to be used to resolve function references and data type references that are used in dynamically prepared SQL statements.[20] CURRENT PATH is also used to resolve stored procedure references in CALL statements. The initial value is the default value specified below. For static SQL, the

---

19. For partitioned databases, the db2nodes.cfg file exists and contains partition (or node) definitions. For details refer to the *Administration Guide*.

20. CURRENT FUNCTION PATH is a synonym for CURRENT PATH.

FUNCPATH bind option provides a SQL path that is used for function and data type resolution (see the *Command Reference* for more information on the FUNCPATH bind option).

The CURRENT PATH special register contains a list of one or more schema-names, where the schema-names are enclosed in double quotes and separated by commas (any quotes within the string are repeated as they are in any delimited identifier).

For example, a SQL path specifying that the database manager is to first look in the FERMAT, then XGRAPHIC, then SYSIBM schemas is returned in the CURRENT PATH special register as:

```
"FERMAT","XGRAPHIC","SYSIBM"
```

The default value is "SYSIBM","SYSFUN",X where X is the value of the USER special register delimited by double quotes.

Its value can be changed by the SET CURRENT FUNCTION PATH statement (see "SET PATH" on page 892). 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. SYSIBM does not take any of the 254 characters if it is implicitly assumed.

The use of the SQL path for function resolution is described in "Functions" on page 138. A data type that is not qualified with a schema name will be implicitly qualified with the schema name that is earliest in the SQL path and contains a data type with the same unqualified name specified. There are exceptions to this rule as described in the following statements: CREATE DISTINCT TYPE, CREATE FUNCTION, COMMENT ON and DROP.

### Example

Using the SYSCAT.VIEWS catalog view, find all views that were created with the exact same setting as the current value of the CURRENT PATH special register.

```
SELECT VIEWNAME, VIEWSCHEMA FROM SYSCAT.VIEWS
  WHERE FUNC_PATH = CURRENT PATH
```

## 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 (see the *Command Reference* for additional information on the QUERYOPT bind option). The possible values range from 0 to 9. For example,

if the query optimization class is set to the minimal class of optimization (0), then the value in the special register is 0. The default value is determined by the dft_queryopt database configuration parameter.

Its value can be changed by the SET CURRENT QUERY OPTIMIZATION statement (see "SET CURRENT QUERY OPTIMIZATION" on page 874).

**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 timestamp duration value with a data type of DECIMAL(20,6). This duration is the maximum duration since a REFRESH TABLE statement has been processed on a REFRESH DEFERRED summary table such that the summary table can be used to optimize the processing of a query. If CURRENT REFRESH AGE has a value of 99 999 999 999 999 (ANY), and QUERY OPTIMIZATION class is 5 or more, REFRESH DEFERRED summary tables are considered to optimize the processing of a dynamic SQL query. A summary table with the REFRESH IMMEDIATE attribute and not in check pending state is assumed to have a refresh age of zero.

Its value can be changed by the SET CURRENT REFRESH AGE statement (see "SET CURRENT REFRESH AGE" on page 877). Summary tables defined with REFRESH DEFERRED are never considered by static embedded SQL queries.

The initial value of CURRENT REFRESH AGE is zero.

## CURRENT SCHEMA

The CURRENT SCHEMA special register specifies a VARCHAR(128) value that identifies the schema name used to qualify unqualified database object references where applicable in dynamically prepared SQL statements. [21]

The initial value of CURRENT SCHEMA is the authorization ID of the current session user.

---

21. For compatibility with DB2 for OS/390, the special register CURRENT SQLID is treated as a synonym for CURRENT SCHEMA.

Its value can be changed by the SET SCHEMA statement (see "SET SCHEMA" on page 895).

The QUALIFIER bind option controls the schema name used to qualify unqualified database object references where applicable for static SQL statements (see *Command Reference* for more information).

**Example**

Set the schema for object qualification to 'D123'.

```
SET CURRENT SCHEMA = 'D123'
```

## CURRENT SERVER

The CURRENT SERVER special register specifies a VARCHAR(18) value that identifies the current application server. The actual name of the application server (not an alias) is contained in the register.

The CURRENT SERVER can be changed by the CONNECT statement, but only under certain conditions (see "CONNECT (Type 1)" on page 512).

**Example**

Set the host variable APPL_SERVE (VARCHAR(18)) to the name of the application server to which the application is connected.

```
VALUES CURRENT SERVER
   INTO :APPL_SERVE
```

## CURRENT TIME

The 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.

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.

**Example**

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 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.

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.

### 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 special register specifies the difference between UTC [22] 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. [23]

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.

---

22. Coordinated Universal Time, formerly known as GMT.

23. The CURRENT TIMEZONE value is determined from C runtime functions. See the *Quick Beginnings* for any installation requirements regarding time zone.

**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 )
```

## USER

The USER special register specifies the run-time authorization ID passed to
the database manager when an application starts on a database. The data type
of the register is VARCHAR(128).

**Example**

Select all notes from the IN_TRAY table that the user placed there himself.

```
SELECT * FROM IN_TRAY
  WHERE SOURCE = USER
```

# 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 a column function, a column name specifies all values of the column
    in the group or intermediate result table to which the function is applied.
    (Groups and intermediate result tables are explained under "Chapter 5.
    Queries" on page 367.) 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.

**Column Names**

## 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 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 126 and "Column Name Qualifiers in Correlated References" on page 128.

## 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 or table function 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, view, nickname, or alias. In the case of a nested table expression or table function, a correlation name is required to identify the result table. 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 below:

```
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 below 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.

A nested table expression or table function will consider *table-reference*s that precede it in the FROM clause as object tables. The *table-reference*s 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.

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.

## Column Names

- 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. See "Chapter 5. Queries" on page 367 for more information on fullselects. 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 another discussion of the evaluation of subqueries, see the descriptions of the WHERE and HAVING clauses in "Chapter 5. Queries" on page 367.)

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 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 (see the *Application Development Guide* for more information on declaring host variables for SQL statements in application programs). 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 in the VALUES INTO clause or the INTO clause of a FETCH or a SELECT 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.

### Host Variables in Dynamic SQL

In dynamic SQL statements, parameter markers are used instead of host variables. A parameter marker is a question mark (?) representing 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 parameter markers:

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

## References to Host Variables

For more information on parameter markers, see "Parameter Markers" in "PREPARE" on page 824.

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

```
►►──:host-identifier─────────────────────────────────────►◄
                    ┌─INDICATOR─┐
                    └───────────┴─:host-identifier─┘
```
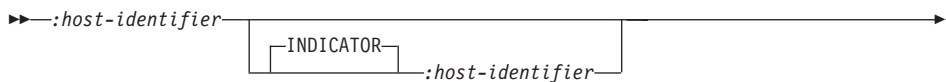
Each *host-identifier* must be declared in the source program. The variable designated by the second host-identifier must have a data type of small integer.

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*. The purposes of the indicator variable are to:

- Specify the null value. A negative value of the indicator variable specifies the null value. A value of -2 indicates a numeric conversion or arithmetic expression error occurred in deriving the result.
- Record the original length of a truncated string (if the source of the value is not a large object type)
- Record the seconds portion of a time if the time is truncated on assignment to a host variable.

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 a VALUES INTO clause or in a FETCH or SELECT INTO statement, and if the value returned is null, HV1 is not changed and HV2 is set to a negative value.[24] If the value returned is not null, that value is assigned to HV1 and HV2 is set to zero (unless the assignment to

---

24. If the database is configured with DFT_SQLMATHWARN yes (or was during binding of a static SQL statement), then HV2 could be -2. If HV2 is -2, then a value for HV1 could not be returned because of an error converting 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 (dec(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.

## References to BLOB, CLOB, and DBCLOB Host Variables

Regular BLOB, CLOB, and DBCLOB variables, LOB locator variables (see "References to Locator Variables" on page 134), and LOB file reference variables (see "References to BLOB, CLOB, and DBCLOB File Reference Variables" on page 135) 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

---

the numeric type of HV1 or an error evaluating an arithmetic expression that is used to determine the value for HV1. When accessing a database with a client version earlier than DB2 Universal Database Version 5, HV2 will be -1 for arithmetic exceptions.

## References to Host Variables

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 and/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.

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.

### References to Locator Variables

A *locator variable* is a host variable that contains the locator representing a LOB value on the application server. (See "Manipulating Large Objects (LOBs) with Locators" on page 74 for information on how locators can be used to manipulate LOB values.)

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*.

When the indicator variable associated with a locator is null, the value of the referenced LOB is null.

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 BLOB, CLOB, and DBCLOB 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 -see below). 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

## References to Host Variables

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** [25]
  This is a regular file that can be opened, read and closed.

- Output (from server to client)

  **SQL_FILE_CREATE** [26]
  Create a new file. If the file already exists, it is an error.

  **SQL_FILE_OVERWRITE (Overwrite)** [27]
  If an existing file with the specified name exists, it is overwritten; otherwise a new file is created.

  **SQL_FILE_APPEND** [28]
  If an existing file with the specified name exists, the output is appended to it; otherwise a new file is created.

  **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.

---

25. SQL-FILE-READ in COBOL, sql_file_read in FORTRAN, READ in REXX.

26. SQL-FILE-CREATE in COBOL, sql_file_create in FORTRAN, CREATE in REXX.

27. SQL-FILE-OVERWRITE in COBOL, sql_file_overwrite in FORTRAN, OVERWRITE in REXX.

28. SQL-FILE-APPEND in COBOL, sql_file_append in FORTRAN, APPEND in REXX.

**Example of an Output File Reference Variable (in C)**

Given a declare section is 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 above, 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);
```

## Functions

A *database function* is a relationship between a set of input data values and a set of result values. For example, the TIMESTAMP function can be passed input data values of type DATE and TIME and the result is a TIMESTAMP. Functions can either be built-in or user-defined.

- *Built-in functions* are provided with the database manager providing a single result value and are identified as part of the SYSIBM schema. Examples of such functions include column functions such as AVG, operator functions such as "+", casting functions such as DECIMAL, and others such as SUBSTR.

- *User-defined functions* are functions that are registered to a database in SYSCAT.FUNCTIONS (using the CREATE FUNCTION statement). User-defined functions are never part of the SYSIBM schema. One such set of functions is provided with the database manager in a schema called SYSFUN.

With user-defined functions, DB2 allows users and application developers to extend the function of the database system by adding function definitions provided by users or third party vendors to be applied in the database engine itself. This allows higher performance than retrieving rows from the database and applying those functions on the retrieved data to further qualify or to perform data reduction. Extending database functions also lets the database exploit the same functions in the engine that an application uses, provides more synergy between application and database, and contributes to higher productivity for application developers because it is more object-oriented.

A complete list of functions in the SYSIBM and SYSFUN schemas is documented in Table 15 on page 188.

A user-defined function can be *external* or *sourced*. 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 can not be column 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 column functions. They are very useful for supporting the use of existing functions with user-defined types.

Each user-defined function is also categorized as a *scalar*, *column* or *table* function. A *scalar function* is one which 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. A *column function* is one which conceptually is passed a set of like values (a column) and returns a single-valued answer from this set. These are also sometimes called

*aggregating functions* in DB2. An example of a column function is the built-in function AVG(). An external column UDF cannot be defined to DB2, but a column UDF which is sourced upon one of the built-in column 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 a column function. A *table function* is a function which returns a table to the SQL statement which references it, and it may only be referenced in the FROM clause of a SELECT. Such a function can be used to apply SQL language processing power to data which is not DB2 data, or to convert such data into a DB2 table. It could, for example, take a file and convert it to a table, sample data from the world-wide web and tabularize it, or access a Lotus Notes database and return information about mail messages, such as the date, sender, and the text of the message. This information can be joined with other tables in the database. A table function can only be an external function (a table function cannot be a sourced function).

A function is identified by its schema, a function name, the number of parameters and the data types of its parameters. This is called a *function signature* which must be unique within the database. There can be more than one function with the same name in a schema 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 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 in the schema (which of necessity have different parameter types). A function name can also be overloaded in a SQL path, in which case there is more than one function by that name in the path, and these functions do not necessarily have different parameter types.

A function can be invoked by referring in an allowable context to the qualified name (schema and function name) followed by the list of arguments enclosed in parentheses. A function can also be invoked without the schema name resulting in a choice of possible functions in different schemas with the same or acceptable parameters. In this case, the *SQL path* is used to assist in *function resolution*. The function path is a list of schemas that are searched to identify a function with the same name, number of parameters and acceptable data types. For static SQL statements, SQL path is specified using the FUNCPATH bind option (see *Command Reference* for details). For dynamic SQL statements, SQL path is the value of the CURRENT FUNCTION PATH special register (see "CURRENT PATH" on page 118).

## Function Resolution

Given a function invocation, the database manager must decide which of the possible functions with the same name is the "best" fit. This includes resolving functions from the built-in and user-defined functions.

## Functions

An *argument* is a value passed to a function upon invocation. 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. When a function is defined to the database, either internally (the built-in functions) or by a user (user-defined functions), its parameters (zero or more) are specified, the order of their definitions defining their positions and thus their semantics. Therefore, every parameter is a particular positional input of a function. 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 function given in the invocation, the number and data types of the arguments, all the functions with the same name in the SQL path, and the data types of their corresponding parameters as the basis for deciding whether or not to select a function. The following are the possible outcomes of the decision process:

1. A particular function is deemed to be the best fit. For example, given the functions named RISK in the schema TEST with signatures defined as:

   ```
   TEST.RISK(INTEGER)
   TEST.RISK(DOUBLE)
   ```

   a SQL path including the TEST schema and the following function reference (where DB is a DOUBLE column):

   ```
   SELECT ... RISK(DB) ...
   ```

   then, the second RISK will be chosen.

   The following function reference (where SI is a SMALLINT column):

   ```
   SELECT ... RISK(SI) ...
   ```

   would choose the first RISK, since SMALLINT can be promoted to INTEGER and is a better match than DOUBLE which is further down the precedence list (as shown in Table 5 on page 87).

2. No function 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 ... RISK(C) ...
   ```

   the argument is inconsistent with the parameter of both RISK functions.

3. A particular function is selected based on the SQL path and the number and data types of the arguments passed on invocation. For example, given functions named RANDOM with signatures defined as:

   ```
   TEST.RANDOM(INTEGER)
   PROD.RANDOM(INTEGER)
   ```

and a SQL path of:

```
"TEST","PROD"
```

Then the following function reference:

```
SELECT ... RANDOM(432) ...
```

will choose TEST.RANDOM since both RANDOM functions are equally good matches (exact matches in this particular case) and both schemas are in the path, but TEST precedes PROD in the SQL path.

## Method of Choosing the Best Fit

A comparison of the data types of the arguments with the defined data types of the parameters of the functions under consideration forms the basis for the decision of which function in a group of like-named functions is the "best fit". Note that the data type of the result of the function or the type of function (column, scalar, or table) under consideration **does not** enter into this determination.

Function resolution is done using the steps that follow.

1. First, find all functions from the catalog (SYSCAT.FUNCTIONS) and built-in functions such that all of the following are true:

   a. For invocations where the schema name was specified (i.e. qualified references), then the schema name and the function name match the invocation name.

   b. For invocations where the schema name was not specified (i.e. unqualified references), then the function name matches the invocation name and has a schema name that matches one of the schemas in the SQL path.

   c. The number of defined parameters matches the invocation.

   d. Each invocation argument matches the function's corresponding defined parameter in data type, or is "promotable" to it (see "Promotion of Data Types" on page 87).

2. Next, consider each argument of the function invocation, from left to right. 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 in Table 5 on page 87 for which there exists a function with a parameter of that data type. Lengths, precisions, scales 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, and a VARCHAR(19) argument is an exact match for a VARCHAR(6) parameter.

## Functions

3. If more than one candidate function remains after Step 2, then it has to be the case (the way the algorithm works) that all the remaining candidate functions have identical signatures but are in different schemas. Choose the function whose schema is earliest in the user's SQL path.

4. If there are no candidate functions remaining after step 2, an error is returned (SQLSTATE 42884).

### Conservative Binding Semantics

There are situations within a database where the functions and data types are resolved when the statement is processed and the database manager must be able to repeat this resolution. This is true of references to functions and data types in:

- static DML statements in packages,
- views,
- triggers, and
- check constraints.

For static DML statements in packages, the function and data type references are resolved during the bind operation. Function and data type references in views, triggers, and check constraints are resolved when the object is created.

If function resolution is performed again on any function references in these objects, it could change the behaviour if a new function has been added with a signature that is a better match but the actual executable perfoms different operations. Similarly, if resolution is performed again on any data type in these objects, it could change the behaviour if a new data type has been added with the same name in a different schema that is also on the SQL path. In order to avoid this, where necessary, the database manager applies the concept of *conservative binding semantics*. This concept ensures that the function and data type references will be resolved using the same SQL path as when it was bound and that the creation timestamp of functions[29] and data types considered during resolution is not later than the time when the statement was bound[30] . In this way, only the functions and data types that were considered during function and data type resolution when the statement was originally processed will be considered. Hence, newly created functions and data types are not considered when conservative binding semantics are applied.

---

29. Built-in functions added starting with Version 6.1 have a creation timestamp that is based on the time of database creation or migration.

30. For views, conservative binding also ensures that the columns of the view are the same as those that existed at the time the view was created. For example, a view defined using the asterisk in the select list will not consider any columns added to the underlying tables after the view was created.

For static DML in packages, the packages can rebind either implicitly or by explicitly issuing the REBIND command (or corresponding API) or the BIND command (or corresponding API). The implicit rebind is always performed to resolve functions with conservative binding semantics. The REBIND command provides the choice to resolve with conservative binding semantics (RESOLVE CONSERVATIVE option) or to resolve considering any new functions (by default or using the RESOLVE ANY option).

## Function Path Considerations for Built-in Functions

Built-in functions reside in a special schema called SYSIBM. Additional functions are available in the SYSFUN schema which are not considered built-in functions since they are developed as user-defined functions and have no special processing considerations. Users can not define additional functions in SYSIBM or SYSFUN schemas (or in any schema whose name begins with the letters "SYS").

As already stated, the built-in functions participate in the function resolution process exactly as do the user-defined functions. One difference between built-in and user-defined functions, from a function resolution perspective, is that the built-in function must always be considered by function resolution. Therefore, omission of SYSIBM from the path results in an 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 since SYSIBM is implicitly first in the path because it was not specified.

It is possible to minimize potential problems in this area by:
- never using the names of built-in functions for user-defined functions, or
- qualifying any references to these functions, if for some reason it is deemed necessary to create a user-defined function with the same name as a built-in function.

**Example of Function Resolution**

The following is an example of successful function resolution.

There are six FOO functions, in two different schemas, registered as (note that not all required keywords appear):

```
CREATE FUNCTION AUGUSTUS.FOO (CHAR(5), INT, DOUBLE)      SPECIFIC FOO_1 ...
CREATE FUNCTION AUGUSTUS.FOO (INT, INT, DOUBLE)          SPECIFIC FOO_2 ...
CREATE FUNCTION AUGUSTUS.FOO (INT, INT, DOUBLE, INT)     SPECIFIC FOO_3 ...
CREATE FUNCTION JULIUS.FOO   (INT, DOUBLE, DOUBLE)       SPECIFIC FOO_4 ...
CREATE FUNCTION JULIUS.FOO   (INT, INT, DOUBLE)          SPECIFIC FOO_5 ...
CREATE FUNCTION JULIUS.FOO   (SMALLINT, INT, DOUBLE)     SPECIFIC FOO_6 ...
CREATE FUNCTION NERO.FOO     (INT, INT, DEC(7,2))        SPECIFIC FOO_7 ...
```

The function reference is as follows (where I1 and I2 are INTEGER columns, and D is a DECIMAL column):

```
SELECT ... FOO(I1, I2, D) ...
```

Assume that the application making this reference has a SQL path established as:

```
"JULIUS","AUGUSTUS","CAESAR"
```

Following through the algorithm...

FOO_7 is eliminated as a candidate, because the schema "NERO" is not included in the SQL path.

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 cannot be promoted to the data type of the first parameter. Because there is more than one candidate remaining, the arguments are then considered in order.

For the first argument, all remaining functions — FOO_2, FOO_4 and FOO_5 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, FOO_2 and FOO_5 are exact matches while FOO_4 is not, so it is eliminated from consideration. The next argument is examined to determine some differentiation between FOO_2 and FOO_5. On the third and last argument, neither FOO_2 nor FOO_5 match the argument type exactly, but both are equally good.

There are two functions remaining, FOO_2 and FOO_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 FOO_5 is finally chosen.

## Function Invocation

Once the function is selected, there are still possible reasons why the use of the function may not be permitted. Each function 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 function is invoked, an error will occur. For example, given functions named STEP defined, this time, with different data types as the result:

```
STEP(SMALLINT) returns CHAR(5)
STEP(DOUBLE)   returns INTEGER
```

and the following function reference (where S is a SMALLINT column):

```
SELECT ... 3 + 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.

A couple of other examples where this can happen are as follows, both of which will result in an error on the statement:

1. The function was referenced in a FROM clause, but the function selected by the function resolution step was a scalar or column function.

2. The reverse case, where the context calls for a scalar or column function, and function resolution selects a table function.

In cases where the arguments of the function invocation were not an exact 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 using the same rules as assignment to columns (see "Assignments and Comparisons" on page 91). This includes the case where precision, scale, or length differs between the argument and the parameter.

## Expressions
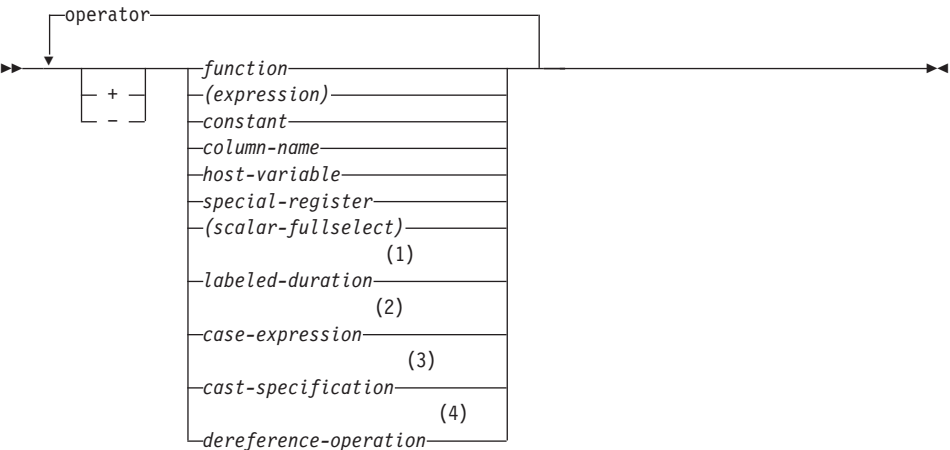


**operator:**



**Notes:**

1   See "Labeled Durations" on page 153 for more information.

2   See "CASE Expressions" on page 160 for more information.

3   See "CAST Specifications" on page 162 for more information.

4   See "Dereference Operations" on page 165 for more information.

5   | | may be used as a synonym for CONCAT.

An *expression* specifies a value.

A *scalar fullselect* as supported in an expression 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. See "fullselect" on page 407 for more information.

## Without Operators

If no operators are used, the result of the expression is the specified value.

Examples: `SALARY` `:SALARY` `'SALARY'` `MAX(SALARY)`

## With the Concatenation Operator

The concatenation operator (CONCAT) links two string operands to form a *string expression*.

The operands of concatenation must be compatible strings. Note that a binary string cannot be concatenated with a character string, including character strings defined as FOR BIT DATA (SQLSTATE 42884). For more information on compatibility, refer to the compatibility matrix on page Table 7 on page 91.

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.

The data type and length attribute of the result is determined from that of the operands as shown in the following table:

*Table 10. Data Type and Length of Concatenated Operands*

| Operands | Combined Length Attributes | Result |
|---|---|---|
| CHAR(A) CHAR(B) | <255 | CHAR(A+B) |
| CHAR(A) CHAR(B) | >254 | 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 |
|  |  |  |

*Table 10. Data Type and Length of Concatenated Operands  (continued)*

| Operands | Combined Length Attributes | Result |
|---|---|---|
| CLOB(A) CHAR(B) | - | CLOB(MIN(A+B, 2G)) |
| CLOB(A) VARCHAR(B) | - | CLOB(MIN(A+B, 2G)) |
| CLOB(A) LONG VARCHAR | - | CLOB(MIN(A+32K, 2G)) |
| CLOB(A) CLOB(B) | - | CLOB(MIN(A+B, 2G)) |
|  |  |  |
| 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 |
| VARGRAPHIC(A) LONG VARGRAPHIC | - | LONG VARGRAPHIC |
|  |  |  |
| LONG VARGRAPHIC LONG VARGRAPHIC | - | LONG VARGRAPHIC |
|  |  |  |
| DBCLOB(A) GRAPHIC(B) | - | DBCLOB(MIN(A+B, 1G)) |
| DBCLOB(A) VARGRAPHIC(B) | - | DBCLOB(MIN(A+B, 1G)) |
| DBCLOB(A) LONG VARGRAPHIC | - | DBCLOB(MIN(A+16K, 1G)) |
| DBCLOB(A) DBCLOB(B) | - | DBCLOB(MIN(A+B, 1G)) |
|  |  |  |
| BLOB(A) BLOB(B) | - | BLOB(MIN(A+B, 2G)) |

Note that, for compatibility with previous versions, there is no automatic escalation of results involving LONG 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 as explained in "Rules for String Conversions" on page 108.

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:

```
COLA defined as VARCHAR(5) with value 'AA'
:host_var defined as a character host variable with length 5 and value
'BB   '
COLC defined as CHAR(5) with value 'CC'
COLD defined as CHAR(5) with value 'DDDDD'
```

The value of: COLA **CONCAT** `:host_var` **CONCAT** COLC **CONCAT** COLD is:

```
        'AABB   CC   DDDDD'
```

The data type is VARCHAR, the length attribute is 17 and the result code page is the database code page.

*Example 3:* Given:

```
COLA is defined as CHAR(10)
COLB is defined as VARCHAR(5)
```

The parameter marker in the expression:

```
COLA CONCAT COLB CONCAT ?
```

is considered VARCHAR(15) since `COLA CONCAT COLB` is evaluated first giving a result which is the first operand of the second CONCAT operation.

**User-defined Types**

A user-defined type cannot be used with the concatenation operator even if its source data type is character. 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, then the following user-defined function, ATTACH, could be used to concatenate them.

```
CREATE FUNCTION ATTACH (TITLE, TITLE_DESCRIPTION)
  RETURNS VARCHAR(50) SOURCE CONCAT (VARCHAR(), VARCHAR())
```

Alternately, the concatenation operator could 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())
```

## With Arithmetic Operators

If arithmetic operators are used, the result of the expression is a value derived from the application of the operators to the values of the operands.

If any operand can be null, or the database is configured with DFT_SQLMATHWARN set to yes, the result can be null.

If any operand has the null value, the result of the expression is the null value.

Arithmetic operators must not be applied to character strings. For example, USER+2 is invalid.

The prefix operator + (unary plus) does not change its operand. The prefix operator − (unary minus) reverses the sign of a nonzero operand; and if the data type of A is small integer, then the data type of −A is large integer. The first character of the token following a prefix operator must not be a plus or minus sign.

The *infix operators* +, −, *, and / specify addition, subtraction, multiplication, and division, respectively. The value of the second operand of division must not be zero. These operators can also be treated as functions. Thus, the expression "+"(a,b) is equivalent to the expression *a+b*. "operator" function.

### Arithmetic Errors

If an arithmetic error such as zero divide or a numeric overflow occurs during the processing of an expression, an error is returned and the SQL statement processing the expression fails with an error (SQLSTATE 22003 or 22012).

A database can be configured (using DFT_SQLMATHWARN set to yes) so that arithmetic errors return a null value for the expression, issue a warning (SQLSTATE 01519 or 01564), and proceed with processing of the SQL statement. 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 a column function causes the row to be ignored in the determining the result of the column 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. Some examples are:

- add a case expression to check for zero divide and set the desired 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 unary minus) 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 which 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 must not have a precision greater than 31. The result of decimal addition, subtraction, and multiplication is derived from

a temporary result which may 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

The following formulas define the precision and scale of the result of decimal operations in SQL. The symbols $p$ and $s$ denote the precision and scale of the first operand, and the symbols $p'$ and $s'$ denote the precision and scale of the second operand.

### Addition and Subtraction

The precision is $\min(31,\max(p\text{-}s,p'\text{-}s')+\max(s,s')+1)$. The scale of the result of addition and subtraction is $\max(s,s')$.

### Multiplication

The precision of the result of multiplication is $\min(31,p+p')$ and the scale is $\min(31,s+s')$.

### Division

The precision of the result of division is 31. The scale is $31\text{-}p+s\text{-}s'$. The scale must not be negative.

## Floating-Point Operands

If either operand of an arithmetic operator is floating-point, the operation is performed in floating-point, the operands having first been 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.

## User-defined Types as Operands

A 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

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)
```

## Datetime Operations and Durations

Datetime values can be incremented, decremented, and subtracted. These operations may involve decimal numbers called *durations*. Following is a definition of durations and a specification of the rules for datetime arithmetic.

A duration is a number representing an interval of time. There are four types of durations:

### Labeled Durations

**labeled-duration:**

```
├──┬─function────┬──┬─YEAR─────────┬──────────────────────────────────┤
   ├─(expression)─┤  ├─YEARS────────┤
   ├─constant─────┤  ├─MONTH────────┤
   ├─column-name──┤  ├─MONTHS───────┤
   └─host-variable┘  ├─DAY──────────┤
                     ├─DAYS─────────┤
                     ├─HOUR─────────┤
                     ├─HOURS────────┤
                     ├─MINUTE───────┤
                     ├─MINUTES──────┤
                     ├─SECOND───────┤
                     ├─SECONDS──────┤
                     ├─MICROSECOND──┤
                     └─MICROSECONDS─┘
```

A *labeled duration* represents 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[31] . The number specified is converted as if it were assigned to a DECIMAL(15,0) number. A labeled duration can only be

---

[31]. Note that the singular form of these keywords is also acceptable: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and MICROSECOND.

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 *date duration* represents 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. [32] The result of subtracting one date value from another, as in the expression HIREDATE – BRTHDATE, is a date duration.

### Time Duration

A *time duration* represents 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. [32] The result of subtracting one time value from another is a time duration.

### Timestamp duration

A *timestamp duration* represents a number of years, months, days, hours, minutes, seconds, and microseconds, expressed as a DECIMAL(20,6) number. To be properly interpreted, the number must have the format *yyyymmddhhmmss.zzzzzz*, where *yyyy*, *mm*, *dd*, *hh*, *mm*, *ss*, and *zzzzzz* represent, respectively, the number of years, months, days, hours, minutes, seconds, and microseconds. The result of subtracting one timestamp value from another is a timestamp duration.

## 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.

---

32. The period in the format indicates a DECIMAL data type.

- 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 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.
- If the first operand is a timestamp, the second operand must be a timestamp, a string representation of a timestamp, or a duration.
- If the second operand is a timestamp, the first operand must be a timestamp or a string representation of a timestamp.
- Neither operand of the subtraction operator can be a parameter marker.

### Date Arithmetic

Dates can be subtracted, incremented, or decremented.

**Subtracting Dates:**  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).

**Incrementing and Decrementing Dates:**  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.

**Subtracting Times:**  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).

**Incrementing and Decrementing Times:**  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
```

**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 (e.g. time('24:00:00')±0 seconds = '00:00:00').

**Timestamp Arithmetic**

Timestamps can be subtracted, incremented, or decremented.

**Subtracting Timestamps:** 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 microseconds between the two timestamps. The data type of the result is DECIMAL(20,6).

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 MICROSECOND(TS2) <= MICROSECOND(TS1)
    then MICROSECOND(RESULT) = MICROSECOND(TS1) −
        MICROSECOND(TS2).

If MICROSECOND(TS2) > MICROSECOND(TS1)
    then MICROSECOND(RESULT) = 1000000 +
        MICROSECOND(TS1) − MICROSECOND(TS2)
            and SECOND(TS2) is incremented by 1.
```

The seconds and 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.

**Incrementing and Decrementing Timestamps:** The result of adding a duration to a timestamp, or of subtracting a duration from a timestamp is itself a timestamp. Date and time arithmetic 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. Microseconds overflow into seconds.

## Precedence of Operations

Expressions within parentheses and dereference operations are evaluated first from left to right.[33] When the order of evaluation is not specified by parentheses, prefix operators are applied before multiplication and division, and multiplication and division are applied before addition and subtraction. Operators at the same precedence level are applied from left to right.
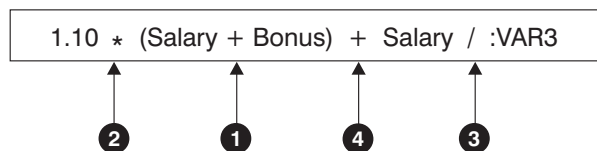


*Figure 10.*

---

33. Note that parentheses are also used in subselect statements, search conditions, and functions. However, they should not be used to arbitrarily group sections within SQL statements.

## CASE Expressions

**case-expression:**

```
┌─ELSE NULL───────────────────┐
├──CASE──┬─searched-when-clause─┬──┤                          ├──END──────────────────────┤
         └─simple-when-clause───┘  └─ELSE─result-expression──┘
```

**searched-when-clause:**

```
      ┌──────────────────────────────────────────────┐
      ▼                                              │
├────────WHEN──search-condition──THEN──┬─result-expression─┬────────────────────┤
                                       └─NULL──────────────┘
```

**simple-when-clause:**

```
               ┌──────────────────────────────────────────┐
               ▼                                          │
├──expression────WHEN──expression──THEN──┬─result-expression─┬────────────────────┤
                                         └─NULL──────────────┘
```
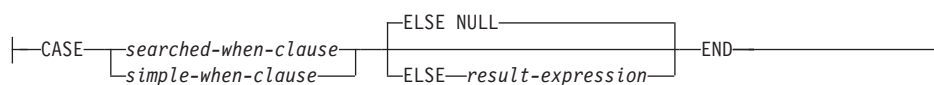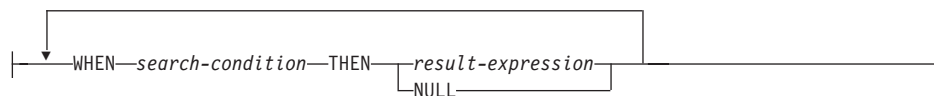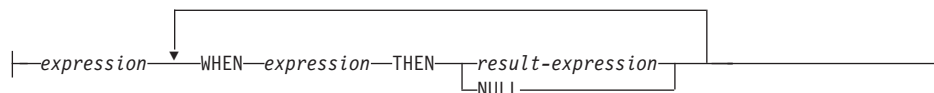
CASE expressions allow an expression to be selected based on the evaluation of one or more conditions. In general, the value of the case-expression is the value of the *result-expression* following the first (leftmost) case that evaluates to true. If no case evaluates to true and the ELSE keyword is 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* prior to the first *WHEN* keyword is tested for equality with the value of the *expression* following the WHEN keyword. The data type of the *expression* prior to the first *WHEN* keyword must therefore be comparable to the data types of each *expression* following the WHEN keyword(s). The *expression* prior to the first *WHEN* keyword in a *simple-when-clause* cannot include a function that is variant 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-expression*s must have compatible data types (SQLSTATE 42804), where the attributes of the result are determined based on the "Rules for Result Data Types" on page 105.

**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'
 ...
```

## Expressions

There are two scalar functions, NULLIF and COALESCE, that are specialized to handle a subset of the functionality provided by CASE. Table 11 shows the equivalent expressions using CASE or these functions.

*Table 11. 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) |

## CAST Specifications

**cast-specification:**
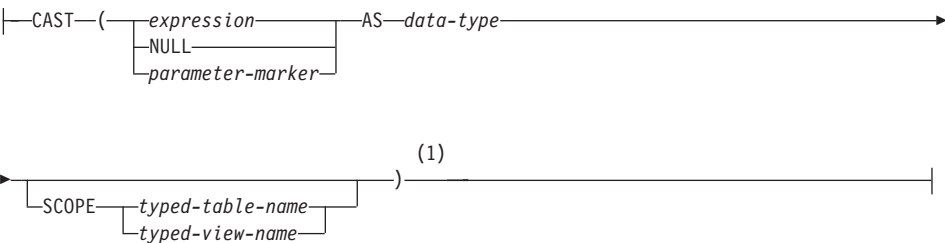


**Notes:**

**1**    The SCOPE clause only applies to the REF data type.

The CAST specification returns the cast operand (the first operand) cast to the type specified by the *data type*.

*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*.

The supported casts are shown in Table 6 on page 90 where the first column represents the data type of the cast operand (source data type) and the data types across the top represent the target data type of the CAST specification. If the cast is not supported an error will occur (SQLSTATE 42846).

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.

**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 (specified as a question mark character) 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.

*data type*

The name of an existing data type. If the type name is not qualified, the SQL path is used to do data type resolution. A data type that has an associated attributes like length or precision and scale should include these attributes when specifying *data type* (CHAR defaults to a length of 1 and DECIMAL defaults to a precision of 5 and 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*, see "Casting Between Data Types" on page 88 for the target data types that are supported based on the data type of the cast operand (source data type).

- For a cast operand that is the keyword NULL, any existing data type can be 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 type, the application using the parameter marker will use the source data type of the user-defined type.

**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 the result data type is a fixed-length character string (see "CHAR" on page 237). When character data is cast to numeric, the result data type depends on the type of number specified. For example, if cast to integer, it would become a large integer (see "INTEGER" on page 286).

**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 requirment 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 function path includes the schema name of the schema (or schemas) where the two distinct types are defined.

## Dereference Operations

**dereference-operation:**

```
├──scoped-ref-expression── -> ──attribute-name─────────────────────────────┤
```

The dereference operation returns the named column value from the target table or a subtable, or the target view or a subview, of the scoped reference expression from the row with the matching *object identifier column* (OID column). See "CREATE TABLE" on page 639 for further information on the OID column. The result can be null, regardless of whether the column corresponding to the *attribute-name* can be null. If there is no row in the target table with a matching OID, the result is null. The dereference operation takes precedence over all other operators.

*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.

*attribute-name*
> Specifies the name of an attribute of the target type (column in the target table or view) of the *scoped-ref-expression* (SQLSTATE 42704). The *attribute-name* must be an attribute of the target type of the reference. The data type of the attribute determines the data type of the result of the dereference operation. When the dereference operation is used in a select list and is not included as part of an expression, the attribute name becomes the result column name.

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).

**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
  ```

## Predicates

A *predicate* specifies a condition that is true, false, or unknown about a given 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 4 000, a graphic string with a length attribute greater than 2 000, or a LOB string of any size.
- The value of a host variable may 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, are done according to "Rules for String Conversions" on page 108
- Use of a DATALINK value is limited to the NULL predicate.

A fullselect is a form of the SELECT statement which is described under "Chapter 5. Queries" on page 367. A fullselect used in a predicate is also called a *subquery*.

## Basic Predicate

```
►►──expression──┬─ = ──────────┬──expression─────────────────────────►◄
                │      (1)      │
                ├─ <> ─────────┤
                ├─ < ──────────┤
                ├─ > ──────────┤
                │      (1)      │
                ├─ <= ─────────┤
                │      (1)      │
                └─ >= ─────────┘
```
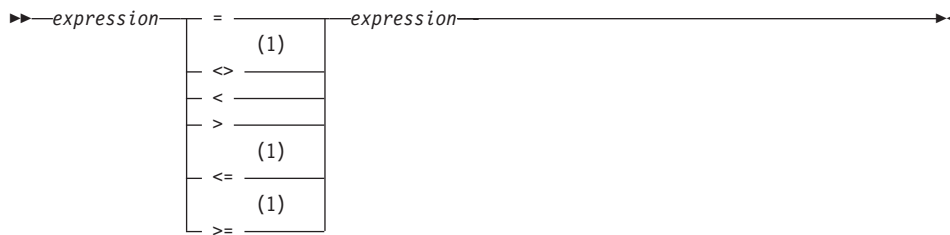
**Notes:**

**1**    Other comparison operators are also supported [34]

A *basic predicate* compares two values.

If the value of either operand is null, the result of the predicate is unknown. Otherwise the result is either true or false.

For values $x$ and $y$:

| Predicate | Is True If and Only If... |
|-----------|---------------------------|
| $x$ **=** $y$ | $x$ is equal to $y$ |
| $x$ **<>** $y$ | $x$ is not equal to $y$ |
| $x$ **<** $y$ | $x$ is less than $y$ |
| $x$ **>** $y$ | $x$ is greater than $y$ |
| $x$ **>=** $y$ | $x$ is greater than or equal to $y$ |
| $x$ **<=** $y$ | $x$ is less than or equal to $y$ |

Examples:
```
EMPNO='528671'
SALARY < 20000
PRSTAFF <> :VAR1
SALARY > (SELECT AVG(SALARY) FROM EMPLOYEE)
```

---

34. The following forms of the comparison operators are also supported in basic and quantified predicates; ˆ=, ˆ<, ˆ>, !=, !< and !>. In addition, in code pages 437, 819, and 850, the forms ¬=, ¬<, and ¬> are supported.

   All these product-specific forms of the comparison operators are intended only to support existing SQL that uses these operators, and are not recommended for use when writing new SQL statements.

## Quantified Predicate

### Quantified Predicate
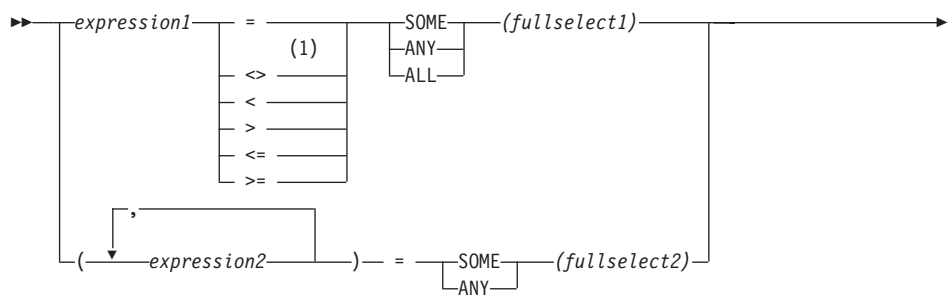


**Notes:**

**1** Other comparison operators are also supported [34].

A *quantified 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 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 11.*

*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:

## Quantified Predicate

```
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
```

## BETWEEN Predicate

```
►►──expression──┬─────┬──BETWEEN──expression──AND──expression────────────────────►◄
                └─NOT─┘
```
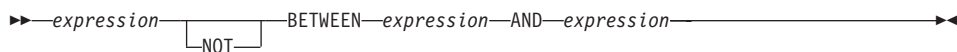
The BETWEEN predicate compares a value with a range of values.

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 values for the expressions in the BETWEEN predicate can have different code pages. The operands are converted as if the above equivalent search conditions were specified.

The first operand (expression) cannot include a function that is variant or has an external action (SQLSTATE 426804).

Given a mixture of datetime values and string representations of datetime values, all values are converted to the data type of the datetime operand.

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.

*Example 3*

## BETWEEN Predicate

Given the following:

*Table 12.*

| Expressions | Type | Code Page |
|---|---|---|
| HV_1 | host variable | 437 |
| HV_2 | host variable | 437 |
| Col_1 | column | 850 |

When evaluating the predicate:

```
:HV_1 BETWEEN :HV_2 AND COL_1
```

It will be interpreted as:

```
   :HV_1 >= :HV_2
AND :HV_1 <= COL_1
```

The first occurrence of :HV_1 will remain in the application code page since it is being compared to :HV_2 which will also remain in the application code page. The second occurrence of :HV_1 will be converted to the database code page since it is being compared to a column value.

## EXISTS Predicate

```
►►──EXISTS──(fullselect)──────────────────────────────────────────────►◄
```

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
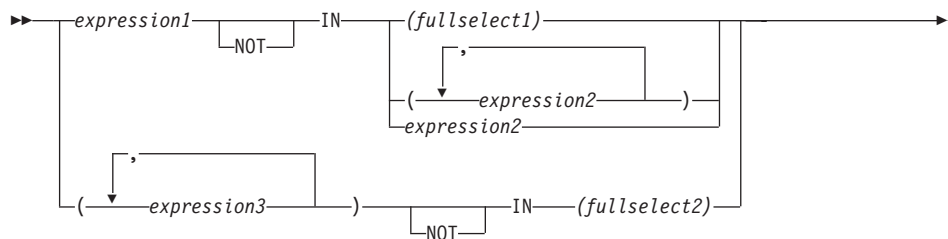


The IN predicate compares a value or values with a collection of values.

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)
```

The values for *expression1* and *expression2* or the column of *fullselect1* in the IN predicate must be compatible. Each *expression3* value and its corresponding column of *fullselect2* in the IN predicate must be compatible. The "Rules for Result Data Types" on page 105 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 then the code page is determined by applying "Rules for String Conversions" on page 108 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 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 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 13. 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" on page 108.

*Example 4:* The following 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):

## IN Predicate

```
YEAR(EMENDATE) IN (YEAR(CURRENT DATE),
                   YEAR(CURRENT DATE - 1 YEAR),
                   YEAR(CURRENT DATE - 2 YEARS))
```

*Example 5:* The following 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)
```

## LIKE Predicate

```
►►──match-expression──┬──────┬──LIKE──pattern-expression──────────────────────────►
                      └─NOT──┘
```

```
►──┬──────────────────────────────────────┬──────────────────────────────────────►◄
   └─ESCAPE──escape-expression──┘
```
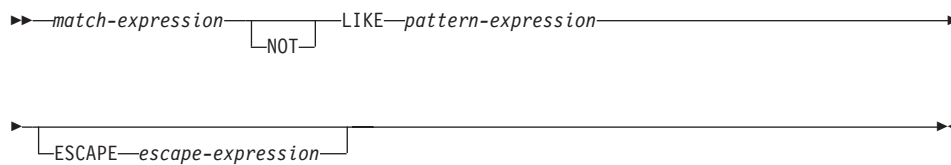
The LIKE predicate searches for strings that have a certain pattern. The pattern is specified by a string in which the underscore and percent sign may have special meanings. Trailing blanks in a pattern are part of the pattern.

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.

  The expression can be specified by any one of:
  • a constant
  • a special register
  • a host variable (including a locator variable or a file reference variable)
  • a scalar function
  • a large object locator
  • a column name
  • an expression concatenating any of the above

*pattern-expression*
  An expression that specifies the string that is to be matched.

  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 above
  • an expression concatenating any of the above

with the restrictions that:
- No element in the expression can be of type LONG VARCHAR, CLOB, LONG VARGRAPHIC 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 pattern 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 preceed either the underscore or percent character in the pattern.

A **rigorous description** of the use of the LIKE pattern 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.

It follows that 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, 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 then it can contain **mixed data**. In this case, the pattern can include both SBCS and MBCS characters. The special characters in the pattern are interpreted as follows:

- An SBCS underscore refers to one SBCS character.
- A DBCS underscore refers to one MBCS character.
- A percent (either SBCS or DBCS) refers to a string of zero or more SBCS or MBCS characters.

*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 host variable
- a scalar function whose operands are any of the above
- an expression concatenating any of the above

with the restrictions that:

- No element in the expression can be of type LONG VARCHAR, CLOB, LONG VARGRAPHIC or DBCLOB. In addition, it cannot be a BLOB file reference variable.

- The result of the expression must be one SBCS or DBCS character or a binary string containing exactly 1 byte (SQLSTATE 22019).

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 a 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 (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 (in which case there is no conversion).

**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_'
  ```

- 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 sequence of characters '%_\' in column A of the table T.

  ```
  SELECT A FROM T WHERE T.A LIKE
  '\%\_\\%' ESCAPE '\'
  ```

- Use the BLOB scalar function, to obtain a one byte escape character which is compatible with the match and pattern data types (both BLOBs).

  ```
  SELECT COLBLOB FROM TABLET
    WHERE COLBLOB LIKE :pattern_var ESCAPE BLOB(X'OE')
  ```

## NULL Predicate

```
►►──expression──IS──┬──────┬──NULL──────────────────────────────◄
                    └─NOT──┘
```
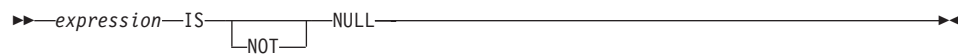
The NULL predicate tests for null values.

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.

Examples:
```
PHONENO IS NULL
SALARY IS NOT NULL
```

## TYPE Predicate

```
►►──expression──┬─IS─────────┬──OF──────────────────────────────►
                │      └NOT─┘                                     
                └─┬─IS─┐─────────OF DYNAMIC TYPE─┘               
                  └─NOT─┘                                        
```

```
                      ┌──────,─────────┐
►──(──────────────────┴──typename──────┴──)──────────────────────►◄
        └─ONLY─┘
```
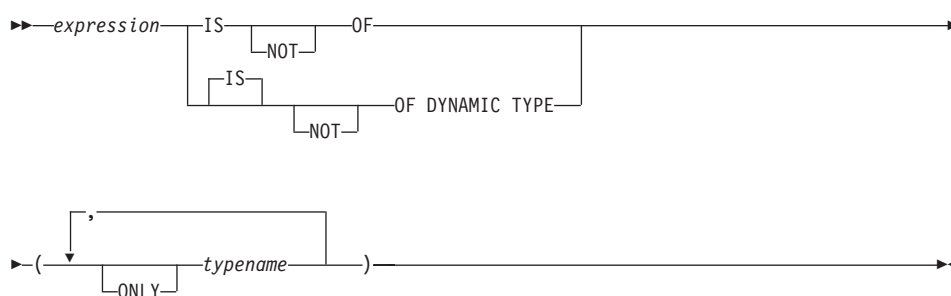
A *TYPE predicate* compares the type of an expression with one or more user-defined structured types.

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 may 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. See "DEREF" on page 260 for more information about the DEREF function.

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.

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

## TYPE Predicate

EMPLOYEE. The following is a type predicate that evaluates to true when a
row corresponding to WHO_RESPONSIBLE is a manager :

```
DEREF (WHO_RESPONSIBLE) IS OF DYNAMIC TYPE (MGR)
```

## Search Conditions

**search-condition:**
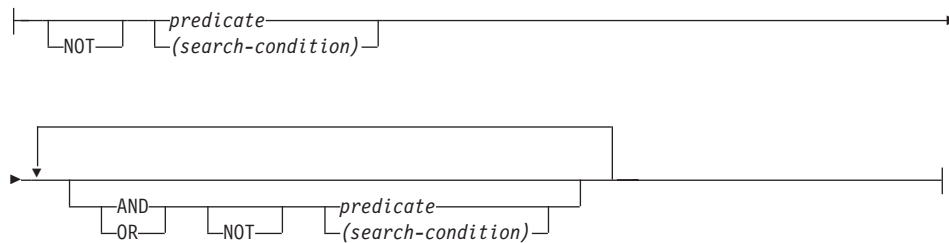


A *search condition* specifies a condition that is "true," "false," or "unknown" about a given row.

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 14, in which P and Q are any predicates:

*Table 14. 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.

MAJPROJ = 'MA2100' **AND**  DEPTNO = 'D11' **OR**  DEPTNO = 'B03' **OR** DEPTNO = 'E11'

```
          1              2 or 3            2 or 3
```

MAJPROJ = 'MA2100' **AND**  (DEPTNO = 'D11' **OR**  DEPTNO = 'B03') **OR**  DEPTNO = 'E11'

```
          2              1                 3
```
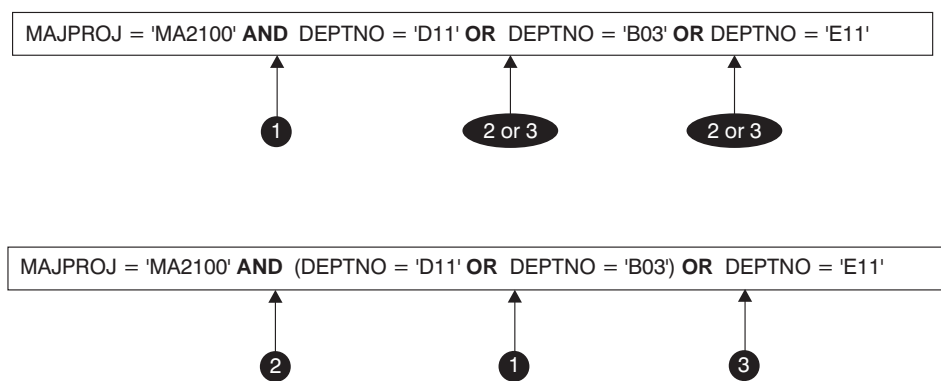
*Figure 12. Search Conditions Evaluation Order*

# Chapter 4. Functions

A function is an operation that is denoted by a function name followed by a pair of parentheses enclosing the specification of arguments (there may be no arguments).

Functions are classified as *column functions*, *scalar functions* or *table functions*. The argument of a column function is a collection of like values. It returns a single value (possibly null), and can be specified in an SQL statement where an *expression* can be used. Additional restrictions apply to the use of column functions as specified in "Column Functions" on page 205. The argument(s) of a scalar function are individual scalar values, which can be of different types and have different meanings. It returns a single value (possibly null), and can be specified in an SQL statement wherever an *expression* can be used. The argument(s) of a table function are individual scalar values, which can be of different types and have different meanings. It returns a table to the SQL statement, and can be specified only within the FROM clause of a SELECT. Additional restrictions apply to the use of table functions as specified in "from-clause" on page 374.

Table 15 on page 188 shows the functions that are supported. The "Function Name" combined with the "Schema" give the fully qualified name of the function. "Description" briefly describes what the function does. "Input Parameters" gives the data type that is expected for each argument during function invocation. Many of the functions include variations of the input parameters allowing either different data types or different numbers of arguments to be used. The combination of schema, function name and input parameters make up a function signature. Each function signature may return a value of a different type which is shown in the "Returns" columns. There are some distinctions that should be understood about the input parameter types. In some cases the type is specified as a specific built-in data type and in other cases it will use a general variable like *any-numeric-type*. When a specific data type is listed, this means that an exact match will only occur with the specified data type. When a general variable is used, each of the data types associated with that variable will result in an exact match. This distinction impacts function selection as described in "Function Resolution" on page 139.

There may be additional functions available because user-defined functions may be created in different schemas using one of these function signatures as a source (see "CREATE FUNCTION" for details) or users may create external functions using their own programs.

# Functions

*Table 15. Supported Functions*

| Function name | Schema | Description | |
|---|---|---|---|
| | **Input Parameters** | | **Returns** |
| | SYSFUN | Returns the absolute value of the argument. | |
| | SMALLINT | | SMALLINT |
| ABS or ABSVAL | INTEGER | | INTEGER |
| | BIGINT | | BIGINT |
| | DOUBLE | | DOUBLE |
| | SYSFUN | Returns the arccosine of the argument as an angle expressed in radians. | |
| ACOS | DOUBLE | | DOUBLE |
| | SYSFUN | Returns the ASCII code value of the leftmost character of the argument as an integer. | |
| | CHAR | | INTEGER |
| ASCII | VARCHAR(4000) | | INTEGER |
| | CLOB(1M) | | INTEGER |
| | SYSFUN | Returns the arcsine of the argument as an angle, expressed in radians. | |
| ASIN | DOUBLE | | DOUBLE |
| | SYSFUN | Returns the arctangent of the argument as an angle, expressed in radians. | |
| ATAN | DOUBLE | | DOUBLE |
| | SYSFUN | Returns the arctangent of $x$ and $y$ coordinates, specified by the first and second arguments respectively, as an angle, expressed in radians. | |
| ATAN2 | DOUBLE, DOUBLE | | DOUBLE |
| | SYSIBM | Returns the average of a set of numbers (column function). | |
| AVG | *numeric-type* [4] | | *numeric-type* [1] |
| | SYSIBM | Returns a 64 bit integer representation of a number or character string in the form of an integer constant. | |
| BIGINT | *numeric-type* | | BIGINT |
| | VARCHAR | | BIGINT |
| | SYSIBM | Casts from source type to BLOB, with optional length. | |
| BLOB | *string-type* | | BLOB |
| | *string-type*, INTEGER | | BLOB |
| | SYSFUN | Returns the smallest integer greater than or equal to the argument. | |
| | SMALLINT | | SMALLINT |
| CEIL or CEILING | INTEGER | | INTEGER |
| | BIGINT | | BIGINT |
| | DOUBLE | | DOUBLE |

*Table 15. Supported Functions  (continued)*

| Function name | Schema | Description | |
|---|---|---|---|
| | **Input Parameters** | | **Returns** |
| CHAR | SYSIBM | Returns a string representation of the source type. | |
| | *character-type* | | CHAR |
| | *character-type*, INTEGER | | CHAR(*integer*) |
| | *datetime-type* | | CHAR |
| | *datetime-type*, *keyword* [2] | | CHAR |
| | SMALLINT | | CHAR(6) |
| | INTEGER | | CHAR(11) |
| | BIGINT | | CHAR(20) |
| | DECIMAL | | CHAR(2+*precision*) |
| | DECIMAL, VARCHAR | | CHAR(2+*precision*) |
| CHAR | SYSFUN | Returns a character string representation of a floating-point number. | |
| | DOUBLE | | CHAR(24) |
| CHR | SYSFUN | Returns the character that has the ASCII code value specified by the argument. The value of the argument should be between 0 and 255; otherwise, the return value is null. | |
| | INTEGER | | CHAR(1) |
| CLOB | SYSIBM | Casts from source type to CLOB, with optional length. | |
| | *character-type* | | CLOB |
| | *character-type*, INTEGER | | CLOB |
| COALESCE [3] | SYSIBM | Returns the first non-null argument in the set of arguments. | |
| | *any-type*, *any-union-compatible-type*, ... | | *any-type* |
| CONCAT or | | | SYSIBM | Returns the concatenation of 2 string arguments. | |
| | *string-type*, *compatible-string-type* | | *max string-type* |
| CORRELATION or CORR | SYSIBM | Returns the coefficient of correlation of a set of number pairs. | |
| | *numeric-type*, *numeric-type* | | DOUBLE |
| COS | SYSFUN | Returns the cosine of the argument, where the argument is an angle expressed in radians. | |
| | DOUBLE | | DOUBLE |
| COT | SYSFUN | Returns the cotangent of the argument, where the argument is an angle expressed in radians. | |
| | DOUBLE | | DOUBLE |
| COUNT | SYSIBM | Returns the count of the number of rows in a set of rows or values (column function). | |
| | *any-builtin-type* [4] | | INTEGER |
| COUNT_BIG | SYSIBM | Returns the number of rows or values in a set of rows or values (column function). Result can be greater than the maximum value of integer. | |
| | *any-builtin-type* [4] | | DECIMAL(31,0) |

## Functions

*Table 15. Supported Functions  (continued)*

| Function name | Schema | Description | |
|---|---|---|---|
| | **Input Parameters** | | **Returns** |
| COVARIANCE or COVAR | SYSIBM | Returns the covariance of a set of number pairs. | |
| | *numeric-type, numeric-type* | | DOUBLE |
| DATE | SYSIBM | Returns a date from a single input value. | |
| | DATE | | DATE |
| | TIMESTAMP | | DATE |
| | DOUBLE | | DATE |
| | VARCHAR | | DATE |
| DAY | SYSIBM | Returns the day part of a value. | |
| | VARCHAR | | INTEGER |
| | DATE | | INTEGER |
| | TIMESTAMP | | INTEGER |
| | DECIMAL | | INTEGER |
| DAYNAME | SYSFUN | Returns a mixed case character string containing the name of the day (e.g. Friday) for the day portion of the argument based on what the locale was when db2start was issued. | |
| | VARCHAR(26) | | VARCHAR(100) |
| | DATE | | VARCHAR(100) |
| | TIMESTAMP | | VARCHAR(100) |
| DAYOFWEEK | SYSFUN | Returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Sunday. | |
| | VARCHAR(26) | | INTEGER |
| | DATE | | INTEGER |
| | TIMESTAMP | | INTEGER |
| DAYOFYEAR | SYSFUN | Returns the day of the year in the argument as an integer value in the range 1-366. | |
| | VARCHAR(26) | | INTEGER |
| | DATE | | INTEGER |
| | TIMESTAMP | | INTEGER |
| DAYS | SYSIBM | Returns an integer representation of a date. | |
| | VARCHAR | | INTEGER |
| | TIMESTAMP | | INTEGER |
| | DATE | | INTEGER |
| DBCLOB | SYSIBM | Casts from source type to DBCLOB, with optional length. | |
| | *graphic-type* | | DBCLOB |
| | *graphic-type*, INTEGER | | DBCLOB |

*Table 15. Supported Functions  (continued)*

| Function name | Schema | Description | |
|---|---|---|---|
| | **Input Parameters** | | **Returns** |
| DECIMAL or DEC | SYSIBM | Returns decimal representation of a number, with optional precision and scale. | |
| | *numeric-type* | | DECIMAL |
| | *numeric-type*, INTEGER | | DECIMAL |
| | *numeric-type* INTEGER, INTEGER | | DECIMAL |
| DECIMAL or DEC | SYSIBM | Returns decimal representation of a character string, with optional precision, scale, and decimal-character. | |
| | VARCHAR | | DECIMAL |
| | VARCHAR, INTEGER | | DECIMAL |
| | VARCHAR, INTEGER, INTEGER | | DECIMAL |
| | VARCHAR, INTEGER, INTEGER, VARCHAR | | DECIMAL |
| DEGREES | SYSFUN | Returns the number of degrees converted from the argument in expressed in radians. | |
| | DOUBLE | | DOUBLE |
| DEREF | SYSIBM | Returns an instance of the target type of the reference type argument. | |
| | REF(*any-structured-type*) *with defined scope* | | *any-structured-type (same as input target type)* |
| DIFFERENCE | SYSFUN | Returns the difference between the sounds of the words in the two argument strings as determined using the SOUNDEX function. A value of 4 means the strings sound the same. | |
| | VARCHAR(4000), VARCHAR(4000) | | INTEGER |
| DIGITS | SYSIBM | Returns the character string representation of a number. | |
| | DECIMAL | | CHAR |
| DLCOMMENT | SYSIBM | Returns the comment attribute of a datalink value. | |
| | DATALINK | | VARCHAR(254) |
| DLLINKTYPE | SYSIBM | Returns the link type attribute of a datalink value. | |
| | DATALINK | | VARCHAR(4) |
| DLURLCOMPLETE | SYSIBM | Returns the complete URL (including access token) of a datalink value. | |
| | DATALINK | | VARCHAR |
| DLURLPATH | SYSIBM | Returns the path and file name (including access token) of a datalink value. | |
| | DATALINK | | VARCHAR |
| DLURLPATHONLY | SYSIBM | Returns the path and file name (without any access token) of a datalink value. | |
| | DATALINK | | VARCHAR |
| DLURLSCHEME | SYSIBM | Returns the scheme from the URL attribute of a datalink value. | |
| | DATALINK | | VARCHAR |

# Functions

*Table 15. Supported Functions  (continued)*

| Function name | Schema | Description | |
|---|---|---|---|
| | **Input Parameters** | | **Returns** |
| DLURLSERVER | SYSIBM | Returns the server from the URL attribute of a datalink value. | |
| | DATALINK | | VARCHAR |
| DLVALUE | SYSIBM | Builds a datalink value from a data-location argument, link type argument and optional comment-string argument. | |
| | VARCHAR | | DATALINK |
| | VARCHAR, VARCHAR | | DATALINK |
| | VARCHAR, VARCHAR, VARCHAR | | DATALINK |
| DOUBLE or DOUBLE_PRECISION | SYSIBM | Returns the floating-point representation of a number. | |
| | *numeric-type* | | DOUBLE |
| DOUBLE | SYSFUN | Returns the floating-point number corresponding to the character string representation of a number. Leading and trailing blanks in *argument* are ignored. | |
| | VARCHAR | | DOUBLE |
| EVENT_MON_STATE | SYSIBM | Returns the operational state of particular event monitor. | |
| | VARCHAR | | INTEGER |
| EXP | SYSFUN | Returns the exponential function of the argument. | |
| | DOUBLE | | DOUBLE |
| FLOAT | SYSIBM | Same as DOUBLE. | |
| FLOOR | SYSFUN | Returns the largest integer less than or equal to the argument. | |
| | SMALLINT | | SMALLINT |
| | INTEGER | | INTEGER |
| | BIGINT | | BIGINT |
| | DOUBLE | | DOUBLE |
| GENERATE_UNIQUE | SYSIBM | Returns a bit data character string that is unique compared to any other execution of the same function. | |
| | *no argument* | | CHAR(13) FOR BIT DATA |
| GRAPHIC | SYSIBM | Cast from source type to GRAPHIC, with optional length. | |
| | *graphic-type* | | GRAPHIC |
| | *graphic-type*, INTEGER | | GRAPHIC |
| GROUPING | SYSIBM | Used with grouping-sets and super-groups to indicate sub-total rows generated by a grouping set (column function). The value returned is:<br><br>**1**      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.<br><br>**0**      otherwise. | |
| | *any-type* | | SMALLINT |

*Table 15. Supported Functions (continued)*

| Function name | Schema | Description | |
|---|---|---|---|
| | **Input Parameters** | | **Returns** |
| HEX | SYSIBM | Returns the hexadecimal representation of a value. | |
| | *any-builtin-type* | | VARCHAR |
| HOUR | SYSIBM | Returns the hour part of a value. | |
| | VARCHAR | | INTEGER |
| | TIME | | INTEGER |
| | TIMESTAMP | | INTEGER |
| | DECIMAL | | INTEGER |
| INSERT | SYSFUN | Returns a string where *argument3* bytes have been deleted from *argument1* beginning at *argument2* and where *argument4* has been inserted into *argument1* beginning at *argument2*. | |
| | VARCHAR(4000), INTEGER, INTEGER, VARCHAR(4000) | | VARCHAR(4000) |
| | CLOB(1M), INTEGER, INTEGER, CLOB(1M) | | CLOB(1M) |
| | BLOB(1M), INTEGER, INTEGER, BLOB(1M) | | BLOB(1M) |
| INTEGER or INT | SYSIBM | Returns the integer representation of a number. | |
| | *numeric-type* | | INTEGER |
| | VARCHAR | | INTEGER |
| JULIAN_DAY | SYSFUN | Returns an integer value representing the number of days from January 1, 4712 B.C. (the start of the Julian date calendar) to the date value specified in the *argument*. | |
| | VARCHAR(26) | | INTEGER |
| | DATE | | INTEGER |
| | TIMESTAMP | | INTEGER |
| LCASE or LOWER | SYSIBM | Returns a string in which all the characters have been converted to lower case characters. | |
| | CHAR | | CHAR |
| | VARCHAR | | VARCHAR |
| LCASE | SYSFUN | Returns a string in which all the characters have been converted to lower case characters. LCASE will only handle characters in the invariant set. Therefore, LCASE(UCASE(string)) will not necessarily return the same result as LCASE(string). | |
| | VARCHAR(4000) | | VARCHAR(4000) |
| | CLOB(1M) | | CLOB(1M) |
| LEFT | SYSFUN | Returns a string consisting of the leftmost *argument2* bytes in *argument1*. | |
| | VARCHAR(4000), INTEGER | | VARCHAR(4000) |
| | CLOB(1M), INTEGER | | CLOB(1M) |
| | BLOB(1M), INTEGER | | BLOB(1M) |

# Functions

*Table 15. Supported Functions  (continued)*

| Function name | Schema | Description | |
|---|---|---|---|
| | **Input Parameters** | | **Returns** |
| LENGTH | SYSIBM | Returns the length of the operand in bytes (except for double byte string types which return the length in characters). | |
| | *any-builtin-type* | | INTEGER |
| LN | SUSFUN | Returns the natural logarithm of the argument (same as LOG). | |
| | DOUBLE | | DOUBLE |
| LOCATE | SYSFUN | Returns the starting position of the first occurrence of *argument1* within *argument2*. If the optional third argument is specified, it indicates the character position in *argument2* at which the search is to begin. If *argument1* is not found within *argument2*, the value 0 is returned. | |
| | VARCHAR(4000), VARCHAR(4000) | | INTEGER |
| | VARCHAR(4000), VARCHAR(4000), INTEGER | | INTEGER |
| | CLOB(1M), CLOB(1M) | | INTEGER |
| | CLOB(1M), CLOB(1M), INTEGER | | INTEGER |
| | BLOB(1M), BLOB(1M) | | INTEGER |
| | BLOB(1M), BLOB(1M), INTEGER | | INTEGER |
| LOG | SYSFUN | Returns the natural logarithm of the argument (same as LN). | |
| | DOUBLE | | DOUBLE |
| LOG10 | | Returns the base 10 logarithm of the argument. | |
| | DOUBLE | | DOUBLE |
| LONG_VARCHAR | SYSIBM | Returns a long string. | |
| | *character-type* | | LONG VARCHAR |
| LONG_VARGRAPHIC | SYSIBM | Casts from source type to LONG_VARGRAPHIC. | |
| | *graphic-type* | | LONG VARGRAPHIC |
| LTRIM | SYSIBM | Returns the characters of the argument with leading blanks removed. | |
| | CHAR | | VARCHAR |
| | VARCHAR | | VARCHAR |
| | GRAPHIC | | VARGRAPHIC |
| | VARGRAPHIC | | VARGRAPHIC |
| LTRIM | SYSFUN | Returns the characters of the argument with leading blanks removed. | |
| | VARCHAR(4000) | | VARCHAR(4000) |
| | CLOB(1M) | | CLOB(1M) |
| MAX | SYSIBM | Returns the maximum value in a set of values (column function). | |
| | *any-builtin-type* [5] | | *same as input type* |

*Table 15. Supported Functions (continued)*

| Function name | Schema | Description | |
|---|---|---|---|
| | **Input Parameters** | | **Returns** |
| MICROSECOND | SYSIBM | Returns the microsecond (time-unit) part of a value. | |
| | VARCHAR | | INTEGER |
| | TIMESTAMP | | INTEGER |
| | DECIMAL | | INTEGER |
| MIDNIGHT_SECONDS | SYSFUN | Returns an integer value in the range 0 to 86 400 representing the number of seconds between midnight and time value specified in the *argument*. | |
| | VARCHAR(26) | | INTEGER |
| | TIME | | INTEGER |
| | TIMESTAMP | | INTEGER |
| MIN | SYSIBM | Returns the minimum value in a set of values (column function). | |
| | *any-builtin-type* [5] | | *same as input type* |
| MINUTE | SYSIBM | Returns the minute part of a value. | |
| | VARCHAR | | INTEGER |
| | TIME | | INTEGER |
| | TIMESTAMP | | INTEGER |
| | DECIMAL | | INTEGER |
| MOD | SYSFUN | Returns the remainder ( modulus) of *argument1* divided by *argument2*. The result is negative only if *argument1* is negative. | |
| | SMALLINT, SMALLINT | | SMALLINT |
| | INTEGER, INTEGER | | INTEGER |
| | BIGINT, BIGINT | | BIGINT |
| MONTH | SYSIBM | Returns the month part of a value. | |
| | VARCHAR | | INTEGER |
| | DATE | | INTEGER |
| | TIMESTAMP | | INTEGER |
| | DECIMAL | | INTEGER |
| MONTHNAME | SYSFUN | Returns a mixed case character string containing the name of month (e.g. January) for the month portion of the argument that is a date or timestamp, based on what the locale was when the database was started. | |
| | VARCHAR(26) | | VARCHAR(100) |
| | DATE | | VARCHAR(100) |
| | TIMESTAMP | | VARCHAR(100) |
| NODENUMBER [3] | SYSIBM | Returns the node number of the row. The argument is a column name within a table. | |
| | *any-type* | | INTEGER |

# Functions

*Table 15. Supported Functions  (continued)*

| Function name | Schema | Description | |
|---|---|---|---|
| | **Input Parameters** | | **Returns** |
| NULLIF [3] | SYSIBM | Returns NULL if the arguments are equal, else returns the first argument. | |
| | *any-type* [5], *any-comparable-type*[5] | | *any-type* |
| PARTITION [3] | SYSIBM | Returns the partitioning map index (0 to 4095) of the row. The argument is a column name within a table. | |
| | *any-type* | | INTEGER |
| POSSTR | SYSIBM | Returns the position at which one string is contained in another. | |
| | *string-type, compatible-string-type* | | INTEGER |
| POWER | SYSFUN | Returns the value of *argument1* to the power of *argument2*. | |
| | INTEGER, INTEGER | | INTEGER |
| | BIGINT, BIGINT | | BIGINT |
| | DOUBLE, INTEGER | | DOUBLE |
| | DOUBLE, DOUBLE | | DOUBLE |
| QUARTER | SYSFUN | Returns an integer value in the range 1 to 4 representing the quarter of the year for the date specified in the argument. | |
| | VARCHAR(26) | | INTEGER |
| | DATE | | INTEGER |
| | TIMESTAMP | | INTEGER |
| RADIANS | SYSFUN | Returns the number of radians converted from argument which is expressed in degrees. | |
| | DOUBLE | | DOUBLE |
| RAISE_ERROR | SYSIBM | Raises an error in the SQLCA. The sqlstate returned is indicated by *argument1*. The second argument contains any text to be returned. | |
| | VARCHAR, VARCHAR | | *any-type* [6] |
| RAND | SYSFUN | Returns a random floating point value between 0 and 1 using the argument as the optional seed value. | |
| | *no argument required* | | DOUBLE |
| | INTEGER | | DOUBLE |
| REAL | SYSIBM | Returns the single-precision floating-point representation of a number. | |
| | *numeric-type* | | REAL |
| REGR_AVGX | SYSIBM | Returns quantities used to compute diagnostic statistics. | |
| | *numeric-type, numeric-type* | | DOUBLE |
| REGR_AVGY | SYSIBM | Returns quantities used to compute diagnostic statistics. | |
| | *numeric-type, numeric-type* | | DOUBLE |
| REGR_COUNT | SYSIBM | Returns the the number of non-null number pairs used to fit the regression line. | |
| | *numeric-type, numeric-type* | | INTEGER |

*Table 15. Supported Functions (continued)*

| Function name | Schema | Description | |
|---|---|---|---|
| | **Input Parameters** | | **Returns** |
| REGR_INTERCEPT or REGR_ICPT | SYSIBM | Returns the y-intercept of the regression line. | |
| | *numeric-type, numeric-type* | | DOUBLE |
| REGR_R2 | SYSIBM | Returns the coefficient of determination for the regression. | |
| | *numeric-type, numeric-type* | | DOUBE |
| REGR_SLOPE | SYSIBM | Returns the slope of the line. | |
| | *numeric-type, numeric-type* | | DOUBLE |
| REGR_SXX | SYSIBM | Returns quantities used to compute diagnostic statistics. | |
| | *numeric-type, numeric-type* | | DOUBLE |
| REGR_SXY | SYSIBM | Returns quantities used to compute diagnostic statistics. | |
| | *numeric-type, numeric-type* | | DOUBLE |
| REGR_SYY | SYSIBM | Returns quantities used to compute diagnostic statistics. | |
| | *numeric-type, numeric-type* | | DOUBLE |
| REPEAT | SYSFUN | Returns a character string composed of *argument1* repeated *argument2* times. | |
| | VARCHAR(4000), INTEGER | | VARCHAR(4000) |
| | CLOB(1M), INTEGER | | CLOB(1M) |
| | BLOB(1M), INTEGER | | BLOB(1M) |
| REPLACE | SYSFUN | Replaces all occurrences of *argument2* in *argument1* with *argument3*. | |
| | VARCHAR(4000), VARCHAR(4000), VARCHAR(4000) | | VARCHAR(4000) |
| | CLOB(1M), CLOB(1M), CLOB(1M) | | CLOB(1M) |
| | BLOB(1M), BLOB(1M), BLOB(1M) | | BLOB(1M) |
| RIGHT | SYSFUN | Returns a string consisting of the rightmost *argument2* bytes in *argument1*. | |
| | VARCHAR(4000), INTEGER | | VARCHAR(4000) |
| | CLOB(1M), INTEGER | | CLOB(1M) |
| | BLOB(1M), INTEGER | | BLOB(1M) |
| ROUND | SYSFUN | Returns the first argument rounded to *argument2* places **right** of the decimal point. If *argument2* is negative, *argument1* is rounded to the absolute value of *argument2* places to the **left** of the decimal point. | |
| | INTEGER, INTEGER | | INTEGER |
| | BIGINT, INTEGER | | BIGINT |
| | DOUBLE, INTEGER | | DOUBLE |
| RTRIM | SYSIBM | Returns the characters of the argument with trailing blanks removed. | |
| | CHAR | | VARCHAR |
| | VARCHAR | | VARCHAR |
| | GRAPHIC | | VARGRAPHIC |
| | VARGRAPHIC | | VARGRAPHIC |

## Functions

*Table 15. Supported Functions (continued)*

| Function name | Schema | Description | |
|---|---|---|---|
| | **Input Parameters** | | **Returns** |
| | SYSFUN | Returns the characters of the argument with trailing blanks removed. | |
| RTRIM | VARCHAR(4000) | | VARCHAR(4000) |
| | CLOB(1M) | | CLOB(1M) |
| | SYSIBM | Returns the second (time-unit) part of a value. | |
| | VARCHAR | | INTEGER |
| SECOND | TIME | | INTEGER |
| | TIMESTAMP | | INTEGER |
| | DECIMAL | | INTEGER |
| | SYSFUN | Returns an indicator of the sign of the argument. If the argument is less than zero, -1 is returned. If argument equals zero, 0 is returned. If argument is greater than zero, 1 is returned. | |
| SIGN | SMALLINT | | SMALLINT |
| | INTEGER | | INTEGER |
| | BIGINT | | BIGINT |
| | DOUBLE | | DOUBLE |
| | SYSFUN | Returns the sine of the argument, where the argument is an angle expressed in radians. | |
| SIN | DOUBLE | | DOUBLE |
| | SYSIBM | Returns the small integer representation of a number. | |
| SMALLINT | *numeric-type* | | SMALLINT |
| | VARCHAR | | SMALLINT |
| | SYSFUN | 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. See also DIFFERENCE. | |
| SOUNDEX | VARCHAR(4000) | | CHAR(4) |
| | SYSFUN | Returns a character string consisting of *argument1* blanks. | |
| SPACE | INTEGER | | VARCHAR(4000) |
| | SYSFUN | Returns a table of the snapshot of the db2 dynamic SQL statement cache. | |
| SQLCACHE_SNAPSHOT | Refer to "SQLCACHE_SNAPSHOT" on page 364. | | |
| | SYSFUN | Returns the square root of the argument. | |
| SQRT | DOUBLE | | DOUBLE |
| | SYSIBM | Returns the standard deviation of a set of numbers (column function). | |
| STDDEV | DOUBLE | | DOUBLE |

*Table 15. Supported Functions  (continued)*

| Function name | Schema | Description | |
|---|---|---|---|
| | **Input Parameters** | | **Returns** |
| SUBSTR | SYSIBM | Returns a substring of a string *argument1* starting at *argument2* for *argument3* characters. If *argument3* is not specified, the remainder of the string is assumed. | |
| | *string-type*, INTEGER | | *string-type* |
| | *string-type*, INTEGER, INTEGER | | *string-type* |
| SUM | SYSIBM | Returns the sum of a set of numbers (column function). | |
| | *numeric-type* [4] | | *max-numeric-type* [1] |
| TABLE_NAME | SYSIBM | Returns an unqualified name of a table or view based on the object name given in *argument1* and the optional schema name given in *argument2*. It is used to resolve aliases. | |
| | VARCHAR | | VARCHAR(128) |
| | VARCHAR, VARCHAR | | VARCHAR(128) |
| TABLE_SCHEMA | SYSIBM | Returns the schema name portion of the two part table or view name given by the object name in *argument1* and the optional schema name in *argument2*. It is used to resolve aliases. | |
| | VARCHAR | | VARCHAR(128) |
| | VARCHAR, VARCHAR | | VARCHAR(128) |
| TAN | SYSFUN | Returns the tangent of the argument, where the argument is an angle expressed in radians. | |
| | DOUBLE | | DOUBLE |
| TIME | SYSIBM | Returns a time from a value. | |
| | TIME | | TIME |
| | TIMESTAMP | | TIME |
| | VARCHAR | | TIME |
| TIMESTAMP | SYSIBM | Returns a timestamp from a value or a pair of values. | |
| | TIMESTAMP | | TIMESTAMP |
| | VARCHAR | | TIMESTAMP |
| | VARCHAR, VARCHAR | | TIMESTAMP |
| | VARCHAR, TIME | | TIMESTAMP |
| | DATE, VARCHAR | | TIMESTAMP |
| | DATE, TIME | | TIMESTAMP |

# Functions

Table 15. Supported Functions  (continued)

| Function name | Schema | Description | |
|---|---|---|---|
| | **Input Parameters** | | **Returns** |
| | SYSFUN | 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. | |
| TIMESTAMP_ISO | DATE | | TIMESTAMP |
| | TIME | | TIMESTAMP |
| | TIMESTAMP | | TIMESTAMP |
| | VARCHAR(26) | | TIMESTAMP |
| TIMESTAMPDIFF | SYSFUN | 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. Valid values of interval (*argument1*) are: **1** Fractions of a second **2** Seconds **4** Minutes **8** Hours **16** Days **32** Weeks **64** Months **128** Quarters **256** Years | |
| | INTEGER, CHAR(22) | | INTEGER |
| TRANSLATE | SYSIBM | Returns a string in which one or more characters may have been translated into other characters. | |
| | CHAR | | CHAR |
| | VARCHAR | | VARCHAR |
| | CHAR, VARCHAR, VARCHAR | | CHAR |
| | VARCHAR, VARCHAR, VARCHAR | | VARCHAR |
| | CHAR, VARCHAR, VARCHAR, VARCHAR | | CHAR |
| | VARCHAR, VARCHAR, VARCHAR, VARCHAR | | VARCHAR |
| | GRAPHIC, VARGRAPHIC, VARGRAPHIC | | GRAPHIC |
| | VARGRAPHIC, VARGRAPHIC, VARGRAPHIC | | VARGRAPHIC |
| | GRAPHIC, VARGRAPHIC, VARGRAPHIC, VARGRAPHIC | | GRAPHIC |
| | VARGRAPHIC, VARGRAPHIC, VARGRAPHIC, VARGRAPHIC | | VARGRAPHIC |

*Table 15. Supported Functions (continued)*

| Function name | Schema | Description | |
|---|---|---|---|
| | **Input Parameters** | | **Returns** |
| TRUNC or TRUNCATE | SYSFUN | Returns *argument1* truncated to *argument2* places **right** of the decimal point. If *argument2* is negative, *argument1* is truncated to the absolute value of *argument2* places to the **left** of the decimal point. | |
| | INTEGER, INTEGER | | INTEGER |
| | BIGINT, INTEGER | | BIGINT |
| | DOUBLE, INTEGER | | DOUBLE |
| TYPE_ID [3] | SYSIBM | Returns the internal data type identifier of the dynamic data type of the argument. Note that the result of this function is not portable across databases. | |
| | *any-structured-type* | | INTEGER |
| TYPE_NAME [3] | SYSIBM | Returns the unqualified name of the dynamic data type of the argument. | |
| | *any-structured-type* | | VARCHAR(18) |
| TYPE_SCHEMA [3] | SYSIBM | Returns the schema name of the dynamic type of the argument. | |
| | *any-structured-type* | | VARCHAR(128) |
| UCASE or UPPER | SYSIBM | Returns a string in which all the characters have been converted to upper case characters. | |
| | CHAR | | CHAR |
| | VARCHAR | | VARCHAR |
| UCASE | SYSFUN | Returns a string in which all the characters have been converted to upper case characters. | |
| | VARCHAR | | VARCHAR |
| VALUE [3] | SYSIBM | Same as COALESCE. | |
| VARCHAR | SYSIBM | Returns a VARCHAR representation of the first argument. If a second argument is present, it specifies the length of the result. | |
| | *character-type* | | VARCHAR |
| | *character-type*, INTEGER | | VARCHAR |
| | *datetime-type* | | VARCHAR |
| VARGRAPHIC | SYSIBM | Returns a VARGRAPHIC representation of the first argument. If a second argument is present, it specifies the length of the result. | |
| | *graphic-type* | | VARGRAPHIC |
| | *graphic-type*, INTEGER | | VARGRAPHIC |
| | VARCHAR | | VARGRAPHIC |
| VARIANCE or VAR | SYSIBM | Returns the variance of a set of numbers (column function). | |
| | DOUBLE | | DOUBLE |

## Functions

*Table 15. Supported Functions  (continued)*

| Function name | Schema | Description | |
|---|---|---|---|
| | **Input Parameters** | | **Returns** |
| WEEK | SYSFUN | Returns the week of the year in of the argument as an integer value in the range of 1-54. | |
| | VARCHAR(26) | | INTEGER |
| | DATE | | INTEGER |
| | TIMESTAMP | | INTEGER |
| YEAR | SYSIBM | Returns the year part of a value. | |
| | VARCHAR | | INTEGER |
| | DATE | | INTEGER |
| | TIMESTAMP | | INTEGER |
| | DECIMAL | | INTEGER |
| "+" | SYSIBM | Adds two numeric operands. | |
| | *numeric-type, numeric-type* | | *max numeric-type* |
| "+" | SYSIBM | Unary plus operator. | |
| | *numeric-type* | | *numeric-type* |
| "+" | SYSIBM | Datetime plus operator. | |
| | DATE, DECIMAL(8,0) | | DATE |
| | TIME, DECIMAL(6,0) | | TIME |
| | TIMESTAMP, DECIMAL(20,6) | | TIMESTAMP |
| | DECIMAL(8,0), DATE | | DATE |
| | DECIMAL(6,0), TIME | | TIME |
| | DECIMAL(20,6), TIMESTAMP | | TIMESTAMP |
| | *datetime-type*, DOUBLE, *labeled-duration-code* | | *datetime-type* |
| "–" | SYSIBM | Subtracts two numeric operands. | |
| | *numeric-type, numeric-type* | | *max numeric-type* |
| "–" | SYSIBM | Unary minus operator. | |
| | *numeric-type* | | *numeric-type* [1] |

Functions

*Table 15. Supported Functions  (continued)*

| Function name | Schema | Description | |
|---|---|---|---|
| | **Input Parameters** | | **Returns** |
| | SYSIBM | Datetime minus operator. | |
| | DATE, DATE | | DECIMAL(8,0) |
| | TIME, TIME | | DECIMAL(6,0) |
| | TIMESTAMP, TIMESTAMP | | DECIMAL(20,6) |
| | DATE, VARCHAR | | DECIMAL(8,0) |
| | TIME, VARCHAR | | DECIMAL(6,0) |
| | TIMESTAMP, VARCHAR | | DECIMAL(20,6) |
| "–" | VARCHAR, DATE | | DECIMAL(8,0) |
| | VARCHAR, TIME | | DECIMAL(6,0) |
| | VARCHAR, TIMESTAMP | | DECIMAL(20,6) |
| | DATE, DECIMAL(8,0) | | DATE |
| | TIME, DECIMAL(6,0) | | TIME |
| | TIMESTAMP, DECIMAL(20,6) | | TIMESTAMP |
| | *datetime-type*, DOUBLE, *labeled-duration-code* | | *datetime-type* |
| "*" | SYSIBM | Multiplies two numeric operands. | |
| | *numeric-type, numeric-type* | | *max numeric-type* |
| "/" | SYSIBM | Divides two numeric operands. | |
| | *numeric-type, numeric-type* | | *max numeric-type* |
| "‖" | SYSIBM | Same as CONCAT. | |

**Notes**

- References to string data types that are not qualified by a length should be assumed to support the maximum length for the data type
- References to a DECIMAL data type without precision and scale should be assumed to allow any supported precision and scale.

## Functions

<table>
<tr><td colspan="2"><strong>Key to Table</strong></td></tr>
<tr><td><em>any-builtin-type</em></td><td>Any data type that is not a distinct type.</td></tr>
<tr><td><em>any-type</em></td><td>Any type defined to the database.</td></tr>
<tr><td><em>any-structured-type</em></td><td></td></tr>
<tr><td></td><td>Any user-defined structured type defined to the database.</td></tr>
<tr><td><em>any-comparable-type</em></td><td></td></tr>
<tr><td></td><td>Any type that is comparable with other argument types as defined in "Assignments and Comparisons" on page 91.</td></tr>
<tr><td><em>any-union-compatible-type</em></td><td></td></tr>
<tr><td></td><td>Any type that is compatible with other argument types as defined in "Rules for Result Data Types" on page 105.</td></tr>
<tr><td><em>character-type</em></td><td>Any of the character string types: CHAR, VARCHAR, LONG VARCHAR, CLOB.</td></tr>
<tr><td><em>compatible-string-type</em></td><td></td></tr>
<tr><td></td><td>A string type that comes from the same grouping as the other argument (e.g. if one argument is a <em>character-type</em> the other must also be a <em>character-type</em>).</td></tr>
<tr><td><em>datetime-type</em></td><td>Any of the datetime types: DATE, TIME, TIMESTAMP.</td></tr>
<tr><td><em>graphic-type</em></td><td>Any of the double byte character string types: GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, DBCLOB.</td></tr>
<tr><td><em>labeled-duration-code</em></td><td></td></tr>
<tr><td></td><td>As a type this is a SMALLINT. If the function is invoked using the infix form of the plus or minus operator, labeled-durations as defined in "Labeled Durations" on page 153 can be used. For a source function that does not use the plus or minus operator character as the name, the following values must be used for the labeled-duration-code argument when invoking the function.</td></tr>
</table>

| | |
|---|---|
| **1** | YEAR or YEARS |
| **2** | MONTH or MONTHS |
| **3** | DAY or DAYS |
| **4** | HOUR or HOURS |
| **5** | MINUTE or MINUTES |
| **6** | SECOND or SECONDS |
| **7** | MICROSECOND or MICROSECONDS |

<table>
<tr><td><em>LOB-type</em></td><td>Any of the large object types: BLOB, CLOB, DBCLOB.</td></tr>
<tr><td><em>max-numeric-type</em></td><td>The maximum numeric type of the arguments where maximum is defined as the rightmost <em>numeric-type</em>.</td></tr>
<tr><td><em>max-string-type</em></td><td>The maximum string type of the arguments where maximum is defined as the rightmost <em>character-type</em> or <em>graphic-type</em>. If arguments are BLOB, the <em>max-string-type</em> is BLOB.</td></tr>
<tr><td><em>numeric-type</em></td><td>Any of the numeric types: SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, DOUBLE.</td></tr>
<tr><td><em>string-type</em></td><td>Any type from <em>character type</em>, <em>graphic-type</em> or BLOB.</td></tr>
</table>

<table>
<tr><td colspan="2"><strong>Table Footnotes</strong></td></tr>
<tr><td><strong>1</strong></td><td>When the input parameter is SMALLINT, the result type is INTEGER. When the input parameter is REAL, the result type is DOUBLE.</td></tr>
<tr><td><strong>2</strong></td><td>Keywords allowed are ISO, USA, EUR, JIS, and LOCAL. This function signature is not supported as a sourced function.</td></tr>
<tr><td><strong>3</strong></td><td>This function cannot be used as a source function.</td></tr>
<tr><td><strong>4</strong></td><td>The keyword ALL or DISTINCT may be used before the first parameter. If DISTINCT is specified, use of long string types or a DATALINK type is not supported.</td></tr>
<tr><td><strong>5</strong></td><td>Use of long string types or a DATALINK type is not supported.</td></tr>
<tr><td><strong>6</strong></td><td>The type returned by RAISE_ERROR depends upon the context of its use. RAISE_ERROR, if not cast to a particular type, will return a type appropriate to its invocation within a CASE expression.</td></tr>
</table>

## Column Functions

The argument of a column function is a set of values derived from an expression. The expression may include columns but cannot include a *scalar-fullselect* or another column function (SQLSTATE 42607). The scope of the set is a group or an intermediate result table as explained in "Chapter 5. Queries" on page 367.

If a GROUP BY clause is specified in a query and the intermediate result from the FROM, WHERE, GROUP BY and HAVING clauses is the empty set; then the column 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 is of the FROM, WHERE, and HAVING clauses is the empty set, then the column 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 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. If ALL is implicitly or explicitly specified, duplicate values are not eliminated.

Expressions can be used in column functions, for example:

```
SELECT MAX(BONUS + 1000)
  INTO :TOP_SALESREP_BONUS
  FROM EMPLOYEE
  WHERE COMM > 5000
```

The column functions that follow are in the SYSIBM schema and may be qualified with the schema name (for example, SYSIBM.COUNT(*)).

## AVG

```
>>─AVG─(──┬──ALL──────┬──expression──)──────────────────────────><
          └─DISTINCT──┘
```

The schema is SYSIBM.

The AVG function returns the average of a set of numbers.

The argument values must be numbers and their sum must be within the range of the data type of the result. The result can be null.

The data type of the result is the same as the data type of the argument values, except that:
- The result is a large integer if the argument values are small integers.
- The result is double-precision floating point if the argument values are single-precision floating point.

If the data type of the argument values is decimal with precision $p$ and scale $s$, the precision of the result is 31 and the scale is 31-$p$+$s$.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, duplicate values are eliminated.

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.

If the type of the result is integer, the fractional part of the average is lost.

Examples:
- 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.
- 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

## CORRELATION

```
►►──┬─CORRELATION─┬──(─expression1─,─expression2─)──────────────────────►◄
    └─CORR────────┘
```
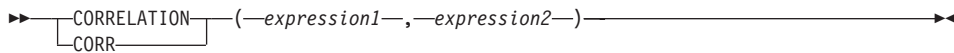
The schema is SYSIBM.

The CORRELATION function returns the coefficient of correlation of a set of number pairs.

The argument values must be numbers.

The data type of the result is double-precision floating point. The result can be null. When not null, the result is between 0 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.

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

```
►►──COUNT──(──┬──────ALL──────┬──expression──)────────────────────────►◄
              │                │
              └──DISTINCT──────┘
              │
              *
```
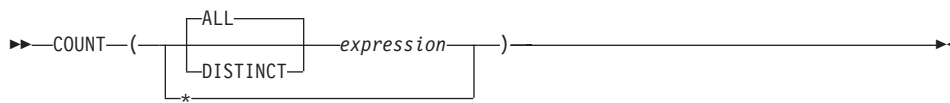
The schema is SYSIBM.

The COUNT function returns the number of rows or values in a set of rows or values.

If DISTINCT is used, the resulting expression must not have a length greater than 254 for a character column or 127 for a graphic column.

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:
- 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.
- 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.

## COUNT

```
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

```
►►──COUNT_BIG──(──┬──────ALL──────┬──expression──┬──)────────────────────►◄
                  │   └─DISTINCT─┘              │
                  └────────*───────────────────┘
```
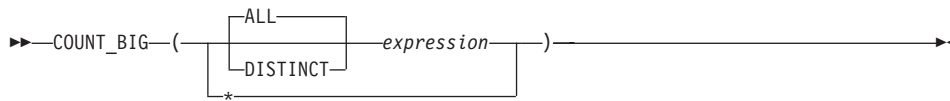
The schema is SYSIBM.

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.

If DISTINCT is used, the resulting expression must not have a length greater than 254 for a character column or 127 for a graphic column.

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:
- 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.
- 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.

## COUNT_BIG

```
CREATE FUNCTION RICK.COUNT() RETURNS DECIMAL(15,0)
      SOURCE SYSIBM.COUNT_BIG();
SET CURRENT FUNCTION 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

```
►►──┬─COVARIANCE─┬─(─expression1─,─expression2─)──────────────────────►◄
    └─COVAR──────┘
```
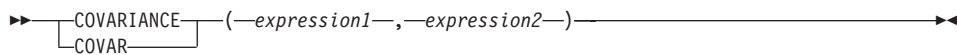
The schema is SYSIBM.

The COVARIANCE function returns the (population) covariance of a set of number pairs.

The argument values must be numbers.

The data type of the result is double-precision floating point. 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 result is equivalent to the following:

1. Let avgexp1 be the result of AVG(*expression1*) and let avgexp2 be the result of AVG(*expression2*).
2. The result of COVARIANCE(*expression1*, *expression2*) is AVG( (*expression1* - avgexp1) * (*expression2* - avgexp2 )

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 COVARNCE (double precision floating point) to the covariance between salary and bonus for those employees in department (WORKDEPT) 'A00'.

```
SELECT COVARIANCE(SALARY, BONUS)
  INTO :COVARNCE
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00'
```

COVARNCE is set to approximately 1.68888888888889E+006 when using the sample table.

## GROUPING

```
►►──GROUPING──(─expression─)──────────────────────────────────────────◄
```

The schema is SYSIBM.

Used in conjunction with grouping-sets and super-groups (see "group-by-clause" on page 382 for details), the GROUPING function returns a value which 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 argument can be of any type, but must be an item of a GROUP BY clause.

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 above.

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.

# MAX

## MAX

```
►►──MAX──(──┬──ALL──────┬──expression──)────────────────────────◄◄
            └──DISTINCT──┘
```

The schema is SYSIBM.

The MAX function returns the maximum value in a set of values.

The argument values can be of any built-in type other than a long string or DATALINK.

If DISTINCT is used, the resulting expression must not have a length greater than 254 for a character column or 127 for a graphic column.

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.

Examples:
- 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.
- 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.

- 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.

## MIN

```
►►──MIN──(──┬──ALL────┬──expression──)────────────────────────────►◄
            └─DISTINCT─┘
```
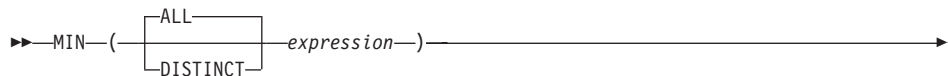
The schema is SYSIBM.

The MIN function returns the minimum value in a set of values.

The argument values can be of any built-in type other than a long string or DATALINK.

If DISTINCT is used, the resulting expression must not have a length greater than 254 for a character column or 127 for a graphic column.

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.

Examples:
- 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.
- 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.

## REGRESSION Functions

```
►►──┬─REGR_AVGX─────────┬───(──expression1──,──expression2──)──────────────────►◄
    ├─REGR_AVGY─────────┤
    ├─REGR_COUNT────────┤
    ├─┬─REGR_INTERCEPT─┬┤
    │ └─REGR_ICPT──────┘│
    ├─REGR_R2───────────┤
    ├─REGR_SLOPE────────┤
    ├─REGR_SXX──────────┤
    ├─REGR_SXY──────────┤
    └─REGR_SYY──────────┘
```
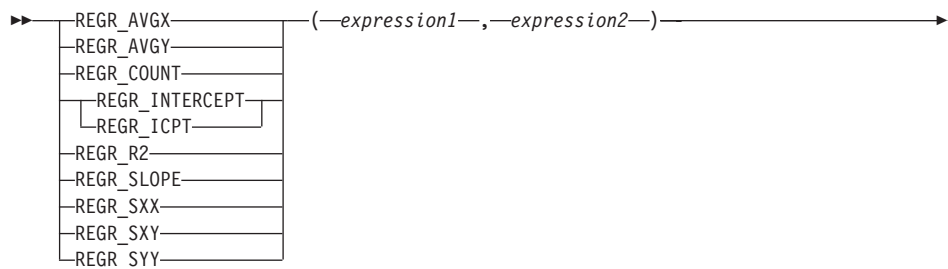
The schema is SYSIBM.

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 first element of each pair (*expression1*) is interpreted as a value of the dependent variable (i.e., a ″y value″). The second element of each pair (*expression2* ) is interpreted as a value of the independent variable (i.e., an ″x value″).

The function REGR_COUNT returns the number of non-null number pairs used to fit the regression line (see below).

The function REGR_INTERCEPT (the short form is REGR_ICPT) returns the y-intercept of the regression line (″b″ in the above equation)

The function REGR_R2 returns the coefficient of determination (also called ″R-squared″ or ″goodness-of-fit″) for the regression.

The function REGR_SLOPE returns the slope of the line (the parameter ″a″ in the above equation).

The functions REGR_AVGX, REGR_AVGY, REGR_SXX, REGR_SYY, and REGR_SXY 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 (see below).

The argument values must be numbers.

The data type of the result of REGR_COUNT is integer. For the remaining functions, the data type of the result is double-precision floating point. 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 above. 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 above functions. For example:

**Adjusted R2**
    1 - ( (1 - REGR_R2) * ((REGR_COUNT - 1) / (REGR_COUNT - 2)) )

## REGRESSION Functions

**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
```

## REGRESSION Functions

```
CNT:  3
AVGSAL:  +4.28333333333333E+004
AVGBONUS:  +8.33333333333333E+002
SXX:  +2.96291666666667E+008
SYY:  +8.66666666666667E+004
SXY:  +5.06666666666667E+006
```

## STDDEV

```
►►──STDDEV──(──┬──ALL──────┬──expression──)──────────────────────────►◄
               └──DISTINCT──┘
```
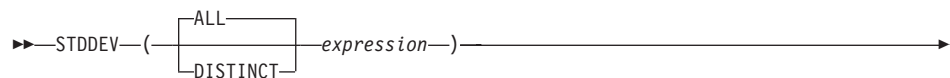
The schema is SYSIBM.

The STDDEV function returns the standard deviation of a set of numbers.

The argument values must be numbers.

The data type of 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, duplicate values are eliminated.

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 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 DEV (double precision floating point) to the standard deviation of the salaries for those employees in department (WORKDEPT) 'A00'.

```
SELECT STDDEV(SALARY)
  INTO :DEV
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00'
```

Results in DEV being set to approximately 9938.00 when using the sample table.

## SUM

```
►►──SUM──(──┬─ALL──────┬──expression──)──────────────────────────────────►◄
            └─DISTINCT─┘
```
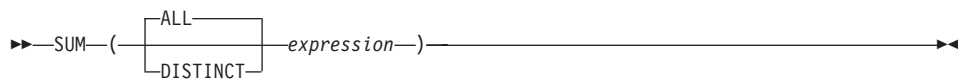
The schema is SYSIBM.

The SUM function returns the sum of a set of numbers.

The argument values must be numbers (built-in types only) and their sum must be within the range of the data type of the result.

The data type of the result is the same as the data type of the argument values except that:
* The result is a large integer if the argument values are small integers.
* The result is double-precision floating point if the argument values are single-precision floating point.

If the data type of the argument values is decimal, the precision of the result is 31 and the scale is the same as the scale of the argument values. 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, duplicate values are also eliminated.

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.

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**

## VARIANCE

```
►►──┬─VARIANCE─┬──(──┬────ALL────┬──expression──)──────────────►◄
    └─VAR──────┘     └─DISTINCT──┘
```
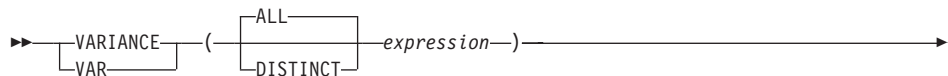
The schema is SYSIBM.

The VARIANCE function returns the variance of a set of numbers.

The argument values must be numbers.

The data type of 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, duplicate values are eliminated.

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 order in which the values are added 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 VARNCE (double precision floating point) to the variance of the salaries for those employees in department (WORKDEPT) 'A00'.

  ```
  SELECT VARIANCE(SALARY)
    INTO :VARNCE
    FROM EMPLOYEE
    WHERE WORKDEPT = 'A00'
  ```

  Results in VARNCE being set to approximately 98763888.88 when using the sample table.

## Scalar Functions

A scalar function can be used wherever an expression can be used. However, the restrictions that apply to the use of expressions and column functions also apply when an expression or column function is used within a scalar function. For example, the argument of a scalar function can be a column function only if a column function is allowed in the context in which the scalar function is used.

The restrictions on the use of column functions do not apply to scalar functions because a scalar function is applied to a single value rather than a set of values.

*Example:* 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'
```

The scalar functions that follow may be qualified with the schema name (for example, SYSIBM.CHAR(123)).

## ABS or ABSVAL

```
►►──┬─ABS─────┬──(─expression─)──────────────────────────────►◄
    └─ABSVAL─┘
```
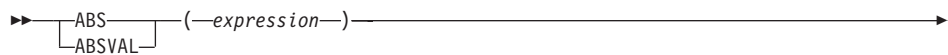
The schema is SYSFUN.

Returns the absolute value of the argument.

The argument can be of any built-in numeric data type. If it is of type DECIMAL or REAL, it is converted to a double-precision floating-point number for processing by the function.

The result of the function is:
- SMALLINT if the argument is SMALLINT
- INTEGER if the argument is INTEGER
- BIGINT if the argument is BIGINT
- DOUBLE if the argument is DOUBLE, DECIMAL or REAL[35] .

The result can be null; if the argument is null, the result is the null value.

---

35. Also returns DOUBLE when the argument is the smallest value of BIGINT, −9 223 372 036 854 775 808.

## ACOS

►►──ACOS─(─*expression*─)──────────────────────────────────────►◄

The schema is SYSFUN.

Returns the arccosine of the argument as an angle expressed in radians.

The argument can be of any built-in numeric data type. It 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 is null, the result is the null value.

# ASCII

## ASCII

```
►►──ASCII──(──expression──)───────────────────────────────►◄
```

The schema is SYSFUN.

Returns the ASCII code value of the leftmost character of the argument as an integer.

The argument can be of any built-in character string type. For a VARCHAR the maximum length is 4 000 bytes and for a CLOB the maximum length is 1 048 576 bytes. LONG VARCHAR is converted to CLOB for processing by the function.

The result of the function is always INTEGER.

The result can be null; if the argument is null, the result is the null value.

## ASIN

►►——ASIN—(—*expression*—)——————————————————————►◄

The schema is SYSFUN.

Returns the arcsine on the argument as an angle expressed in radians.

The argument can be of any built-in numeric type. It 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 is null, the result is the null value.

## ATAN

```
►►──ATAN──(──expression──)──────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the arctangent of the argument as an angle expressed in radians.

The argument can be of any built-in numeric data type. It 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 is null, the result is the null value.

## ATAN2

```
►►──ATAN2──(──expression──,──expression──)──────────────────────────►◄
```

The schema is SYSFUN.

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 first and the second arguments can be of any built-in numeric data type. Both are 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 any argument is null, the result is the null value.

# BIGINT

## BIGINT

```
►►──BIGINT──(──┬──numeric-expression────┬──)──────────────────────►◄
               └──character-expression──┘
```
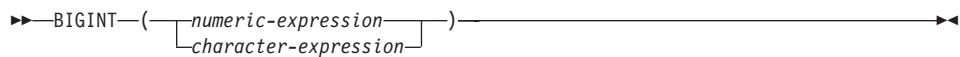
The schema is SYSIBM.

The BIGINT function returns a 64 bit integer representation of a number or character string in the form of an integer constant.

*numeric-expression*
> An expression that returns a value of any built-in numeric data type.
>
> If the argument is a *numeric-expression*, the result is the same number that would occur if the argument were assigned to a big integer column or variable. If the whole part of the argument is not within the range of integers, an error occurs. The decimal part of the argument is truncated if present.

*character-expression*
> An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant (SQLSTATE 22018). The character string cannot be a long string.
>
> If the argument is a *character-expression*, the result is the same number that would occur if the corresponding integer constant were assigned to a big integer column or variable.

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.

Examples:
- 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
  ```
- Using the EMPLOYEE table, select the EMPNO column in big integer form for further processing in the application.
  ```
  SELECT BIGINT(EMPNO) FROM EMPLOYEE
  ```

**BLOB**

```
►►──BLOB──(──string-expression─────────────────)──────────────────►◄
                              └─,─integer─┘
```

The schema is SYSIBM.

The BLOB function returns a BLOB representation of a string of any type.

*string-expression*
>   A *string-expression* whose value can be a character string, graphic string, or a binary string.

*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.

Examples
- 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  || ': ') || TOPOGRAPHIC_MAP
  FROM ONTARIO_SERIES_4
  WHERE TOPOGRAPIC_MAP LIKE BLOB('%Pellow Island%')
```

**CEILING or CEIL**

## CEILING or CEIL

```
►►──┬─CEILING─┬──(──expression──)──────────────────────────►◄
    └─CEIL────┘
```

The schema is SYSFUN.

Returns the smallest integer value greater than or equal to the argument.

The argument can be of any built-in numeric type. If the argument is of type DECIMAL or REAL, it is converted to a double-precision floating-point number for processing by the function. If the argument is of type SMALLINT or INTEGER, the argument value 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
- DOUBLE if the argument is DECIMAL, REAL or DOUBLE. Decimal values with more than 15 digits to the left of the decimal will not return the desired integer value due to loss of precision in the conversion to DOUBLE.

The result can be null; if the argument is null, the result is the null value.

## CHAR

**Datetime to Character:**
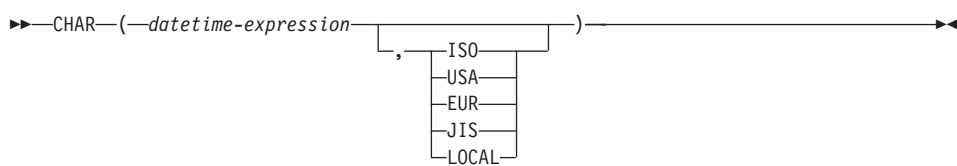
```
►►──CHAR──(──datetime-expression──────┬──────────────┬──)──────────────────────►◄
                                      └──,──┬──ISO───┬─┘
                                            ├──USA───┤
                                            ├──EUR───┤
                                            ├──JIS───┤
                                            └──LOCAL─┘
```
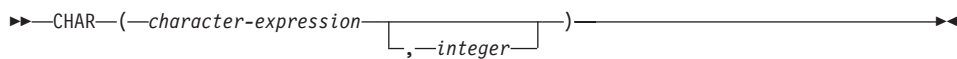
**Character to Character:**

```
►►──CHAR──(──character-expression──┬─────────────┬──)──────────────────────────►◄
                                   └──,──integer──┘
```

**Integer to Character:**

```
►►──CHAR──(──integer-expression──)─────────────────────────────────────────────►◄
```

**Decimal to Character:**

```
►►──CHAR──(──decimal-expression──┬─────────────────────────┬──)─────────────────►◄
                                 └──,──decimal-character────┘
```

**Floating-point to Character:**

```
►►──CHAR──(──floating-point-expression──)───────────────────────────────────────►◄
```
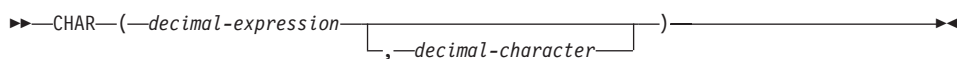
The schema is SYSIBM. However, the schema for CHAR(floating-point-expression) is SYSFUN.

The CHAR function returns a character-string representation of a:
- Datetime value if the first argument is a date, time or timestamp
- Character string value if the first argument is any type of character string
- Integer number if the first argument is a SMALLINT, INTEGER or BIGINT
- Decimal number if the first argument is a decimal number
- Double-precision floating-point number if the first argument is a DOUBLE or REAL.

# CHAR

The result of the function is a fixed-length character 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.

**Datetime to Character**

*datetime-expression*
> An expression that is one of the following three 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 occurs 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 occurs if the second argument is specified and is not a valid value (SQLSTATE 42703).

> **timestamp**
> > The second argument is not applicable and must not be specified. SQLSTATE 42815 The result is the character string representation of the timestamp. The length of the result is 26.

> The code page of the string is the code page of the database at the application server.

**Character to Character**

*character-expression*
> An expression that returns a value that is CHAR, VARCHAR, LONG VARCHAR, or CLOB data type.

*integer*
> the length attribute for the resulting fixed length character string. The value must be between 0 and 254.

> If the length of the character-expression is less than the length attribute of the result, the result is padded with blanks up to the length of the result. If the length of the character-expression is greater than the length attribute of the result, truncation is performed. A warning is returned (SQLSTATE 01004) unless the truncated characters were all blanks and the character-expression was not a long string (LONG VARCHAR or CLOB).

**Integer to Character**

*integer-expression*
>    An expression that returns a value that is an integer data type
>    (either SMALLINT, INTEGER or BIGINT).

The result is the character string representation of the argument in the
form of an SQL integer constant. The result consists of n characters
that are the significant digits that represent the value of the argument
with a preceding minus sign if the argument is negative. It is left
justified.

- If the first argument is a small integer:

  The length of the result is 6. If the number of characters in the
  result is less than 6, then the result is padded on the right with
  blanks to length 6.

- If the first argument is a large integer:

  The length of the result is 11. If the number of characters in the
  result is less than 11, then the result is padded on the right with
  blanks to length 11.

- If the first argument is a big integer:

  The length of the result is 20. If the number of characters in the
  result is less than 20, then the result is padded on the right with
  blanks to length 20.

The code page of the string is the code page of the database at the
application server.

**Decimal to Character**

*decimal-expression*
>    An expression that returns a value that is a decimal data type. If a
>    different precision and scale is desired, 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
>    cannot be a digit, plus ('+'), minus ('-') or blank. (SQLSTATE
>    42815). The default is the period ('.') character

The result is the fixed-length character-string representation of the
argument. The result includes a decimal character and $p$ digits, where
$p$ is the precision of the *decimal-expression* with a preceding minus sign
if the argument is negative. The length of the result is $2+p$, where $p$ is
the precision of the *decimal-expression*. This means that a positive value
will always include one trailing blank.

**CHAR**

The code page of the string is the code page of the database at the application server.

**Floating-point to Character**

*floating-point-expression*
An expression that returns a value that is a floating-point data type (DOUBLE or REAL).

The result is the fixed-length character-string representation of the argument in the form of a floating-point constant. The length of the result is 24. If the argument is negative, the first character of the result is a minus sign. Otherwise, the first character is a digit. If the argument value is zero, the result is 0E0. Otherwise, the result includes the smallest number of characters that can represent the value of the argument such that the mantissa consists of a single digit other than zero followed by a period and a sequence of digits. If the number of characters in the result is less than 24, then the result is padded on the right with blanks to length 24.

The code page of the string is the code page of the database at the application server.

Examples:
- Assume the column PRSTDATE has an internal value equivalent to 1988-12-25.

  **CHAR**(PRSTDATE, USA)

  Results in the value '12/25/1988'.
- Assume the column STARTING has an internal value equivalent to 17.12.30, the host variable HOUR_DUR (decimal(6,0)) is a time duration with a value of 050000. (that is, 5 hours).

  **CHAR**(STARTING, USA)

  Results in the value '5:12 PM'.

  **CHAR**(STARTING + :HOUR_DUR, USA)

  Results in the value '10:12 PM'.
- Assume the column RECEIVED (timestamp) has an internal value equivalent to the combination of the PRSTDATE and STARTING columns.

  **CHAR**(RECEIVED)

  Results in the value '1988-12-25-17.12.30.000000'.

- Use the CHAR function to make the type fixed length character and reduce the length of the displayed results to 10 characters for the LASTNAME column (defined as VARCHAR(15)) of the EMPLOYEE table.

  **SELECT CHAR(**LASTNAME,10**) FROM** EMPLOYEE

  For rows having a LASTNAME with a length greater than 10 characters (excluding trailing blanks), a warning that the value is truncated is returned.

- Use the CHAR function to return the values for EDLEVEL (defined as smallint) as a fixed length character string.

  **SELECT CHAR(**EDLEVEL**) FROM** EMPLOYEE

  An EDLEVEL of 18 would be returned as the CHAR(6) value '18 ' (18 followed by four blanks).

- Assume that STAFF has a SALARY column defined as decimal with precision of 9 and scale of 2. The current value is 18357.50 and it is to be displayed with a comma as the decimal character (18357,50).

  **CHAR(**SALARY, ','**)**

  returns the value '00018357,50 '.

- Assume the same SALARY column subtracted from 20000.25 is to be displayed with the default decimal character.

  **CHAR(**20000.25 - SALARY**)**

  returns the value '-0001642.75'.

- Assume a host variable, SEASONS_TICKETS, has an integer data type and a 10000 value.

  **CHAR(**DECIMAL(:SEASONS_TICKETS,7,2)**)**

  Results in the character value '10000.00 '.

- Assume a host variable, DOUBLE_NUM has a double data type and a value of -987.654321E-35.

  **CHAR(**:DOUBLE_NUM**)**

  Results in the character value of '-9.87654321E-33          '. Since the result data type is CHAR(24), there are 9 trailing blanks in the result.

**CHR**

## CHR

►►──CHR──(──*expression*──)────────────────────────────►◄

The schema is SYSFUN.

Returns the character that has the ASCII code value specified by the argument.

The argument can be either INTEGER or SMALLINT. The value of the argument should be between 0 and 255; otherwise, the return value is null.

The result of the function is CHAR(1). The result can be null; if the argument is null, the result is the null value.

## CLOB

```
►►──CLOB──(──character-string-expression──┬──────────────┬──)──────────────────►◄
                                          └──,──integer──┘
```

The schema is SYSIBM.

The CLOB function returns a CLOB representation of a character string type.

*character-string-expression*
> An *expression* that returns a value that is a character string.

*integer*
> An integer value specifying the length attribute of the resulting CLOB data type. The value must be between 0 and 2 147 483 647. If *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. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

## COALESCE

### COALESCE

```
         (1)
►►──COALESCE────────(──expression──┬──,──expression──┬──)──────────────►◄
```
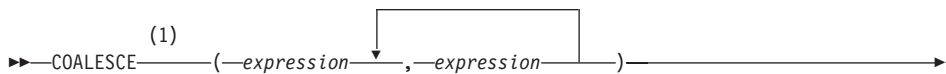
**Notes:**

**1**    VALUE is a synonym for COALESCE.

The schema is SYSIBM.

COALESCE returns the first argument that is not null.

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. The result can be null only if all the arguments can be null, and the result is null only if all the arguments are null. The selected argument is converted, if necessary, to the attributes of the result.

The arguments must be compatible. See "Rules for Result Data Types" on page 105 for what data types are compatible and the attributes of the result. They can be of either a built-in or user-defined data type. [36]

Examples:
- 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
  ```
- 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
  ```

---

36. This function may not be used as a source function when creating a user-defined function. Since it accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.

## CONCAT

```
         (1)
►►─CONCAT────────(─expression1─,─expression2─)──────────────────────►◄
```

**Notes:**

**1**    | | may be used as a synonym for CONCAT.

The schema is SYSIBM.

Returns the concatenation of two string arguments. The two arguments must be compatible types.

The result of the function is a string. Its length is sum of the lengths of the two arguments. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

See "With the Concatenation Operator" on page 147 for more information.

## COS

**COS**

►►──COS──(──*expression*──)──────────────────────────────►◄

The schema is SYSFUN.

Returns the cosine of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric type. It 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 is null, the result is the null value.

## COT

```
►►──COT──(──expression──)──────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the cotangent of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric type. It 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 is null, the result is the null value.

## DATE

```
►►──DATE──(──expression──)──────────────────────────────►◄
```

The schema is SYSIBM.

The DATE function returns a date from a value.

The argument must be a date, timestamp, a positive number less than or equal to 3 652 059, a valid character string representation of a date or timestamp, or a character string of length 7 that is neither a CLOB nor a LONG VARCHAR.

If the argument is a character 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.

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 character string with a length of 7:
  - The result is the date represented by the character string.

Examples:
- Assume that the column RECEIVED (timestamp) has an internal value equivalent to '1988-12-25-17.12.30.000000'.

  **DATE**(RECEIVED)

  Results in an internal representation of '1988-12-25'.
- This example results in an internal representation of '1988-12-25'.

  **DATE**('1988-12-25')
- This example results in an internal representation of '1988-12-25'.

  **DATE**('25.12.1988')
- This example results in an internal representation of '0001-02-04'.

DATE

**DATE**(35)

## DAY

```
►►──DAY──(──expression──)──────────────────────────────────────►◄
```

The schema is SYSIBM.

The DAY function returns the day part of a value.

The argument must be a date, timestamp, date duration, timestamp duration, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

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:
- 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.
- Assume that the column DATE1 (date) has an internal value equivalent to 2000-03-15 and the column DATE2 (date) has an internal value equivalent to 1999-12-31.

```
DAY(DATE1 - DATE2)
```

Results in the value 15.

## DAYNAME

►►—DAYNAME—(—*expression*—)————————————————————————►◄

The schema is SYSFUN.

Returns a mixed case character string containing the name of the day (e.g. Friday) for the day portion of the argument based on the locale when the database was started.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

The result of the function is VARCHAR(100). The result can be null; if the argument is null, the result is the null value.

## DAYOFWEEK

►►—DAYOFWEEK—(—*expression*—)————————————————►◄

The schema is SYSFUN.

Returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Sunday.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

## DAYOFYEAR

►►──DAYOFYEAR──(──*expression*──)────────────────────────────────►◄

The schema is SYSFUN.

Returns the day of the year in the argument as an integer value in the range 1-366.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

## DAYS

```
▶▶──DAYS──(──expression──)──────────────────────────────────────────────▶◀
```

The schema is SYSIBM.

The DAYS function returns an integer representation of a date.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

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:
- 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 when using the sample table.
- 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.

## DBCLOB

```
►►──DBCLOB──(──graphic-expression─────────────────────)──────────────────────►◄
                                   └─,──integer──┘
```

The schema is SYSIBM.

The DBCLOB function returns a DBCLOB representation of a graphic string type.

*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 *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 DBCLOB. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

## DECIMAL

**Numeric to Decimal:**

```
►►──┬─DECIMAL─┬──(─numeric-expression──────────────────────────────────►
    └─DEC─────┘

►──┬──────────────────────────────────────────┬──)─────────────────►◄
   └─,─precision-integer─┬────────────────────┘
                        └─,─scale-integer──┘
```

**Character to Decimal:**

```
►►──┬─DECIMAL─┬──(─character-expression────────────────────────────────►
    └─DEC─────┘

►──┬─────────────────────────────────────────────────────────────┬──)─►◄
   └─,─precision-integer─┬──────────────────────────────────────┐
                        └─,─scale-integer─┬─────────────────────┘
                                         └─,─decimal-character──┘
```
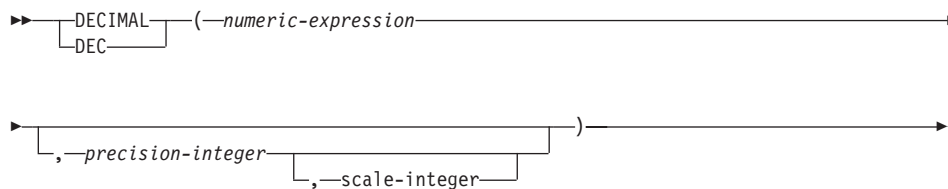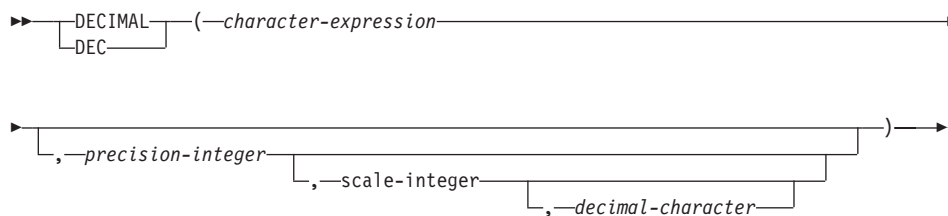
The schema is SYSIBM.

The DECIMAL function returns a decimal representation of
- A number
- A character string representation of a decimal number
- A character string representation of a integer number.

The result of the function is a decimal number with precision of $p$ and scale of $s$, where $p$ and $s$ are the second and third arguments. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

### Numeric to Decimal

*numeric-expression*
> An expression that returns a value of any numeric data type.

*precision-integer*
> An integer constant with a value in the range of 1 to 31.
>
> The default for the *precision-integer* depends on the data type of the *numeric-expression*:
> - 15 for floating-point and decimal
> - 19 for big integer

- 11 for large integer
- 5 for small integer.

*scale-integer*
> An integer constant in the range of 0 to the *precision-integer* value. The default 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 *p* and a scale of *s*, where *p* and *s* are the second and third arguments. An error occurs if the number of significant decimal digits required to represent the whole part of the number is greater than *p-s*.

## Character to Decimal

*character-expression*
> An *expression* that returns a value that is a character string with a length not greater than the maximum length of a character constant (4 000 bytes). It cannot have a CLOB or LONG VARCHAR data type. Leading and trailing blanks are eliminated from the string. The resulting substring must conform to the rules for forming an SQL integer or decimal constant (SQLSTATE 22018).
>
> The *character-expression* is converted to the database code page if required to match the code page of the constant *decimal-character*.

*precision-integer*
> 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-integer*
> An integer constant with a value in the range 0 to *precision-integer* that specifies the scale of the result. If not specified, the default is 0.

*decimal-character*
> Specifies the single byte character constant that is 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 can appear at most once in *character-expression* (SQLSTATE 42815).

The result is a decimal number with precision *p* and scale *s* where *p* and *s* are the second and third arguments. Digits are truncated from the end if the number of digits right of the decimal character is greater than the scale *s*. An error occurs if the number of significant digits left of the decimal character (the whole part of the number) in *character-expression* is greater than *p-s* (SQLSTATE 22003). The default

## DECIMAL

decimal character is not valid in the substring if the *decimal-character* argument is specified (SQLSTATE 22018).

Examples:

- 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
  ```

- 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).

  ```
  SELECTPRSTDATE + DECIMAL(:PERIOD,8)
    FROM PROJECT
  ```

- 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.

- 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.

## DEGREES

```
►►──DEGREES──(──expression──)────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the number of degrees converted from the argument expressed in radians.

The argument can be of any built-in numeric type. It is converted to 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 is null, the result is the null value.

## DEREF

►►──DEREF──(──*expression*──)──────────────────────────────────►◄

The schema is SYSIBM.

The DEREF function returns an instance of the target type of the argument.

The argument can be any 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).

This function can only be used on the left side of the TYPE predicate or in the argument of the TYPE_ID, TYPE_NAME and TYPE_SCHEMA functions.

Examples:

See Examples section in "TYPE_NAME" on page 353.

## DIFFERENCE

►►──DIFFERENCE──(──*expression*──,──*expression*──)──────────────────────────►◄

The schema is SYSFUN.

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 arguments can be character strings that are either CHAR or VARCHAR up to 4 000 bytes.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

Example:
```
VALUES (DIFFERENCE('CONSTRAINT','CONSTANT'),SOUNDEX('CONSTRAINT'),
        SOUNDEX('CONSTANT')),
        (DIFFERENCE('CONSTRAINT','CONTRITE'),SOUNDEX('CONSTRAINT'),
        SOUNDEX('CONTRITE'))
```

This example returns the following.
```
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

```
►►──DIGITS──(──expression──)──────────────────────────────────────►◄
```

The schema is SYSIBM.

The DIGITS function returns a character-string representation of a number.

The argument must be an expression that returns a value of type SMALLINT, INTEGER, BIGINT or DECIMAL.

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:
- 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
  ```
- 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.

## DLCOMMENT

```
►►──DLCOMMENT──(──datalink-expression──)──────────────────────────────────►◄
```

The schema is SYSIBM.

The DLCOMMENT function returns the comment value, if it exists, from a DATALINK value.

The argument must be an expression that results in a value with data type of DATALINK.

The result of the function is VARCHAR(254). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example:
- Prepare a statement to select the date, the description and the comment from the link to the ARTICLES column from the HOCKEY_GOALS table. The rows to be selected are those for goals scored by either of the Richard brothers (Maurice or Henri).

```
stmtvar = "SELECT DATE_OF_GOAL, DESCRIPTION, DLCOMMENT(ARTICLES)
             FROM HOCKEY_GOALS
             WHERE BY_PLAYER = 'Maurice Richard'
             OR BY_PLAYER = 'Henri Richard' ";
EXEC SQL PREPARE HOCKEY_STMT FROM :stmtvar;
```

- Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

```
DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','A comment')
```

then the following function operating on that value:

```
DLCOMMENT(COLA)
```

will return the value:

```
A comment
```

**DLLINKTYPE**

## DLLINKTYPE

```
►►──DLLINKTYPE──(──datalink-expression──)──────────────────────►◄
```

The schema is SYSIBM.

The DLLINKTYPE function returns the linktype value from a DATALINK value.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(4). 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 DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

    **DLVALUE**('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')

    then the following function operating on that value:

    **DLLINKTYPE**(COLA)

    will return the value:

    URL

## DLURLCOMPLETE

►►──DLURLCOMPLETE─(─*datalink-expression*─)──────────────────────────►◄

The schema is SYSIBM.

The DLURLCOMPLETE function returns the complete URL value from a DATALINK value with a link type of URL. The value is the same as what would be returned by the concatenation of DLURLSCHEME with ':// ', then DLURLSERVER, then '/' and then DLURLPATH. When appropriate, the value includes a file access token.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(254). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DATALINK value only includes the comment the result returned is a zero length string.

Example:
- Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:
    **DLVALUE**('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')

  then the following function operating on that value:
    **DLURLCOMPLETE**(COLA)

  will return the value:
    HTTP://DLFS.ALMADEN.IBM.COM/x/y/***************;a.b

(where *************** represents the access token)

## DLURLPATH

▶▶──DLURLPATH──(──*datalink-expression*──)────────────────────────────▶◀

The schema is SYSIBM.

The DLURLPATH function returns the path and file name necessary to access a file within a given server from a DATALINK value with a linktype of URL. When appropriate, the value includes a file access token.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(254). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DATALINK value only includes the comment the result returned is a zero length string.

Example:
• Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

    **DLVALUE**('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')

then the following function operating on that value:

    **DLURLPATH**(COLA)

will return the value:

    /x/y/****************;a.b

(where *************** represents the access token)

## DLURLPATHONLY

▶▶──DLURLPATHONLY──(──*datalink-expression*──)──────────────────────────────▶◀

The schema is SYSIBM.

The DLURLPATHONLY function returns the path and file name necessary to access a file within a given server from a DATALINK value with a linktype of URL. The value returned NEVER includes a file access token.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(254). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DATALINK value only includes the comment the result returned is a zero length string.

Example:
- Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

    **DLVALUE**('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')

  then the following function operating on that value:

    **DLURLPATHONLY**(COLA)

  will return the value:

    /x/y/a.b

## DLURLSCHEME

►►──DLURLSCHEME──(──*datalink-expression*──)──────────────────────────◄◄

The schema is SYSIBM.

The DLURLSCHEME function returns the scheme from a DATALINK value with a linktype of URL. The value will always be in upper case.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(20). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DATALINK value only includes the comment the result returned is a zero length string.

Example:
- Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

      **DLVALUE**('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')

  then the following function operating on that value:

      **DLURLSCHEME**(COLA)

  will return the value:

      HTTP

## DLURLSERVER

►►──DLURLSERVER──(──*datalink-expression*──)──────────────────────────────►◄

The schema is SYSIBM.

The DLURLSERVER function returns the file server from a DATALINK value with a linktype of URL. The value will always be in upper case.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(254). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DATALINK value only includes the comment the result returned is a zero length string.

Example:
- Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

    **DLVALUE**('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')
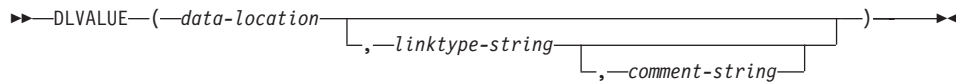
  then the following function operating on that value:

    **DLURLSERVER**(COLA)

  will return the value:

    DLFS.ALMADEN.IBM.COM

**DLVALUE**

## DLVALUE

►►──DLVALUE──(──*data-location*─────────────────────────────────)──►◄
            └─,──*linktype-string*──────────────────┘
                         └─,──*comment-string*──┘

The schema is SYSIBM.

The DLVALUE function returns a DATALINK value. When the function is on
the right hand side of a SET clause in an UPDATE statement or is in a
VALUES clause in an INSERT statement, it usually also creates a link to a file.
However, if only a comment is specified (in which case the data-location is a
zero-length string), the DATALINK value is created with empty linkage
attributes so there is no file link.

*data-location*
>    If the link type is URL, then this is an expression that yields a varying
>    length character string containing a complete URL value. If the expression
>    is not an empty string and does not include the URL scheme and URL
>    server, then the defaults of scheme "FILE" and the server name of the
>    database server are included in the complete URL value.

*linktype-string*
>    An optional VARCHAR expression that specifies the link type of the
>    DATALINK value. The only valid value is 'URL' (SQLSTATE 428D1).

*comment-string*
>    An optional VARCHAR(254) value that provides a comment or additional
>    location information.

The result of the function is a DATALINK value. If any argument of the
DLVALUE function can be null, the result can be null; If the *data-location* is
null, the result is the null value.

When defining a DATALINK value using this function, consider the
maximum length of the target of the value. For example, if a column is
defined as DATALINK(200), then the maximum length of the *data-location* plus
the *comment* is 200 bytes.

Example:
- Insert a row into the table. The URL values for the first two links are
  contained in the variables named url_article and url_snapshot. The variable
  named url_snapshot_comment contains a comment to accompany the
  snapshot link. There is, as yet, no link for the movie, only a comment in the
  variable named url_movie_comment.

```
EXEC SQL  INSERT INTO HOCKEY_GOALS
             VALUES('Maurice Richard',
                    'Montreal Canadien',
                    '?',
                    'Boston Bruins,
                    '1952-04-24',
                    'Winning goal in game 7 of Stanley Cup final',
                     DLVALUE(:url_article),
                     DLVALUE(:url_snapshot, 'URL', :url_snapshot_comment),
                     DLVALUE('', 'URL', :url_movie_comment) );
```
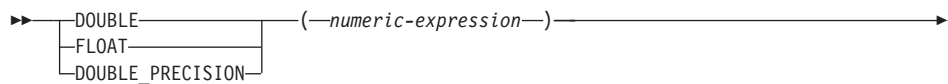
## DOUBLE

**Numeric to Double:**

```
►►──┬─DOUBLE──────────┬──(──numeric-expression──)──────────────────►◄
    ├─FLOAT───────────┤
    └─DOUBLE_PRECISION─┘
```

**Character String to Double:**

```
►►──DOUBLE──(──string-expression──)──────────────────────────────►◄
```

The schema is SYSIBM. However, the schema for DOUBLE(*string-expression*) is SYSFUN.

The DOUBLE function returns a floating-point number corresponding to a:
- number if the argument is a numeric expression
- character string representation of a number if the argument is a string expression.

**Numeric to Double**

*numeric-expression*

>The argument is an expression that returns a value of any built-in numeric data type.
>
>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.
>
>The result is the same number that would occur if the argument were assigned to a double-precision floating-point column or variable.

**Character String to Double**

*string-expression*

>The argument can be of type CHAR or VARCHAR in the form of a numeric constant. Leading and trailing blanks in argument are ignored.
>
>The result of the function is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

The result is the same number that would occur if the string was considered a constant and assigned to a double-precision floating-point column or variable.

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
```

## EVENT_MON_STATE

►►─EVENT_MON_STATE─(──*string-expression*─)──────────────────────────────────►◄

The schema is SYSIBM.

The EVENT_MON_STATE function returns the current state of an event monitor.

The argument is a string expression with a resulting type of CHAR or VARCHAR and a value that is the name of an event monitor. If the named event monitor does not exist in the SYSCAT.EVENTMONITORS catalog table, SQLSTATE 42704 will be returned.

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

```
►►──EXP──(──expression──)─────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the exponential function of the argument.

The argument can be of any built-in numeric data type. It is converted to 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 is null, the result is the null value.

## FLOAT

### FLOAT

▶▶──FLOAT──(──*numeric-expression*──)──────────────────────────────▶◀

The schema is SYSIBM.

The FLOAT function returns a floating-point representation of a number.

FLOAT is a synonym for DOUBLE. See "DOUBLE" on page 272 for details.

## FLOOR

►►──FLOOR──(──*expression*──)──────────────────────────────────────►◄

The schema is SYSFUN.

Returns the largest integer value less than or equal to the argument.

The argument can be of any built-in numeric type. If the argument is of type DECIMAL or REAL, it is converted to a double-precision floating-point number for processing by the function. If the argument is of type SMALLINT, INTEGER or BIGINT the argument value 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
- DOUBLE if the argument is DOUBLE, DECIMAL or REAL. Decimal values with more than 15 digits to the left of the decimal will not return the desired integer value due to loss of precision in the conversion to DOUBLE.

The result can be null; if the argument is null, the result is the null value.

# GENERATE_UNIQUE

## GENERATE_UNIQUE

▶▶──GENERATE_UNIQUE──(──)──────────────────────────────────────────────◀◀

The schema is SYSIBM.

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.[37]

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 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 partition number where the function executed so that a table partitioned across multiple 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 or an insert statement with a fullselect.

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.

Examples:
- 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),
  ```

---

37. The system clock is used to generate the internal Universal Time, Coordinated (UTC) timestamp along with the partition number on which the function executes. Adjustments that move the actual system clock backward could result in duplicate values.

```
 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 MODE DB2SQL
SET NEW_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.

## GRAPHIC

### GRAPHIC

```
►►──GRAPHIC──(──graphic-expression──┬──────────────┬──)────────────────────►◄
                                     └─,──integer──┘
```

The schema is SYSIBM.

The GRAPHIC function returns a GRAPHIC representation of a graphic string
type.

*graphic-expression*
    An *expression* that returns a value that is a graphic string.

*integer*
    An integer value specifying the length attribute of the resulting GRAPHIC
    data type. The value must be between 1 and 127. If *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 GRAPHIC. If the argument can be null, the
result can be null; if the argument is null, the result is the null value.

## HEX

```
►►──HEX──(──expression──)──────────────────────────────────────►◄
```

The schema is SYSIBM.

The HEX function returns a hexadecimal representation of a value as a character string.

The argument can be an expression that is a value of any built-in data type with a maximum length of 16 336 bytes.

The result of the function is a character string. 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 database 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 and length of character string arguments.

- Character string
  - Fixed length not greater than 127
    - Result is a character string of fixed length twice the defined length of the argument.
  - Fixed length greater than 127
    - Result is a character string of varying length twice the defined length of the argument.
  - Varying length
    - Result is a character string of varying length with maximum length twice the defined maximum length of the argument.

**HEX**

- Graphic string
  - Fixed length not greater than 63
    - Result is a character string of fixed length four times the defined length of the argument.
- Fixed length greater than 63
  - Result is a character string of varying length four times the defined length of the argument.
- Varying length
  - Result is a character string of varying length with maximum length four times the defined maximum length of the argument.

Examples:

Assume the use of a DB2 for AIX application server for the following examples.

- 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').

- 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-character string '42'.
- Suppose COL_3 is a column with a data type of decimal(6,2) and a value of 40.1. An eight-character string '0004010C' is the result of applying the HEX function to the internal representation of the decimal value, 40.1.

## HOUR

►►──HOUR──(──*expression*──)──────────────────────────────────────────►◄

The schema is SYSIBM.

The HOUR function returns the hour part of a value.

The argument must be a time, timestamp, time duration, timestamp duration or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR.

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 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
```

## INSERT

►►—INSERT—(—*expression1*—,—*expression2*—,—*expression3*—,—*expression4*—)—————►◄

The schema is SYSFUN.

Returns a string where *expression3* bytes have been deleted from *expression1* beginning at *expression2* and where *expression4* has been inserted into *expression1* beginning at *expression2*. If the length of the result string exceeds the maximum for the return type, an error occurs (SQLSTATE 38552).

The first argument is a character string or a binary string type. The second and third arguments must be a numeric value with a data type of SMALLINT or INTEGER. If the first argument is a character string, then the fourth argument must also be a character string. If the first argument is a binary string, then the fourth argument must be a binary string. For a VARCHAR the maximum length is 4 000 bytes and for a CLOB or a binary string the maximum length is 1 048 576 bytes. For the first and fourth arguments, CHAR is converted to VARCHAR and LONG VARCHAR to CLOB(1M), for second and third arguments SMALLINT is converted to INTEGER for processing by the function.

The result is based on the argument types as follows:
- VARCHAR(4000) if both the first and fourth arguments are VARCHAR (not exceeding 4 000 bytes) or CHAR
- CLOB(1M) if either the first or fourth argument is CLOB or LONG VARCHAR
- BLOB(1M) if both first and fourth arguments are BLOB.

The result can be null; if any argument is null, the result is the null value.

Example:
- Delete one character from the word 'DINING' and insert 'VID', both beginning at the third character.
  ```
  VALUES CHAR(INSERT('DINING', 3, 1, 'VID'), 10)
  ```

  This example returns the following:
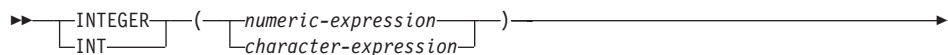  ```
  1
  ----------
  DIVIDING
  ```

As mentioned, the output of the INSERT function is VARCHAR(4000). For the above example the function CHAR has been used to limit the output of

INSERT to 10 bytes. The starting location of a particular string can be found using LOCATE. Refer to "LOCATE" on page 294 for more information.

## INTEGER

```
►►──┬─INTEGER─┬──(──┬─numeric-expression───┬──)──────────────────────►◄
    └─INT─────┘     └─character-expression─┘
```

The schema is SYSIBM.

The INTEGER function returns an integer representation of a number or character string in the form of an integer constant.

*numeric-expression*
> An expression that returns a value of any built-in numeric data type.
>
> If the argument is a *numeric-expression*, the result is the same number that would occur if the argument were assigned to a large integer column or variable. If the whole part of the argument is not within the range of integers, an error occurs. The decimal part of the argument is truncated if present.

*character-expression*
> An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant (SQLSTATE 22018). The character string cannot be a long string.
>
> If the argument is a *character-expression*, the result is the same number that would occur if the corresponding integer constant were assigned to a large integer column or variable.

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.

Examples:
- 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
  ```
- Using the EMPLOYEE table, select the EMPNO column in integer form for further processing in the application.

  ```
  SELECT INTEGER(EMPNO) FROM EMPLOYEE
  ```

## JULIAN_DAY

```
►►—JULIAN_DAY—(—expression—)————————————————————————————►◄
```

The schema is SYSFUN.

Returns an integer value representing the number of days from January 1,4712 B.C. (the start of Julian date calendar) to the date value specified in the argument.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

## LCASE or LOWER

```
►►──┬─LCASE─┬──(──string-expression──)──────────────────────►◄
    └─LOWER─┘
```

The schema is SYSIBM.[38]

The LCASE or LOWER function returns a string in which all the SBCS characters have been converted to lowercase characters (that is, the characters A-Z will be translated to the characters a-z, and characters with diacritical marks will be translated to their lower case equivalents if they exist. For example, in code page 850, É maps to é). Since not all characters are translated, LCASE(UCASE(*string-expression*)) does not necessarily return the same result as LCASE(*string-expression*).

The argument must be an expression whose value is a CHAR or VARCHAR data type.

The result of the function has the same data type 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: Ensure that the characters in the value of column JOB in the EMPLOYEE table are returned in lowercase characters.

```
SELECT LCASE(JOB)
  FROM EMPLOYEE WHERE EMPNO = '000020';
```

The result is the value 'manager'.

---

38. The SYSFUN version of this function continues to be available with support for LONG VARCHAR and CLOB arguments. See "LCASE (SYSFUN schema)" on page 289 for a description.

## LCASE (SYSFUN schema)

►►──LCASE──(──*expression*──)────────────────────────────────────►◄

The schema is SYSFUN.

Returns a string in which all the characters A-Z have been converted to the characters a-z (characters with diacritical marks are not converted). Note that LCASE(UCASE(string)) will therefore not necessarily return the same result as LCASE(string).

The argument can be of any built-in character string 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 the argument is VARCHAR (not exceeding 4 000 bytes) or CHAR
- CLOB(1M) if the argument is CLOB or LONG VARCHAR

The result can be null; if the argument is null, the result is the null value.

## LEFT

►►—LEFT—(—*expression1*—,—*expression2*—)———————————————————►◄

The schema is SYSFUN.

Returns a string consisting of the leftmost *expression2* bytes in *expression1*.

The first argument is a character string or binary string type. For a VARCHAR the maximum length is 4 000 bytes and for a CLOB or a binary string the maximum length is 1 048 576 bytes. The second argument must be of INTEGER or SMALLINT datatype.

The result of the function is:
- VARCHAR(4000) if the argument is VARCHAR (not exceeding 4 000 bytes) or CHAR
- CLOB(1M) if the argument is CLOB or LONG VARCHAR
- BLOB(1M) if the argument is BLOB.

The result can be null; if any argument is null, the result is the null value.

## LENGTH

```
►►──LENGTH──(──expression──)──────────────────────────────────►◄
```

The schema is SYSIBM.

The LENGTH function returns the length of a value.

The argument can be an expression that returns a value of any built-in data type.

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 the length of the argument. The length does not include the null indicator byte of column arguments that allow null values. The length of strings includes blanks but does not include the length control field of varying-length strings. The length of a varying-length string is the actual length, not the maximum length.

The length of a graphic string is the number of DBCS characters. The length of all other values is the number of bytes used to represent the value:
- 2 for small integer
- 4 for large integer
- $(p/2)+1$ for decimal numbers with precision $p$
- 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
- 10 for timestamp

Examples:
- Assume the host variable ADDRESS is a varying length character string with a value of '895 Don Mills Road'.

  **LENGTH**(:ADDRESS)

  Returns the value 18.
- Assume that START_DATE is a column of type DATE.

  **LENGTH**(START_DATE)

  Returns the value 4.

## LENGTH

- This example returns the value 10.

```
LENGTH(CHAR(START_DATE, EUR))
```

**LN**

```
►►──LN──(──expression──)──────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the natural logarithm of the argument (same as LOG).

The argument can be of any built-in numeric data type. It is converted to 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 is null, the result is the null value.

## LOCATE

```
►►──LOCATE──(──expression1──,──expression2─────────────────)──────────►◄
                                          └─,──expression3─┘
```

The schema is SYSFUN.

Returns the starting position of the first occurrence of *expression1* within *expression2*. If the optional *expression3* is specified, it indicates the character position in *expression2* at which the search is to begin. If *expression1* is not found within *expression2*, the value 0 is returned.

If the first argument is a character string, then the second argument must be a character string. For a VARCHAR the maximum length is 4 000 bytes and for a CLOB the maximum length is 1 048 576 bytes. If the first argument is a binary string, then the second argument must be a binary string with a maximum length of 1 048 576 bytes. The third argument must be is INTEGER or SMALLINT.

The result of the function is INTEGER. The result can be null; if any argument is null, the result is the null value.

Example:
- Find the location of the letter 'N' (first occurrence) in the word 'DINING'.
  **VALUES LOCATE (**'N', 'DINING'**)**

  This example returns the following:
  ```
  1
  -----------
            3
  ```

## LOG

►►──LOG──(──*expression*──)────────────────────────────►◄

The schema is SYSFUN.

Returns the natural logarithm of the argument (same as LN).

The argument can be of any built-in numeric data type. It is converted to 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 is null, the result is the null value.

**LOG10**

## LOG10

►►──LOG10──(──*expression*──)────────────────────────────►◄

The schema is SYSFUN.

Returns the base 10 logarithm of the argument.

The argument can be of any built-in numeric type. It is converted to 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 is null, the result is the null value.

## LONG_VARCHAR

►►──LONG_VARCHAR──(──*character-string-expression*──)──────────────────────►◄

The schema is SYSIBM.

The LONG_VARCHAR function returns a LONG VARCHAR representation of a character string data type.

*character-string-expression*
> An *expression* that returns a value that is a character string with a maximum length of 32 700 bytes.

The result of the function is a LONG VARCHAR. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

## LONG_VARGRAPHIC

### LONG_VARGRAPHIC

```
►►──LONG_VARGRAPHIC──(──graphic-expression──)─────────────────────────────►◄
```

The schema is SYSIBM.

The LONG_VARGRAPHIC function returns a LONG VARGRAPHIC
representation of a double-byte character string.

*graphic-expression*
   An *expression* that returns a value that is a graphic string with a maximum
   length of 16 350 double byte characters.

The result of the function is a LONG VARGRAPHIC. If the argument can be
null, the result can be null; if the argument is null, the result is the null value.

## LTRIM

```
►►──LTRIM──(──string-expression──)──────────────────────────►◄
```

The schema is SYSIBM.[39]

The LTRIM function removes blanks from the beginning of *string-expression*.

The argument can be a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type.
- If the argument is a graphic string in a DBCS or EUC database, then the leading double byte blanks are removed.
- If the argument is a graphic string in a Unicode database, then the leading UCS-2 blanks are removed.
- Otherwise, the leading single byte blanks are removed.

The result data type of the function is:
- VARCHAR if the data type of *string-expression* is VARCHAR or CHAR
- VARGRAPHIC if the data type of *string-expression* is VARGRAPHIC or GRAPHIC

The length parameter of the returned type is the same as the length parameter of the argument data type.

The actual length of the result for character strings is the length of *string-expression* minus the number of bytes removed for blank characters. The actual length of the result for graphic strings is the length (in number of double byte characters) of *string-expression* minus the number of double byte blank characters removed. If all of the characters are removed, the result is an empty, varying-length string (length is zero).

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable HELLO is defined as CHAR(9) and has a value of ' Hello'.

```
VALUES LTRIM(:HELLO)
```

The result is 'Hello'.

---

39. The SYSFUN version of this function continues to be available with support for LONG VARCHAR and CLOB arguments. See "LTRIM (SYSFUN schema)" on page 300 for a description.

**LTRIM (SYSFUN schema)**

## LTRIM (SYSFUN schema)

►►──LTRIM──(──*expression*──)──────────────────────────────►◄

The schema is SYSFUN.

Returns the characters of the argument with leading blanks removed.

The argument can be of any built-in character string 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 the argument is VARCHAR (not exceeding 4 000 bytes) or CHAR
- CLOB(1M) if the argument is CLOB or LONG VARCHAR.

The result can be null; if the argument is null, the result is the null value.

## MICROSECOND

```
►►──MICROSECOND──(──expression──)──────────────────────────────────────►◄
```

The schema is SYSIBM.

The MICROSECOND function returns the microsecond part of a value.

The argument must be a timestamp, timestamp duration or a valid character string representation of a timestamp that is neither a CLOB nor a LONG VARCHAR.

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 timestamp or a valid string representation of a timestamp:
  - The integer ranges from 0 through 999 999.
- 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

►►—MIDNIGHT_SECONDS—(—*expression*—)————————————————►◄

The schema is SYSFUN.

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 argument must be a time, timestamp, or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR.

T>he result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

Example:
- 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 47410 seconds between the specified time and midnight.
- 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.

## MINUTE

```
►►──MINUTE──(──expression──)────────────────────────────────────────────────►◄
```

The schema is SYSIBM.

The MINUTE function returns the minute part of a value.

The argument must be a time, timestamp, time duration, timestamp duration
or a valid character string representation of a time or timestamp that is
neither a CLOB nor a LONG VARCHAR.

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 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.

Example:
- 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
    ```

# MOD

## MOD

```
►►──MOD──(──expression──,──expression──)──────────────────────────►◄
```

The schema is SYSFUN.

Returns the remainder of the first argument divided by the second argument. 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

```
►►──MONTH──(──expression──)──────────────────────────────────►◄
```

The schema is SYSIBM.

The MONTH function returns the month part of a value.

The argument must be a date, timestamp, date duration, timestamp duration or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

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

### MONTHNAME

►►──MONTHNAME──(──*expression*──)───────────────────────────►◄

The schema is SYSFUN.

Returns a mixed case character string containing the name of month (e.g. January) for the month portion of the argument, based on the locale when the database was started.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

The result of the function is VARCHAR(100). The result can be null; if the argument is null, the result is the null value.

## NODENUMBER

```
►►──NODENUMBER──(──column-name──)────────────────────────────►◄
```

The schema is SYSIBM.

The NODENUMBER function returns the partition number of the row. For example, if used in a SELECT clause, it returns the partition number for each row of the table that was used to form the result of the SELECT statement.

The partition number returned on transition variables and tables is derived from the current transition values of the partitioning key columns. For example, in a before insert trigger, the function will return the projected partition number given the current values of the new transition variables. However, the values of the partitioning key columns may be modified by a subsequent before insert trigger. Thus, the final partition number of the row when it is inserted into the database may differ from the projected value.

The argument must be the qualified or unqualified name of a column of a table. The column can have any data type.[40] If *column-name* references 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. See "Notes" on page 722 for the definition of a deletable view.

The specific row (and table) for which the partition number is returned by the NODENUMBER function is determined from the context of the SQL statement that uses the function.

The data type of the result is INTEGER and is never null. Since row-level information is returned, the results are the same, regardless of which column is specified for the table. If there is no db2nodes.cfg file, the result is 0.

The NODENUMBER function cannot be used on replicated tables or within check constraints (SQLSTATE 42881).

Examples:
• Count the number of rows where the row for an EMPLOYEE is on a different partition from the employee's department description in DEPARTMENT.

---

40. This function may not be used as a source function when creating a user-defined function. Since it accepts any data types as an argument, it is not necessary to create additional signatures to support user-defined distinct types.

## NODENUMBER

```
SELECT COUNT(*) FROM DEPARTMENT D, EMPLOYEE E
    WHERE D.DEPTNO=E.WORKDEPT
    AND NODENUMBER(E.LASTNAME) <> NODENUMBER(D.DEPTNO)
```

- Join the EMPLOYEE and DEPARTMENT tables where the rows of the two tables are on the same partition.

```
SELECT * FROM DEPARTMENT D, EMPLOYEE E
    WHERE NODENUMBER(E.LASTNAME) = NODENUMBER(D.DEPTNO)
```

- Log the employee number and the projected partition number of the new row into a table called EMPINSERTLOG1 for any insertion of employees by creating a before trigger on the table EMPLOYEE.

```
CREATE TRIGGER EMPINSLOGTRIG1
BEFORE INSERT ON EMPLOYEE
REFERENCING NEW AW NEWTABLE
FOR EACH MODE ROW MODE DB2SQL
INSERT INTO EMPINSERTLOG1
VALUES(NEWTABLE.EMPNO, NODENUMBER(NEWTABLE.EMPNO))
```

## NULLIF

►►──NULLIF──(──*expression*──,──*expression*──)────────────────────►◄

The schema is SYSIBM.

The NULLIF function returns a null value if the arguments are equal, otherwise it returns the value of the first argument.

The arguments must be comparable (see "Assignments and Comparisons" on page 91). They can be of either a built-in (other than a long string or DATALINK) or distinct data type (other than based on a long string or DATALINK). [41] The attributes of the result are the attributes of the first argument.

The result of using NULLIF(e1,e2) is the same as using the expression
```
CASE WHEN e1=e2 THEN NULL ELSE e1 END
```

Note that 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.

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.

---

41. This function may not be used as a source function when creating a user-defined function. Since it accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.

**PARTITION**

## PARTITION

►►──PARTITION──(──*column-name*──)──────────────────────────►◄

The schema is SYSIBM.

The PARTITION function returns the partitioning map index of the row obtained by applying the partitioning function on the partitioning key value of the row. For example, if used in a SELECT clause, it returns the partitioning map index for each row of the table that was used to form the result of the SELECT statement.

The partitioning map index returned on transition variables and tables is derived from the current transition values of the partitioning key columns. For example, in a before insert trigger, the function will return the projected partitioning map index given the current values of the new transition variables. However, the values of the partitioning key columns may be modified by a subsequent before insert trigger. Thus, the final partitioning map index of the row when it is inserted into the database may differ from the projected value.

The argument must be the qualified or unqualified name of a column of a table. The column can have any data type.[42] If *column-name* references 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. See "Notes" on page 722 for the definition of a deletable view.

The specific row (and table) for which the partitioning map index is returned by the PARTITION function is determined from the context of the SQL statement that uses the function.

The data type of the result is INTEGER in the range 0 to 4095. For a table with no partitioning 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.

The PARTITION function cannot be used on replicated tables or within check constraints(SQLSTATE 42881).

---

42. This function may not be used as a source function when creating a user-defined function. Since it accepts any data type as an arguments, it is not necessary to create additional signatures to support user-defined distinct types.

Example:

- List the employee numbers (EMPNO) from the EMPLOYEE table for all rows with a partitioning map index of 100.

```
SELECT EMPNO FROM EMPLOYEE
  WHERE PARTITION(PHONENO) = 100
```

- Log the employee number and the projected partitioning 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 MODE ROW MODE DB2SQL
INSERT INTO EMPINSERTLOG2
  VALUES(NEWTABLE.EMPNO, PARTITION(NEWTABLE.EMPNO))
```

## POSSTR

```
►►──POSSTR──(──source-string──,──search-string──)──────────────────────────►◄
```

The schema is SYSIBM.

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*). 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 source string in which the search is to take place.

The expression can be specified by any one of:
- a constant
- a special register
- a host variable (including a locator variable or a file reference variable)
- a scalar function
- a large object locator
- a column name
- an expression concatenating any of the above

*search-string*
An expression that specifies the string that is to be searched for.

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 above
- an expression concatenating any of the above

with the restrictions that:
- No element in the expression can be of type LONG VARCHAR, CLOB, LONG VARGRAPHIC or DBCLOB. In addition, it cannot be a BLOB file reference variable.
- The actual length of *search-string* cannot be more than 4 000 bytes.

Note that these rules are the same as those for the *pattern-expression* described in "LIKE Predicate" on page 177.

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 graphic (DBCS) character.

The POSSTR function accepts mixed data strings. However, POSSTR operates on a strict byte-count basis, oblivious to changes between single and multi-byte characters.

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 or LONG VARCHAR, with an actual length of 32 672 bytes or less.
  - If *source-string* is a graphic string, then *search-string* must be a graphic string, but not a DBCLOB or LONG VARGRAPHIC, with an actual length of 16 336 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 32 672 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
  ```

**POWER**

## POWER

►►──POWER──(──*expression1*──,──*expression2*──)────────────────────────►◄

The schema is SYSFUN.

Returns the value of *expression1* to the power of *expression2*.

The arguments can be of any built-in numeric data type. DECIMAL and REAL arguments are converted to double-precision floating-point number.

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
- DOUBLE otherwise.

The result can be null; if any argument is null, the result is the null value.

## QUARTER

```
►►──QUARTER──(──expression──)──────────────────────────────────►◄
```

The schema is SYSFUN.

Returns an integer value in the range 1 to 4 representing the quarter of the year for the date specified in the argument.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

## RADIANS

### RADIANS

►►──RADIANS──(──*expression*──)──────────────────────────────►◄

The schema is SYSFUN.

Returns the number of radians converted from argument which is expressed in degrees.

The argument can be of any built-in numeric data types. It is converted to 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 is null, the result is the null value.

## RAISE_ERROR

```
►►──RAISE_ERROR──(──sqlstate──,──diagnostic-string──)────────────────────►◄
```

The schema is SYSIBM.

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 RAISE_ERROR function always returns NULL with an undefined data type.

*sqlstate*
> A character string containing exactly 5 characters. It must be of type CHAR defined with a length of 5 or type VARCHAR defined with a length of 5 or greater. The *sqlstate* value must follow the rules for application-defined SQLSTATEs as follows:
>
> - 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' since 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 of type CHAR or VARCHAR that returns a character string of up to 70 bytes that describes the error condition. If the string is longer than 70 bytes, it will be truncated.

In order to use this function in a context where Rules for Result Data Types do not apply (such as alone in a select list), 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.

## RAISE_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

```
►►──RAND──(──┬─────────────┬──)──────────────────────────►◄
             └─expression──┘
```

The schema is SYSFUN.

Returns a random floating point value between 0 and 1 using the argument as the optional seed value.

An argument is not required, but if it is specified it can be either INTEGER or SMALLINT.

The result of the function is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

## REAL

```
►►──REAL──(──numeric-expression──)────────────────────────────►◄
```

The schema is SYSIBM.

The REAL function returns a single-precision floating-point representation of a number.

The argument is an expression that returns a value of any built-in numeric data type.

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.

The result is the same number that would occur if the argument were assigned to a single-precision floating-point column or variable.

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 result is desired 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
```

## REPEAT

►►—REPEAT—(—*expression*—,—*expression*—)—————————————————————►◄

The schema is SYSFUN.

Returns a character string composed of the first argument repeated the number of times specified by the second argument.

The first argument is a character string or binary string type. For a VARCHAR the maximum length is 4 000 bytes and for a CLOB or a binary string the maximum length is 1 048 576 bytes. The second argument can be SMALLINT or INTEGER.

The result of the function is:
- VARCHAR(4000) if the first argument is VARCHAR (not exceeding 4 000 bytes) or CHAR
- CLOB(1M) if the first argument is CLOB or LONG VARCHAR
- BLOB(1M) if the first argument is BLOB.

The result can be null; if any argument is null, the result is the null value.

Example:
- List the phrase 'REPEAT THIS' five times.

  **VALUES CHAR(REPEAT(**'REPEAT THIS'**, 5), 60)**

  This example return the following:
  ```
  1
  ------------------------------------------------------------
  REPEAT THISREPEAT THISREPEAT THISREPEAT THISREPEAT THIS
  ```

  As mentioned, the output of the REPEAT function is VARCHAR(4000). For the above example the function CHAR has been used to limit the output of REPEAT to 60 bytes.

## REPLACE

```
►►──REPLACE──(──expression1──,──expression2──,──expression3──)────────────────►◄
```

The schema is SYSFUN.

Replaces all occurrences of *expression2* in *expression1* with *expression3*.

The first argument can be of any built-in character string or binary string type. For a VARCHAR the maximum length is 4 000 bytes and for a CLOB or a binary string the maximum length is 1 048 576 bytes. CHAR is converted to VARCHAR and LONG VARCHAR is converted to CLOB(1M). The second and third arguments are identical to the first argument.

The result of the function is:
- VARCHAR(4000) if the first, second and third arguments are VARCHAR or CHAR
- CLOB(1M) if the first, second and third arguments are CLOB or LONG VARCHAR
- BLOB(1M) if the first, second and third arguments are BLOB.

The result can be null; if any argument is null, the result is the null value.

Example:
- Replace all occurrence of the letter 'N' in the word 'DINING' with 'VID'.
  ```
  VALUES CHAR (REPLACE ('DINING', 'N', 'VID'), 10)
  ```

  This example returns the following:
  ```
  1
  ----------
  DIVIDIVIDG
  ```

  As mentioned, the output of the REPLACE function is VARCHAR(4000). For the above example the function CHAR has been used to limit the output of REPLACE to 10 bytes.

## RIGHT

►►—RIGHT—(—*expression1*—,—*expression2*—)——————————————————►◄

The schema is SYSFUN.

Returns a string consisting of the rightmost *expression2* bytes in *expression1*.

The first argument is a character string or binary string type. For a VARCHAR the maximum length is 4 000 bytes and for a CLOB or a binary string the maximum length is 1 048 576 bytes. The second argument can be INTEGER or SMALLINT.

The result of the function is:
- VARCHAR(4000) if the first argument is VARCHAR (not exceeding 4 000 bytes) or CHAR
- CLOB(1M) if the first argument is CLOB or LONG VARCHAR
- BLOB(1M) if the first argument is BLOB.

The result can be null; if any argument is null, the result is the null value.

**ROUND**

## ROUND

```
►►──ROUND──(──expression1──,──expression2──)──────────────────────────►◄
```

The schema is SYSFUN.

Returns the *expression1* rounded to *expression2* places right of the decimal point. If *expression2* is negative, *expression1* is rounded to the absolute value of *expression2* places to the left of the decimal point.

The first argument can be of any built-in numeric data type. The second argument can be INTEGER or SMALLINT. DECIMAL and REAL are converted to double-precision floating-point number for processing by the function.

The result of the function is:
- INTEGER if the first argument is INTEGER or SMALLINT
- BIGINT if the first argument is BIGINT
- DOUBLE if the first argument is DOUBLE, DECIMAL or REAL.

The result can be null; if any argument is null, the result is the null value.

Example:
- Display the number 873.726 rounded to 2, 1, 0, -1 and -2 decimal places respectively.

```
VALUES (DECIMAL(ROUND(873.726,2),6,3), DECIMAL(ROUND(873.726,1),6,3),
        DECIMAL(ROUND(873.726,0),6,3), DECIMAL(ROUND(873.726,-1),6,3),
        DECIMAL(ROUND(873.726,-2),6,3))
```

The above example returns:
```
1        2        3        4        5
-------- -------- -------- -------- ---------
 873.730  873.700  874.000  870.000   900.000
```

As mentioned, the output of the ROUND function is DOUBLE. For the above example the function DECIMAL has been used to limit the output of ROUND.

## RTRIM

```
►►──RTRIM──(──string-expression──)────────────────────────────►◄
```

The schema is SYSIBM.[43]

The RTRIM function removes blanks from the end of *string-expression*.

The argument can be a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type.
- If the argument is a graphic string in a DBCS or EUC database, then the trailing double byte blanks are removed.
- If the argument is a graphic string in a Unicode database, then the trailing UCS-2 blanks are removed.
- Otherwise, the trailing single byte blanks are removed.

The result data type of the function is:
- VARCHAR if the data type of *string-expression* is VARCHAR or CHAR
- VARGRAPHIC if if the data type of *string-expression* is VARGRAPHIC or GRAPHIC

The length parameter of the returned type is the same as the length parameter of the argument data type.

The actual length of the result for character strings is the length of *string-expression* minus the number of bytes removed for blank characters. The actual length of the result for graphic strings is the length (in number of double byte characters) of *string-expression* minus the number of double byte blank characters removed. If all of the characters are removed, the result is an empty, varying-length string (length is zero).

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable HELLO is defined as CHAR(9) and has a value of 'Hello '.

```
VALUES RTRIM(:HELLO)
```

The result is 'Hello'.

---

43. The SYSFUN version of this function continues to be available with support for LONG VARCHAR and CLOB arguments. See "RTRIM (SYSFUN schema)" on page 326 for a description.

**RTRIM (SYSFUN schema)**

## RTRIM (SYSFUN schema)

```
►►──RTRIM──(──expression──)──────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the characters of the argument with trailing blanks removed.

The argument can be of any built-in character string data types. 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 the argument is VARCHAR (not exceeding 4 000 bytes) or CHAR
- CLOB(1M) if the argument is CLOB or LONG VARCHAR.

The result can be null; if the argument is null, the result is the null value.

## SECOND

►►──SECOND──(──*expression*──)──────────────────────────────────────────◄►

The schema is SYSIBM.

The SECOND function returns the seconds part of a value.

The argument must be a time, timestamp, time duration, timestamp duration or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR.

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 seconds 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 seconds part of the value, which is an integer between –99 and 99. A nonzero result has the same sign as the argument.

Examples:
- Assume that the host variable TIME_DUR (decimal(6,0)) has the value 153045.

    **SECOND**(:TIME_DUR)

  Returns the value 45.
- Assume that the column RECEIVED (timestamp) has an internal value equivalent to 1988-12-25-17.12.30.000000.

    **SECOND**(RECEIVED)

  Returns the value 30.

**SIGN**

## SIGN

```
►►──SIGN──(──expression──)──────────────────────────────────►◄
```

The schema is SYSFUN.

Returns an indicator of the sign of the argument. If the argument is less than zero, −1 is returned. If argument equals zero, 0 is returned. If argument is greater than zero, 1 is returned.

The argument can be of any built-in numeric data types. DECIMAL and REAL are converted to double-precision floating-point number for processing by the function.

The result of the function is:
- SMALLINT if the argument is SMALLINT
- INTEGER if the argument is INTEGER
- BIGINT if the argument is BIGINT
- DOUBLE otherwise.

The result can be null; if the argument is null, the result is the null value.

**SIN**

```
►►──SIN──(──expression──)──────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the sine of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric data types. It is converted to 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 is null, the result is the null value.

## SMALLINT

**SMALLINT**

```
►►──SMALLINT──(──┬──numeric-expression───┬──)──────────────────────────►◄
                 └──character-expression─┘
```

The schema is SYSIBM.

The SMALLINT function returns a small integer representation of a number or character string in the form of a small integer constant.

*numeric-expression*
> An expression that returns a value of any built-in numeric data type.
>
> If the argument is a *numeric-expression*, the result is the same number that would occur if the argument were assigned to a small integer column or variable. If the whole part of the argument is not within the range of small integers, an error occurs. The decimal part of the argument is truncated if present.

*character-expression*
> An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant (SQLSTATE 22018). However, the value of the constant must be in the range of small integers (SQLSTATE 22003). The character string cannot be a long string.
>
> If the argument is a *character-expression*, the result is the same number that would occur if the corresponding integer constant were assigned to a small integer column or variable.

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.

## SOUNDEX

►►──SOUNDEX──(──*expression*──)──────────────────────────────►◄

The schema is SYSFUN.

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 argument can be a character string that is either a CHAR or VARCHAR not exceeding 4 000 bytes.

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 (see "DIFFERENCE" on page 261).

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:

```
EMPNO  LASTNAME
------ ---------------
000110 LUCCHESSI
```

## SPACE

►►—SPACE—(—*expression*—)————————————————◄

The schema is SYSFUN.

Returns a character string consisting of blanks with length specified by the second argument.

The argument can be SMALLINT or INTEGER.

The result of the function is VARCHAR(4000). The result can be null; if the argument is null, the result is the null value.

**SQRT**

►►──SQRT─(─*expression*─)──────────────────────────────────────►◄


The schema is SYSFUN.

Returns the square root of the argument.

The argument can be any built-in numeric data type. It has to be converted to double-precision floating-point number for processing by the function.

The result of the function is double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

SUBSTR

## **SUBSTR**

```
►►──SUBSTR──(──string──,──start──────────────)──────────────────────►◄
                              └─,──length──┘
```

The schema is SYSIBM.

The SUBSTR function returns a substring of a string.

If *string* is a character string, the result of the function is a character string represented in the code page of its first argument. If it is a binary string, the result of the function is a binary string. If it is a graphic string, the result of the function is a graphic string represented in the code page of its first argument. 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.

*string*
> An expression that specifies the string from which the result is derived.
>
> If *string* is either a character string or a binary string, a substring of *string* is zero or more contiguous bytes of *string*. If *string* is a graphic string, a substring of *string* is zero or more contiguous double-byte characters of *string*.

*start*
> An expression that specifies the position of the first byte of the result for a character string or a binary string or the position of the first character of the result for a graphic string. *start* must be an integer between 1 and the length or maximum length of *string*, depending on whether *string* is fixed-length or varying-length (SQLSTATE 22011, if out of range). It must be specified as number of bytes in the context of the database code page and not the application code page.

*length*
> An expression that specifies the length of the result. If specified, *length* must be a binary integer in the range 0 to *n*, where *n* equals (the length attribute of *string*) − *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) so that the specified substring of *string* always exists. The default for *length* is the number of bytes from the byte specified by the *start* to the last byte of *string* in the case of character string or binary string or the number of double-byte characters from the character specified by the *start* to the last character of *string* in the case of a graphic 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 bytes 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)`).

Table 16 shows that the result type and length of the SUBSTR function depend on the type and attributes of its inputs.

*Table 16. Data Type and Length of SUBSTR Result*

| String Argument Data Type | Length Argument | Result Data Type |
|---|---|---|
| CHAR(A) | constant ($l$<255) | CHAR($l$) |
| CHAR(A) | not specified but *start* argument is a constant | CHAR(A-*start*+1) |
| CHAR(A) | not a constant | VARCHAR(A) |
| | | |
| VARCHAR(A) | constant ($l$<255) | CHAR($l$) |
| VARCHAR(A) | constant (254<$l$<32673) | VARCHAR($l$) |
| VARCHAR(A) | not a constant or not specified | VARCHAR(A) |
| | | |
| LONG VARCHAR | constant ($l$<255) | CHAR($l$) |
| | | |
| LONG VARCHAR | constant (254<$l$<4001) | VARCHAR($l$) |
| LONG VARCHAR | constant ($l$>4000) | LONG VARCHAR |
| LONG VARCHAR | not a constant or not specified | LONG VARCHAR |
| | | |
| CLOB(A) | constant ($l$) | CLOB($l$) |
| CLOB(A) | not a constant or not specified | CLOB(A) |
| | | |
| GRAPHIC(A) | constant ($l$<128) | 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 ($l$<128) | GRAPHIC($l$) |
| VARGRAPHIC(A) | constant (127<$l$<16337) | VARGRAPHIC($l$) |
| VARGRAPHIC(A) | not a constant | VARGRAPHIC(A) |

*Table 16. Data Type and Length of SUBSTR Result  (continued)*

| String Argument Data Type | Length Argument | Result Data Type |
|---|---|---|
| | | |
| LONG VARGRAPHIC | constant (*l*<128) | GRAPHIC(*l*) |
| LONG VARGRAPHIC | constant (127<*l*<2001) | VARGRAPHIC(*l*) |
| LONG VARGRAPHIC | constant (*l*>2000) | LONG VARGRAPHIC |
| LONG VARGRAPHIC | not a constant or not specified | LONG VARGRAPHIC |
| | | |
| DBCLOB(A) | constant (*l*) | DBCLOB(*l*) |
| DBCLOB(A) | not a constant or not specified | DBCLOB(A) |
| | | |
| BLOB(A) | constant (*l*) | BLOB(*l*) |
| BLOB(A) | not a constant or not specified | BLOB(A) |
| | | |
| | | |

If *string* is a fixed-length string, omission of *length* is an implicit specification of LENGTH(*string*) - *start* + 1. If *string* is a varying-length string, omission of *length* is an implicit specification of zero or LENGTH(*string*) - *start* + 1, whichever is greater.

Examples:

- Assume the host variable NAME (VARCHAR(50)) has a value of ′BLUE JAY′ and the host variable SURNAME_POS (int) has a value of 6.

  **SUBSTR(**:NAME, :SURNAME_POS**):ehp2**s

  Returns the value 'JAY'

  **SUBSTR(**:NAME, :SURNAME_POS,1)

  Returns the value ′J′.

- 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 preclude initial words such as ′OPERATIONS′.

**Notes:**

1. In dynamic SQL, *string*, *start*, and *length* may 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.

2. Though not explicitly stated in the result definitions above, it follows from these semantics that if *string* is a mixed single- and multi-byte character string, the result may contain fragments of multi-byte characters, depending upon the values of *start* and *length*. That is, the result could possibly begin with the second byte of a double-byte character, and/or end with the first byte of a double-byte character. The SUBSTR function does not detect such fragments, nor provides any special processing should they occur.

## TABLE_NAME

```
►►──TABLE_NAME──(──objectname──────────────────────)──────────────────────►◄
                              └─,──objectschema─┘
```

The schema is SYSIBM.

The TABLE_NAME function returns an unqualified name of the object found after any alias chains have been resolved. 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 unqualified name of the starting point is returned. The resulting name may be of a table, view, or undefined object.

*objectname*
> A character expression representing the unqualified name (usually of an existing alias) to be resolved. *objectname* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 characters.

*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 characters.
>
> If *objectschema* is not supplied, the default schema is used for the qualifier.

The data type of the result of the function is VARCHAR(128). If *objectname* can be null, the result can be null; if *objectname* is null, the result is the null value. If *objectschema* 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:

**table**    The value for *objectname* 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 *objectname* 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 *objectname* 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.

Examples:

See Examples section in "TABLE_SCHEMA" on page 340.

## TABLE_SCHEMA

### TABLE_SCHEMA

```
►►──TABLE_SCHEMA──(──objectname──────────────────)────────────────────────►◄
                               └─,──objectschema─┘
```

The schema is SYSIBM.

The TABLE_SCHEMA function returns the schema name of the object found
after any alias chains have been resolved. 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 of the starting point is
returned. The resulting schema name may be of a table, view, or undefined
object.

*objectname*
> A character expression representing the unqualified name (usually of an
> existing alias) to be resolved. *objectname* must have a data type of CHAR
> or VARCHAR and a length greater than 0 and less than 129 characters.

*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 characters.
>
> If *objectschema* is not supplied, the default schema is used for the qualifier.

The data type of the result of the function is VARCHAR(128). If *objectname* can
be null, the result can be null; if *objectname* is null, the result is the null value.
If *objectschema* 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:

**table**   The value for *objectname* was either a table name (the input or default
value of *objectschema* is returned) or an alias name that resolved to a
table for which the schema name is returned.

**view**   The value for *objectname* was either a view name (the input or default
value of *objectschema* is returned) or an alias name that resolved to a
view for which the schema name is returned.

**undefined object**

> The value for *objectname* was either an undefined object (the input or
> default value of *objectschema* is returned) or an alias name that
> resolved to an undefined object for which the schema name is
> returned.

Therefore, if a non-null *objectname* 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.

Examples:
- 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.
- 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.
- 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'.
- 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

**TAN**

```
►►──TAN──(──expression──)──────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the tangent of the argument, where the argument is an angle expressed in radians.

The argument can be any built-in numeric data type. It has to be converted to double-precision floating-point number for processing by the function.

The result of the function is double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

**TIME**

►►──TIME──(──*expression*──)────────────────────────────────────►◄

The schema is SYSIBM.

The TIME function returns a time from a value.

The argument must be a time, timestamp, or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR.

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 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 character string:
  - The result is the time represented by the character 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

### TIMESTAMP

```
►►─TIMESTAMP─(─expression──────────────)────────────────────►◄
                        └─,expression─┘
```

The schema is SYSIBM.

The TIMESTAMP function returns a timestamp from a value or a pair of values.

The rules for the arguments depend on whether the second argument is specified.

- If only one argument is specified:
  - It must be a timestamp, a valid character string representation of a timestamp, or a character string of length 14 that is neither a CLOB nor a LONG VARCHAR.

    A character string of length 14 must be 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 both arguments are specified:
  - The first argument must be a date or a valid character string representation of a date and the second argument must be a time or a valid string representation of a time.

The result of the function is a timestamp. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

The other rules depend on whether the second argument is specified:

- If both arguments are specified:
  - The result is a timestamp with the date specified by the first argument and the time specified by the second argument. The microsecond part of the timestamp is zero.
- If only one argument is specified and it is a timestamp:
  - The result is that timestamp.
- If only one argument is specified and it is a character string:
  - The result is the timestamp represented by that character string. If the argument is a character string of length 14, the timestamp has a microsecond part of zero.

Example:

- Assume the column START_DATE (date) has a value equivalent to
  1988-12-25, and the column START_TIME (time) has a value equivalent to
  17.12.30.

    **TIMESTAMP**(START_DATE, START_TIME)

  Returns the value '1988-12-25-17.12.30.000000'.

## TIMESTAMP_ISO

### TIMESTAMP_ISO

```
►►──TIMESTAMP_ISO──(──expression──)──────────────────────────►◄
```

The schema is SYSFUN.

Returns a timestamp value based on 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.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

The result of the function is TIMESTAMP. The result can be null; if the argument is null, the result is the null value.

## TIMESTAMPDIFF

►►─TIMESTAMPDIFF─(─*expression*─,─*expression*─)──────────────────────►◄

The schema is SYSFUN.

Returns an estimated number of intervals of the type defined by the first argument, based on the difference between two timestamps.

The first argument can be either INTEGER or SMALLINT. Valid values of interval (the first argument) are:

| | |
|-----|---------------------|
| **1** | Fractions of a second |
| **2** | Seconds |
| **4** | Minutes |
| **8** | Hours |
| **16** | Days |
| **32** | Weeks |
| **64** | Months |
| **128** | Quarters |
| **256** | Years |

The second argument is the result of subtracting two timestamps types and converting the result to CHAR(22).

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

The following assumptions may be used in estimating the 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, which is a timestamp duration, 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 a difference in timestamps for '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 so the assumption of 30 days in a month applies.

## TRANSLATE

**character string expression:**

```
►►──TRANSLATE──(──char-string-exp─────────────────────────────────────────►

►──┬───────────────────────────────────────────────────┬──)──────────────►◄
   │                              ┌──,─' '──────┐       │
   └─,──to-string-exp─,──from-string-exp─┼──────────────┼─┘
                                         └─,──pad-char──┘
```

**graphic string expression:**

```
►►──TRANSLATE──(──graphic-string-exp──,──to-string-exp──,──from-string-exp────►

      ┌──,─' '──────┐
►──┬──┼─────────────┼──┬──)──────────────────────────────────────────────►◄
   │  └─,──pad-char──┘  │
   └───────────────────┘
```
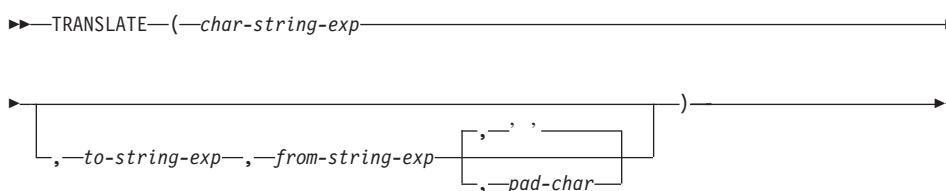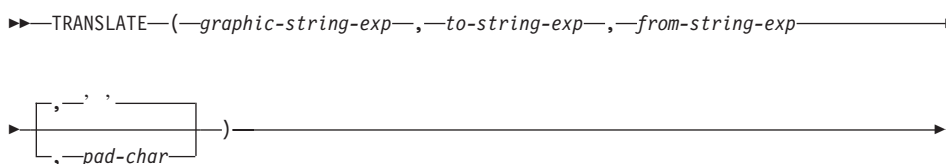
The schema is SYSIBM.

The TRANSLATE function returns a value in which one or more characters in a string expression may have been translated into other characters.

The result of the function has the same data type and code page as the first argument. The length attribute of the result is the same as that of the first argument. If any specified expression can be NULL, the result can be NULL. If any specified expression is NULL, the result will be NULL.

*char-string-exp* or *graphic-string-exp*
  A string to be translated.

*to-string-exp*
  Is a string of characters to which certain characters in the *char-string-exp* will be translated.

  If the *to-string-exp* is not present and the data type is not graphic, all characters in the *char-string-exp* will be monocased (that is, the characters a-z will be translated to the characters A-Z, and characters with diacritical marks will be translated to their upper case equivalents if they exist. For example, in code page 850, é maps to É, but ÿ is not mapped since code page 850 does not include Ÿ).

*from-string-exp*
  Is a string of characters which, if found in the *char-string-exp*, will be

translated to the corresponding character in the *to-string-exp*. If the *from-string-exp* contains duplicate characters, the first one found will be used, and the duplicates will be ignored. If the *to-string-exp* is longer than the *from-string-exp*, the surplus characters will be ignored. If the *to-string-exp* is present, the *from-string-exp* must also be present.

*pad-char-exp*
Is a single character that will be used to pad the *to-string-exp* if the *to-string-exp* is shorter than the *from-string-exp*. The *pad-char-exp* must have a length attribute of one, or an error is returned. If not present, it will be taken to be a single-byte blank.

The arguments may be either strings of data type CHAR or VARCHAR, or graphic strings of data type GRAPHIC or VARGRAPHIC. They may not have data type LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, or DBCLOB.

With *graphic-string-exp*, only the *pad-char-exp* is optional (if not provided, it will be taken to be the double-byte blank), and each argument, including the pad character, must be of graphic data type.

The result is the string that occurs after translating all the characters in the *char-string-exp* or *graphic-string-exp* that occur in the *from-string-exp* to the corresponding character in the *to-string-exp* or, if no corresponding character exists, to the pad character specified by the *pad-char-exp*.

The code page of the result of TRANSLATE is always the same as the code page of the first operand, which is never converted. Each of the other operands is converted to the code page of the first operand unless it or the first operand is defined as FOR BIT DATA (in which case there is no conversion).

If the arguments are of data type CHAR or VARCHAR, the corresponding characters of the *to-string-exp* and the *from-string-exp* must have the same number of bytes. For example, it is not valid to translate a single-byte character to a multi-byte character or vice versa. An error will result if an attempt is made to do this. The *pad-char-exp* must not be the first byte of a valid multi-byte character, or SQLSTATE 42815 is returned. If the *pad-char-exp* is not present, it will be taken to be a single-byte blank.

If only the *char-string-exp* is specified, single-byte characters will be monocased and multi-byte characters will remain unchanged.

Examples:
- Assume the host variable SITE (VARCHAR(30)) has a value of 'Hanauma Bay'.

# TRANSLATE

```
TRANSLATE(:SITE)
```

Returns the value ′HANAUMA BAY′.

```
TRANSLATE(:SITE 'j','B')
```

Returns the value ′Hanauma jay′.

```
TRANSLATE(:SITE,'ei','aa')
```

Returns the value ′Heneume Bey′.

```
TRANSLATE(:SITE,'bA','Bay','%')
```

Returns the value ′HAnAumA bA%′.

```
TRANSLATE(:SITE,'r','Bu')
```

Returns the value ′Hana ma ray′.

## TRUNCATE or TRUNC

►►─┬─TRUNCATE─┬─(─*expression*─,─*expression*─)──────────────────────►◄
　　└─TRUNC────┘

The schema is SYSFUN.

Returns argument1 truncated to argument2 places right of decimal point. If argument2 is negative, argument1 is truncated to the absolute value of argument2 places to the left of the decimal point.

The first argument can be any built-in numeric data type. The second argument has to be an INTEGER or SMALLINT. DECIMAL and REAL are converted to double-precision floating-point number for processing by the function.

The result of the function is:
- INTEGER if the first argument is INTEGER or SMALLINT
- BIGINT if the first argument is BIGINT
- DOUBLE if the first argument is DOUBLE, DECIMAL or DOUBLE.

The result can be null; if any argument is null, the result is the null value.

## TYPE_ID

```
►►──TYPE_ID──(──expression──)─────────────────────────────►◄
```

The schema is SYSIBM.

The TYPE_ID function returns the internal type identifier of the dynamic data type of the *expression*.

The argument must be a user-defined structured type.[44]

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 the type schema and type name.

Examples:
- 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.

---

44. This function may not be used as a source function when creating a user-defined function. Since it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.

## TYPE_NAME

►►──TYPE_NAME──(──*expression*──)──────────────────────────────────►◄

The schema is SYSIBM.

The TYPE_NAME function returns the unqualified name of the dynamic data type of the *expression*.

The argument must be a user-defined structured type.[45]

The data type of the result of the function is VARCHAR(18). 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.

Examples:
- 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.

---

45. This function may not be used as a source function when creating a user-defined function. Since it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.

**TYPE_SCHEMA**

## TYPE_SCHEMA

►►──TYPE_SCHEMA──(──*expression*──)──────────────────────────►◄

The schema is SYSIBM.

The TYPE_SCHEMA function returns the schema name of the dynamic data type of the *expression*.

The argument must be a user-defined structured type.[46]

The data type of the result of the function is VARCHAR(128). 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.

Examples:

See Examples section in "TYPE_NAME" on page 353.

---

46. This function may not be used as a source function when creating a user-defined function. Since it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.

## UCASE or UPPER

```
►►──┬─UCASE─┬──(─expression─)────────────────────────────►◄
    └─UPPER─┘
```
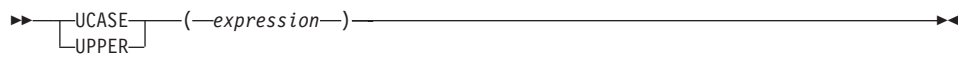
The schema is SYSIBM.[47]

The UCASE or UPPER function is identical to the TRANSLATE function except that only the first argument (*char-string-exp*) is specified. For more information, see "TRANSLATE" on page 348.

---

47. The SYSFUN version of this function continues to be available for upward compatibility. See Version 5 documentation for a description.

## VALUE

### VALUE

```
►►──VALUE──(──expression──┬─,expression──┬──)────────────────────►◄
                          └──────◄────────┘
```
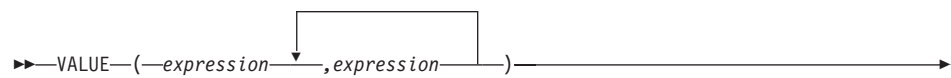
The schema is SYSIBM.

The VALUE function returns the first argument that is not null.

VALUE is a synonym for COALESCE. See "COALESCE" on page 244 for details.

## VARCHAR

**Character to Varchar:**

►►──VARCHAR──(──*character-string-expression*──┬──────────────┬──)──────────────►◄
                                                    └──,──*integer*──┘

**Datetime to Varchar:**

►►──VARCHAR──(──*datetime-expression*──)────────────────────────────────►◄

**Graphic to Varchar:**

►►──VARCHAR──(──*graphic-string-expression*──┬──────────────┬──)──────────────►◄
                                                  └──,──*integer*──┘

The schema is SYSIBM.

The VARCHAR function returns a varying-length character string representation of a character string, datetime value or graphic string (UCS-2 only).

The result of the function is a varying-length string (VARCHAR data type). If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Graphic to Varchar is valid for a UCS-2 database only. For non-Unicode databases, this is not allowed.

**Character to Varchar**

*character-string-expression*
> An expression whose value must be of a character-string data type other than LONG VARGRAPHIC and DBCLOB, with a maximum length of 32 672 bytes.

*integer*
> The length attribute for the resulting varying-length character string. The value must be between 0 and 32 672. If this argument is not specified, the length of the result is the same as the length of the argument.

**Datetime to Varchar**

## VARCHAR

*datetime-expression*
> An expression whose value must be of a date, time, or timestamp data
> type.

**Graphic to Varchar**

*graphic-string-expression*
> An expression whose value must be of a graphic-string data type other
> than LONG VARGRAPHIC and DBCLOB, with a maximum length of
> 16 336 bytes.

*integer*
> The length attribute for the resulting varying-length character string. The
> value must be between 0 and 32 672. If this argument is not specified, the
> length of the result is the same as the length of the argument.

Example:

- Using the EMPLOYEE table, set the host variable JOB_DESC
  (VARCHAR(8)) to the VARCHAR equivalent of the job description (JOB
  defined as CHAR(8)) for employee Delores Quintana.

```
SELECT VARCHAR(JOB)
  INTO :JOB_DESC
  FROM EMPLOYEE
  WHERE LASTNAME = 'QUINTANA'
```

## VARGRAPHIC

**Character to Vargraphic:**

▶▶──VARGRAPHIC──(──*character-string-expression*──)────────────────────▶◀

**Graphic to Vargraphic:**

▶▶──VARGRAPHIC──(──*graphic-string-expression*──┬────────────┬──)──────────▶◀
                                                └─,──*integer*─┘

The schema is SYSIBM.

The VARGRAPHIC function returns a graphic string representation of a:
- character string value, converting single byte characters to double byte characters
- graphic string value, if the first argument is any type of graphic string.

The result of the function is a varying length graphic string (VARGRAPHIC data type). If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

**Character to Vargraphic**

*character-string-expression*
>An expression whose value must be of a character string data type other than LONG VARCHAR or CLOB, and whose maximum length must not be greater than 16 336 bytes.

The length attribute of the result is equal to the length attribute of the argument.

Let S denote the value of the *character-string-expression*. Each single-byte character in S is converted to its equivalent double-byte representation or to the double-byte substitution character in the result; each double-byte character in S is mapped 'as-is'. If the first byte of a double-byte character appears as the last byte of S, it is converted into the double-byte substitution character. The sequential order of the characters in S is preserved.

The following are additional considerations for the conversion.
- For a Unicode database, this function converts the character-string from the code page of the operand into UCS-2. Every character of the operand,

including DBCS characters, is converted. If the second argument is given, it specifies the desired length (number of UCS-2 characters) of the resulting UCS-2 string.

- The conversion to double-byte code points by the VARGRAPHIC function is based on the code page of the operand.
- Double-byte characters of the operand are not converted (see "Appendix O. Japanese and Traditional-Chinese EUC Considerations" on page 1181 for exception). All other characters are converted to their corresponding double-byte depiction. If there is no corresponding double-byte depiction, the double-byte substitution character for the code page is used.
- No warning or error code is generated if one or more double-byte substitution characters are returned in the result.

**Graphic to Vargraphic**

*graphic-string-expression*
  An expression that returns a value that is a graphic string.

*integer*
  The length attribute for the resulting varying length graphic string. The value must be between 0 and 16 336. If this argument is not specified, the length of the result is the same as the length of the argument.

If the length of the *graphic-string-expression* is greater than the length attribute of the result, truncation is performed and a warning is returned (SQLSTATE 01004) unless the truncated characters were all blanks and the *graphic-string-expression* was not a long string (LONG VARGRAPHIC or DBCLOB).

## WEEK

```
►►─WEEK─(─expression─)──────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the week of the year of the argument as an integer value in range 1-54. The week starts with Sunday.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

**YEAR**

## YEAR

```
►►──YEAR──(──expression──)──────────────────────────────────────►◄
```

The schema is SYSIBM.

The YEAR function returns the year part of a value.

The argument must be a date, timestamp, date duration, timestamp duration or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR.

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 9 999.
- If the argument is a date duration or timestamp duration:
  - The result is the year part of the value, which is an integer between –9 999 and 9 999. A nonzero result has the same sign as the argument.

Examples:

- 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)
  ```

- 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
  ```

362

## Table Functions

A table function can be used only in the FROM clause of a statement.
However, expressions or column functions can not be used within a table
function.

Table functions returns columns of a table, resembling a table created by a
simple CREATE TABLE statement.

The table functions that follow may be qualified with the schema name.

## SQLCACHE_SNAPSHOT

## SQLCACHE_SNAPSHOT

►►──SQLCACHE_SNAPSHOT──────────────────────────────────────────►◄

The schema is SYSFUN.

The SQLCACHE_SNAPSHOT returns the the results of a snapshot of the db2 dynamic sql statement cache.

The function does not take any arguments.

It returns a table as listed below. Refer to *System Monitor Guide and Reference* for details on the columns.

*Table 17. Column names and data types of the table returned by SQLCACHE_SNAPSHOT table function*

| Column name | Data type |
| --- | --- |
| NUM_EXECUTIONS | INTEGER |
| NUM_COMPILATIONS | INTEGER |
| PREP_TIME_WORST | INTEGER |
| PREP_TIME_BEST | INTEGER |
| INT_ROWS_DELETED | INTEGER |
| INT_ROWS_INSERTED | INTEGER |
| ROWS_READ | INTEGER |
| INT_ROWS_UPDATED | INTEGER |
| ROWS_WRITE | INTEGER |
| STMT_SORTS | INTEGER |
| TOTAL_EXEC_TIME_S | INTEGER |
| TOTAL_EXEC_TIME_MS | INTEGER |
| TOT_U_CPU_TIME_S | INTEGER |
| TOT_U_CPU_TIME_MS | INTEGER |
| TOT_S_CPU_TIME_S | INTEGER |
| TOT_S_CPU_TIME_MS | INTEGER |
| DB_NAME | VARCHAR(8) |
| STMT_TEXT | CLOB(64K) |

## User-Defined Functions

►►─ *function-name* ─(─┬──────────────────────┬─)───────────────────────►◄
                      │    ┌─ , ◄─────────┐   │
                      └─▼──── *expression* ──┴──┘
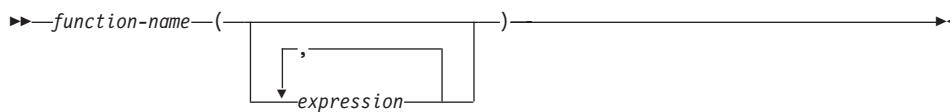
User-defined functions are extensions or additions to the existing built-in
functions of the SQL language. A user-defined function can be a scalar
function, which returns a single value each time it is called, a column
function, which is passed a set of like values and returns a single value for
the set, or a table function, which returns a table. (Note that a UDF can be a
column function only when it is sourced on an existing column function.
Similarly, a UDF can be a table function only if it is an external function.)

A user-defined scalar or column function registered with the database can be
referenced in the same contexts that any built-in function can appear.

A user-defined table function registered with the database can be referenced
only in the FROM clause of a SELECT, as described in "from-clause" on
page 374.

A user-defined function is referenced by means of a qualified or unqualified
function name, followed by parentheses enclosing the function arguments (if
any).

Arguments of the function must correspond in number and position to the
parameters specified in the user-defined function as it was registered with the
database. In addition, the arguments must be of data types promotable to the
data types of the corresponding defined parameters. (see "CREATE
FUNCTION" on page 551).

The result of the function is as specified in the RETURNS clause specified
when the user-defined function was registered. The RETURNS clause
determines if a function is a table function or not.

If the RETURNS NULL ON NULL INPUT clause was specified (or defaulted
to) when the function was registered then, if any argument is null, the result
is null. For table functions, this is interpreted to mean a return table with no
rows (empty table).

There are a collection of user-defined functions provided in the SYSFUN
schema (see Table 15 on page 188).

Examples:

## User-Defined Functions

- Assume that a scalar function called ADDRESS was written to extract the home address from a script format resume. The ADDRESS function expects a CLOB argument and returns a VARCHAR(4000). The following example illustrates the invocation of the ADDRESS function.

  ```
  SELECT EMPNO, ADDRESS(RESUME) FROM EMP_RESUME
    WHERE RESUME_FORMAT = 'SCRIPT'
  ```

- Assume a table T2 with a numeric column A and the ADDRESS function described in the previous example. The following example illustrates an attempt to invoke the ADDRESS function with an incorrect argument.

  ```
  SELECT ADDRESS(A) FROM T2
  ```

  An error (SQLSTATE 42884) is raised since there is no function with a matching name and with a parameter promotable from the argument.

- Assume a table function WHO was written to return information about the sessions on the server machine which were active at the time the statement is executed. The following example illustrates the invocation of WHO in a FROM clause (TABLE keyword with mandatory correlation variable).

  ```
  SELECT ID, START_DATE, ORIG_MACHINE
  FROM TABLE( WHO() ) AS QQ
  WHERE START_DATE LIKE 'MAY%'
  ```

  The column names of the WHO() table are defined in the CREATE FUNCTION statement.

# Chapter 5. 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.

There is another SQL statement that can be used to retrieve at most a single row described under "SELECT INTO" on page 861.

**Authorization**

For each table or view referenced in the query, the authorization ID of the statement must have at least one of the following:

- SYSADM or DBADM authority
- CONTROL privilege
- SELECT privilege.

Group privileges are not checked for queries contained in static SQL statements.

For nicknames referenced in a query, there are no privileges at the federated database to be considered. 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 may be mapped to a different remote authorization ID.

## subselect

```
►►──select-clause─from-clause─────────────────────────────────────►
                              └─where-clause─┘


►────────────────────────────────────────────────────────────────►◄
    └─group-by-clause─┘ └─having-clause─┘
```
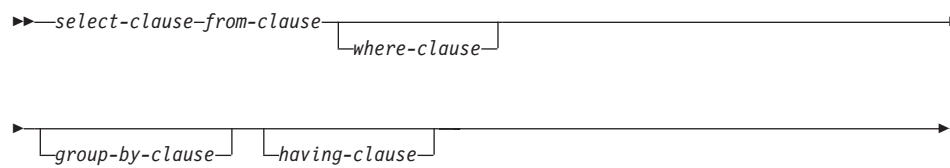
The *subselect* is a component of the fullselect.

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 may be quite different from this description.)

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.

## select-clause

```
>>--SELECT--+--ALL------+--+--*-------------------------------------+--><
            +--DISTINCT-+  |  +-----,-------------------------+      |
                          |  v                               |      |
                          |  +--expression--+------------------+--+ |
                          |  |              +--AS--+          | |
                          |  |              +------new-column-name--+ |
                          |  +--exposed-name.*------------------------+
```
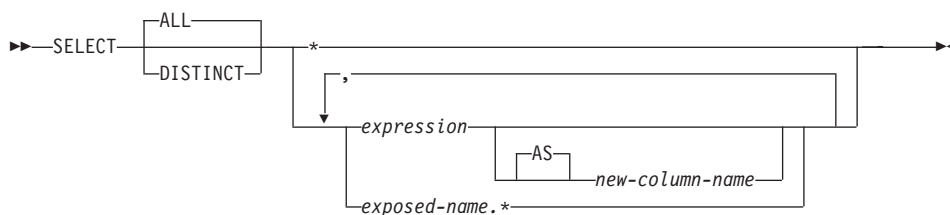
The SELECT clause specifies the columns of the final result table. The column values are produced by the application of the *select list* to R. The select list is the names or expressions specified in the SELECT clause, and R is the result of the previous operation of the subselect. For example, if the 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 have a maximum length that is greater than 254 bytes. DISTINCT may be used more than once in a subselect. This includes SELECT DISTINCT, the use of DISTINCT in a column 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 of the second. (For determining duplicates, two null values are considered equal.)

**Select List Notation:**

\*    Represents a list of names that identify the columns of table R. 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. May be any expression of the type described in Chapter 3., but commonly the expressions used include column names. Each column name used in the select list must unambiguously identify a column of R.

*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.

*name.*\*

Represents the list of names that identify the columns of the result table identified by *exposed-name*. The *exposed-name* may 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.

## Limitations on String Columns

For limitations on the select list, see "Restrictions Using Varying-Length Character Strings" on page 76.

## Applying the Select List

Some of the results of applying the select list to R depend on whether or not 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 a column function) used in the select list must have a GROUP BY clause with:

    – a *grouping-expression* in which each column-name unambiguously
      identifies a column of R (see "group-by-clause" on page 382) 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 column
  functions in the select list.

**If neither GROUP BY nor HAVING is used:**

- Either the select list must not include any column functions, or each
  *column-name* in the select list must be specified within a column function or
  must be a correlated column reference.

- If the select does not include column 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 column 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 that
  allows nulls.

Result columns allow null values if they are derived from:
- Any column function except COUNT or COUNT_BIG
- A column that allows null values
- A scalar function or expression that includes an operand that allows nulls
- A NULLIF function with arguments containing equal values.
- A host variable that has an indicator 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 dereference operation.

**Names of result columns:**

## select-clause

- 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 the result column is derived from a column, then the result column name is the unqualified name of that column.
- If the AS clause is not specified and the result column is derived using a dereference operation, then the result column name is the unqualified name of the target column of the dereference operation.
- All other result column names are unnamed.[48]

**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. |
| an integer constant | INTEGER |
| a decimal constant | DECIMAL, with the precision and scale of the constant |
| a floating-point constant | DOUBLE |
| the name of any numeric variable | the same as the data type of the variable, with the same precision and scale for DECIMAL variables. |
| an expression | For a description of data type attributes, see "Expressions" on page 146. |
| any function | (see Chapter 4. to determine the data type of the result.) |
| a hexadecimal constant representing n bytes | VARCHAR(n). The codepage is the database codepage. |
| 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. |
| a character string constant of length $n$ | VARCHAR($n$) |
| a graphic string constant of length $n$ | VARGRAPHIC($n$) |

---

48. The system assigns temporary numbers (as character strings) to these columns.

| When the expression is ... | The data type of the result column is ... |
|---|---|
| the name of a datetime column | the same as the data type of the column. |

## from-clause

```
     ┌─,──────────┐
►►──FROM──┴──table-reference──┴───────────────────────────►◄
```

The FROM clause specifies an intermediate result table.

If one table-reference is specified, the intermediate result table is simply 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-references (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 375.

## table-reference

```
►►─┬─────────┬─────nickname───────────────────────┬──────────────────────┬──►◄
   │         │     table-name                      │  correlation-clause  │
   │         │     view-name                       └──────────────────────┘
   │         ├──ONLY───┬───(───┬──table-name──┬───)─┤
   │         └──OUTER──┘       └──view-name───┘
   │
   ├──TABLE──(──function-name──(──────────────────────)────)──correlation-clause──┤
   │                            │      ┌──,──────┐    │
   │                            └──────▼─expression─┴─┘
   │
   ├──┬───────┬──(fullselect)──correlation-clause────────────────────────────────┤
   │  └─TABLE─┘
   └──joined-table─────────────────────────────────────────────────────────────┘
```

**correlation-clause:**
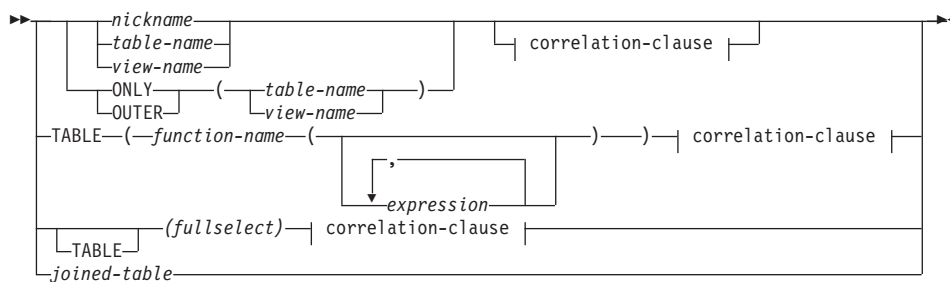
```
   ┌──AS──┐
├──┴──────┴──correlation-name──┬────────────────────────────────┬──┤
                               │    ┌──,────────┐               │
                               └─(──▼─column-name─┴──)──────────┘
```
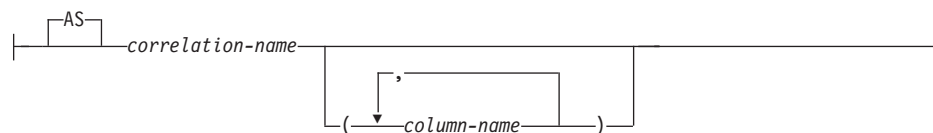
Each *table-name*, *view-name* or *nickname* specified as a table-reference must identify an existing table, view or nickname at the application server or the *table-name* of a common table expression (see "common-table-expression" on page 413) defined preceding the fullselect containing the table-reference. If the *table-name* references a typed table, the name denotes the UNION ALL of the table with all its subtables, with only the columns of the *table-name*. Similarly, if the *view-name* references a typed view, the name denotes the UNION ALL of the view with all its subviews, with only the columns of the *view-name*.

The use of ONLY(*table-name*) or ONLY(*view-name*) means that the rows of the proper subtables or subviews are not included. 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 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. 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.

The previous points also apply to OUTER(*view-name*), substituting *view-name* for *table-name* and subview for subtable.

The use of ONLY or OUTER requires the SELECT privilege on every subtable of *table-name* or subview of *view-name*.

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.

A fullselect in parentheses followed by a correlation name is called a *nested table expression*.

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 379.

The exposed names of all table references should 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*.

Each *correlation-name* is defined as a designator of the immediately preceding *table-name*, *view-name*, *nickname*, *function-name* reference or nested table expression. Any qualified reference to a column for a table, view, table function or nested table expression must use the exposed name. If the same table name, view or nickname name is specified twice, at least one specification should 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-name*, *view-name*, *nickname*, *function-name* reference or nested table expression. For more information, see "Correlation Names" on page 124.

In general, table functions and nested table expressions can be specified on any from-clause. Columns from the table functions and nested table expressions can be referenced in the select list and in the rest of the subselect using the correlation name which must be specified. The scope of this

correlation name is the same as correlation names for other table, view or nickname in the FROM clause. A nested table expression can be used:

- in place of a view to avoid creating the view (when general use of the view is not required)
- when the desired result table is based on host variables.

### Table Function References

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. There are, however, some special considerations which apply.

- Table Function Column Names

  Unless alternate column names are provided following the *correlation-name*, the column names for the table function are those specified in the RETURNS clause of the CREATE FUNCTION statement . This is analogous to the names of the columns of a table, which are of course defined in the CREATE TABLE statement. See "CREATE FUNCTION (External Table)" on page 570 for details about creating a table function.

- 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), used in a statement. Function resolution is covered in "Function Resolution" on page 139 .

- Table Function Arguments

  As with scalar function arguments, table function arguments can in general be any valid SQL expression. So 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
```

### 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 these cases is that the

## table-reference

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 TABLE keyword must appear before the
fullselect. So the following examples are valid syntax:

```
Example 1:  SELECT t.c1, z.c5
            FROM t, TABLE( tf3(t.c2) ) AS z      -- t precedes tf3 in FROM
            WHERE t.c3 = z.c4;                   -- so t.c2 is known

Example 2:  SELECT t.c1, z.c5
            FROM t, TABLE( tf4(2 * t.c2) ) AS z  -- t precedes tf3 in FROM
            WHERE t.c3 = z.c4;                   -- so t.c2 is known

Example 3:  SELECT d.deptno, d.deptname,
                   empinfo.avgsal, empinfo.empcount
            FROM department d,
                TABLE (SELECT AVG(e.salary) AS avgsal,
                              COUNT(*) AS empcount
                        FROM employee e           -- department precedes and
                        WHERE e.workdept=d.deptno -- TABLE is specified
                       ) AS empinfo;              -- 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 but
                        WHERE e.workdept=d.deptno -- TABLE is not specified
                       ) AS empinfo;              -- so d.deptno is unknown
```

## joined-table

```
►►──┬─table-reference─┬─────┬─INNER──┬──────────────┬──JOIN─table-reference─ON─join-condition─┬──────►◄
    │                 │     │        │              │                                         │
    │                 │     ├─LEFT───┤  ┌─OUTER─┐   │                                         │
    │                 │     ├─RIGHT──┤              │                                         │
    │                 │     └─FULL───┘              │                                         │
    │                                                                                         │
    └─(─joined-table─)────────────────────────────────────────────────────────────────────────┘
```
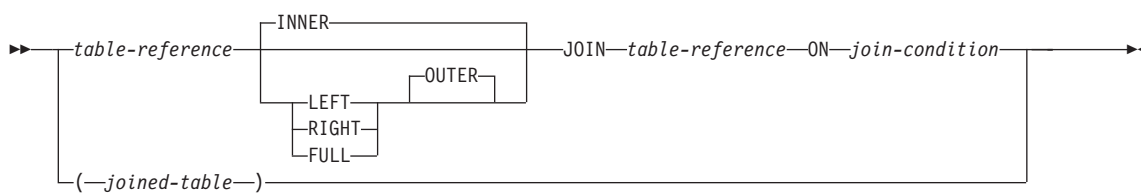
A *joined table* specifies an intermediate result table that is the result of either an inner join or an outer join. The table is derived by applying one of the join operators: INNER, LEFT OUTER, RIGHT OUTER, or FULL OUTER to its operands.

Inner joins can be thought of as the cross product of the tables (combine each row of the left table with every row of the right table), keeping only the rows where the join-condition is true. The result table may be missing rows from either or both of the joined tables. Outer joins include the inner join and preserve these missing rows. There are three types of outer joins:

1. *left outer join* includes rows from the left table that were missing from the inner join.

2. *right outer join* includes rows from the right table that were missing from the inner join.

3. *full outer join* includes rows from both the left and right tables that were missing from the inner join.

If a join-operator is not specified, INNER is implicit. 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. Parentheses are recommended to make the order of nested joins more readable. For example:

```
tb1 left join  tb2 on tb1.c1=tb2.c1
    right join tb3 left join tb4 on tb3.c1=tb4.c1
                                 on tb1.c1=tb3.c1
```

is the same as:

```
(tb1 left join tb2 on tb1.c1=tb2.c1)
    right join (tb3 left join tb4 on tb3.c1=tb4.c1)
                                 on tb1.c1=tb3.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.

A *join-condition* is a *search-condition* except that:

- 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.
- 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.

An error occurs if the join-condition does not comply with these rules (SQLSTATE 42972).

Column references are resolved using the rules for resolution of column name qualifiers. The same rules that apply to predicates apply to *join-condition*s (see "Predicates" on page 166).

### Join Operations

A *join-condition* specifies pairings of T1 and T2, where T1 and T2 are the left and right operand tables of the JOIN operator of the *join-condition*. 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. 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. The execution might involve the generation of a null row. The null row of a table consists of a null value for each column of the table, regardless of whether the columns allow null values.

The following summarizes the result of the join operations:
- 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. All columns derived from T2 allow null values.
- 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. All columns derived from T1 allow null values.
- 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. All columns derived from T1 and T2 allow null values.

## where-clause

►►──WHERE──*search-condition*──────────────────────────────────────────►◄

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.
- A column 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 is executed just once, whereas a subquery with a correlated reference may have to be executed once for each row.

## group-by-clause

```
>>--GROUP BY-----,-----------------------------------------><
                 |  grouping-expression  |
                 |  grouping-sets        |
                 |  super-groups         |
```
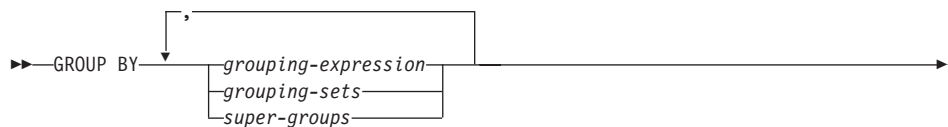
The GROUP BY clause specifies an intermediate result table that consists of a grouping of the rows of R. R is the result of the previous clause of the subselect.

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 *column-name* included in *grouping-expression* must unambiguously identify a column of R (SQLSTATE 42702 or 42703). The length attribute of each *grouping-expression* must not be more than 254 bytes (SQLSTATE 42907). A *grouping-expression* cannot include a scalar-fullselect (SQLSTATE 42822) or any function that is variant or has an external action (SQLSTATE 42845).

More complex forms of the GROUP BY clause include *grouping-sets* and *super-groups*. For a description of these forms, see "grouping-sets" on page 383 and "super-groups" on page 384, respectively.

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.

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 416 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 would be *col1+col2+3*. Associativity rules for expressions would 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)* would also be 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 may not all have the same length. In that case, a reference to the *grouping-expression*

still specifies only one value for each group, but the value for a group is chosen arbitrarily 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, variant 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 A9" on page 393.

### grouping-sets
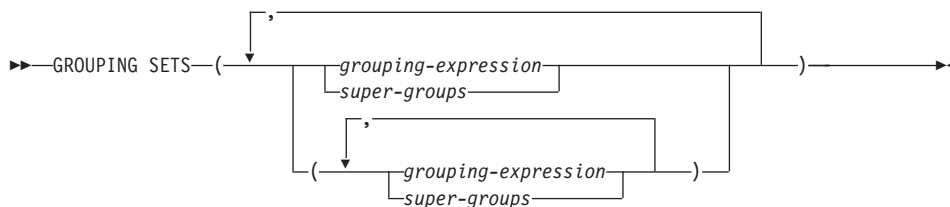


A *grouping-sets* specification allows multiple grouping clauses to be specified in a single statement. This can be thought of as the union of two or more groups of rows into a single result set. It is logically equivalent to the union of multiple subselects with the group by clause in each subselect 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. Using *grouping-sets* allows the groups to be computed with a single pass over the base table.

The *grouping-sets* specification allows either a simple *grouping-expression* to be used, or the more complex forms of *super-groups*. For a description of *super-groups*, see "super-groups" on page 384.

Note that grouping sets are the fundamental building block 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 SET((a))
```

and

```
GROUP BY a,b,c
```

## group-by-clause

is the same as

```
GROUP BY GROUPING SET((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. See "GROUPING" on page 214 for how to distinguish rows with nulls in actual data from rows with nulls generated from grouping sets.

"Example C2" on page 399 through "Example C7" on page 403 illustrate the use of grouping sets.

### super-groups
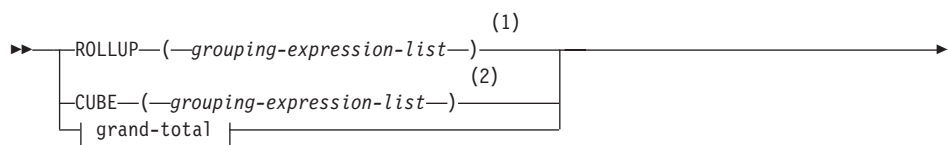
```
                                                    (1)
►►─┬─ROLLUP──(─grouping-expression-list──)──────────┬────────────────────►◄
   │                                      (2)        │
   ├─CUBE──(─grouping-expression-list──)─────────────┤
   └─| grand-total |────────────────────────────────┘
```

### grouping-expression-list:

```
          ┌──,──────────────────┐
          ▼                      │
├─────────┬───grouping-expression─┴──────────────────┬──────────────────────┤
          │        ┌──,──────────────────┐           │
          │        ▼                      │           │
          └─(───────grouping-expression───┴───)───────┘
```
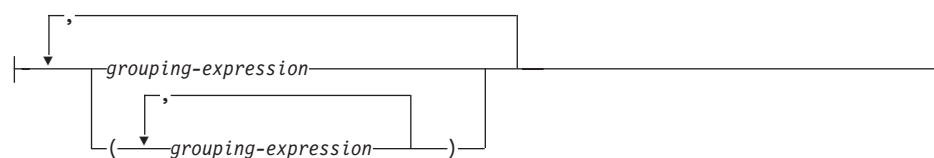
### grand-total:

```
├───(──)─────────────────────────────────────────────────────────────────┤
```

### 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 that contains *sub-total* rows in addition to the

"regular" grouped rows. *Sub-total* rows[49]are "super-aggregate" rows that contain further aggregates whose values are derived by applying the same column functions that were used to obtain the grouped rows.

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)
                       () )
```

Notice that the *n* elements of the ROLLUP translate to *n*+1 grouping sets.

Note that the order in which the *grouping-expressions* is specified is significant for ROLLUP. For example:

```
GROUP BY ROLLUP(a,b)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b)
                       (a)
                       () )
```

while

```
GROUP BY ROLLUP(b,a)
```

is the same as

```
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 C3" on page 399 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.

---

49. These are called sub-total rows, because that is their most common use, however any column function can be used for the aggregation. For instance, MAX and AVG are used in "Example C8" on page 405.

**group-by-clause**

Like 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 instance, 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)
                       () )
```

Notice that the 3 elements of the CUBE translate to 8 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. "Example C4" on page 400 illustrates the use of CUBE.

*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.

The rules for a *grouping-expression* are described in "group-by-clause" on page 382. 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 sub-total 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 desired result. In other words, the two element ROLLUP

```
GROUP BY ROLLUP(Province, (County, City))
```

generates

```
GROUP BY GROUPING SETS((Province, County, City)
                       (Province)
                       () )
```

while the 3 element ROLLUP would generate

```
GROUP BY GROUPING SETS((Province, County, City)
                       (Province, County)
                       (Province)
                       () )
```

"Example C2" on page 399 also utilizes composite column values.

**grand-total**
> Both CUBE and ROLLUP return a row which is the overall (grand total) aggregation. This may be separately specified with empty parentheses within the GROUPING SET clause. It may also be specified directly in the GROUP BY clause, although there is no effect on the result of the query. "Example C4" on page 400 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 like "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)
```

**group-by-clause**

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)
                       () )
```

Like 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 would 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 simply grouped, those within the parenthesis following ROLLUP are rolled up, and those within the parenthesis following CUBE are cubed. Thus, the above 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) )
```

## having-clause

```
►►──HAVING──search-condition──────────────────────────────────────►◄
```

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 a single group with no grouping columns.

Each *column-name* in the search condition must do one of the following:
- Unambiguously identify a grouping column of R.
- Be specified within a column 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 column 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 A6" on page 392 and "Example A7" on page 392.

A correlated reference to a group of R must either identify a grouping column or be contained within a column function.

When HAVING is used without GROUP BY, the select list can only be a column name within a column function, a correlated column reference, a literal, or a special register.

## Examples of subselects

*Example A1:* Select all columns and rows from the EMPLOYEE table.

```
SELECT * FROM EMPLOYEE
```

*Example A2:* 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 A3:* 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 1930.

```
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
    FROM EMPLOYEE, DEPARTMENT
    WHERE WORKDEPT = DEPTNO
    AND YEAR(BIRTHDATE) < 1930
```

*Example A4:* 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 A5:* 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')
```

## Examples of subselects

*Example A6:*  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 would only be executed once in this example.

*Example A7:*  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 A6", the subquery in the HAVING clause would need to be executed for each group.

*Example A8:*  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) in order to get the AVGSALARY and EMPCOUNT columns, as well as 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 overhead 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 query, only the rows for the department of the sales representatives need to be considered by the view.

*Example A9:* Display the average education level and salary for 5 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 subsequently 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
```

## Examples of Joins

*Example B1:* 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 B2:*   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 B1" on page 394.

Notice what the impact 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) since 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.

**Examples of Joins**

```
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 would be the same as the result of the full outer join query in "Example B1" on page 394. 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 significant impact 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 would return 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 B3:* 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 B4:* List every employee number and last name with the employee number and last name of their manager, including employees without a manager.

**Examples of Joins**

```
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.

## Examples of Grouping Sets, Cube, and Rollup

The queries in "Example C1" through "Example C4" on page 400 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 C1:*  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 C2:*  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 C3:*  If you use the 3 distinct columns involved in the grouping sets of "Example C2" 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 C4:*  If you run the same query as "Example C3" on page 399 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
```

**Examples of Grouping Sets, Cube, and Rollup**

*Example C5:* 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 C6:* 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 C6-1:*

```
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
```

# Examples of Grouping Sets, Cube, and Rollup

```
14            -          74
53            1           8
53            -           8
 -            -         155
```

*Example C6-2:*

```
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 C6-3:*

```
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 appear in alphabetic order.
- Null values are sorted high.

*Example C7:* In queries that perform multiple ROLLUPs in a single pass (such as "Example C6-3" on page 402) you may 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. By origin, we mean 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 alternate 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
```

## Examples of Grouping Sets, Cube, and Rollup

*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 we can combine these columns 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 |

## Examples of Grouping Sets, Cube, and Rollup

*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 C8:* The following example illustrates the use of various column 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,
```

## Examples of Grouping Sets, Cube, and Rollup

```
          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
```

## fullselect

```
>>--+--subselect-------+--+----------------------------------------------+--><
    +--(fullselect)---+  |  +-UNION--------+--+--subselect-------+      |
    +--values-clause--+  |  +-UNION ALL----+  +--(fullselect)---+      |
                         |  +-EXCEPT-------+  +--values-clause--+      |
                         +-----------------+-EXCEPT ALL---+-------------+
                                           +-INTERSECT----+
                                           +-INTERSECT ALL-+
```

**values-clause:**

```
|--VALUES--+--values-row--+--------------------------------------|
           +------,-------+
```

**values-row:**

```
|--+--+-expression-+--------------------------------------------|
   |  +-NULL-------+                                            |
   |                +----,-----+                                |
   +--(--+-expression-+-----)--+                                |
         +-NULL-------+                                         |
```
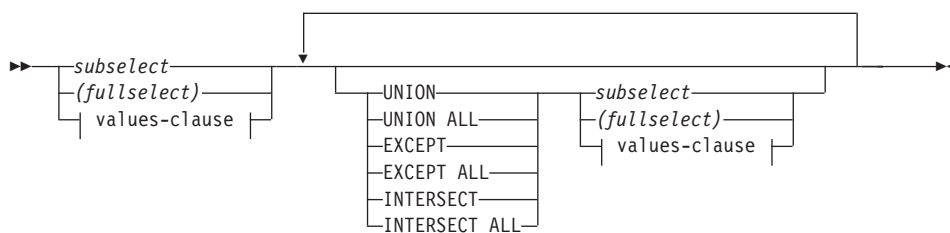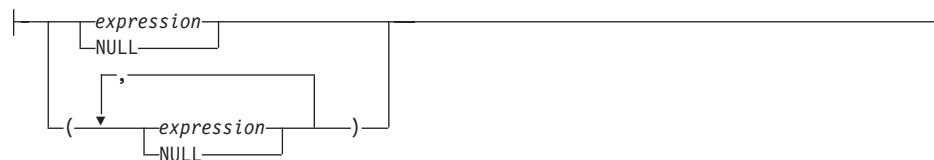
The *fullselect* is a component of the select-statement, the INSERT statement, and the CREATE VIEW statement. It is also a component of certain predicates which, in turn are components of a statement. A *fullselect* that is a component of a predicate is called a *subquery*. A *fullselect* that is enclosed in parentheses is sometimes called a subquery.

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.

**values-clause**
    Derives a result table by specifying the actual values, using expressions, for each column of a row in the result table. Multiple rows may be specified.

    NULL can only be used with multiple *values-row*s and at least one row in the same column must not be NULL (SQLSTATE 42826).

A *values-row* is specified by:
- A single expression for a single column result table or,
- n expressions (or NULL) separated by commas and enclosed in parentheses, where n is the number of columns in the result table.

A multiple row VALUES clause must have the same number of expressions in each *values-row* (SQLSTATE 42826).

The following are examples of values-clauses 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 *values-row*s, $RE_1$ to $RE_n$, where n is greater than 1, is equivalent to

```
RE₁ UNION ALL RE₂ ... UNION ALL REₙ
```

This means that the corresponding expressions of each *values-row* must be comparable (SQLSTATE 42825) and the resulting data type is based on "Rules for Result Data Types" on page 105.

**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.

**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 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 string columns declared larger than 254 bytes or having a data type of LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB or DBCLOB (SQLSTATE 42907).

The columns of the result are named as follows:

- If the *n*th column of R1 and the *n*th column of R2 have the same result column name, then the *n*th column of R has the result column name.
- If the *n*th column of R1 and the *n*th column of R2 have different result column names, a name is generated. This name cannot be used as the column name in an ORDER BY or UPDATE clause.

The generated name can be determined by performing a DESCRIBE of the SQL statement and consulting the SQLNAME field.

Two rows are duplicates of one another if each value in the first is equal to the corresponding value of the second. (For determining duplicates, two null values are considered equal.)

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 |   |   |   |   |   |

| R1 | R2 | UNION ALL | UNION | EXCEPT ALL | EXCEPT | INTER- SECT ALL | INTER- SECT |
|----|----|-----------|-------|------------|--------|-----------------|-------------|
|    |    | 3         |       |            |        |                 |             |
|    |    | 3         |       |            |        |                 |             |
|    |    | 4         |       |            |        |                 |             |
|    |    | 4         |       |            |        |                 |             |
|    |    | 4         |       |            |        |                 |             |
|    |    | 5         |       |            |        |                 |             |

For the rules on how the data types of the result columns are determined, see "Rules for Result Data Types" on page 105.

For the rules on how conversions of string columns are handled, see "Rules for String Conversions" on page 108.

### Examples of a fullselect

*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 may 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

```
>>─┬──────────────────────────────────────┬──fullselect──────────────────>
   │          ┌─,───────────────────────┐ │
   └─WITH───▼──common-table-expression──┴─┘


>──┬─────────────────┬──┬───────────────────┬──●──┬─────────────────────┬──>
   └─order-by-clause─┘  └─fetch-first-clause─┘     ├─read-only-clause────┤
                                                   │            (1)      │
                                                   └─update-clause───────┘


>──●──┬─────────────────────┬──●─────────────────────────────────────────>◄
      └─optimize-for-clause─┘
```
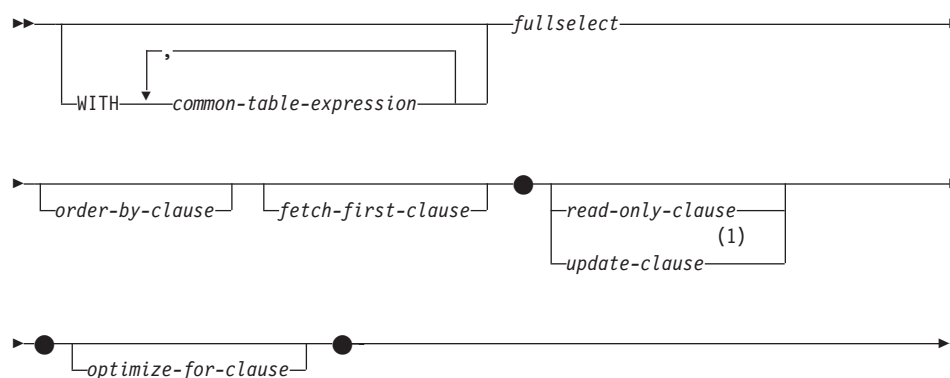
**Notes:**

**1**    The update-clause and the order-by-clause cannot both be specified in the same select-statement.

The *select-statement* is the form of a query that can be directly specified in a DECLARE CURSOR statement, or prepared and then referenced in a DECLARE CURSOR statement. It can also be issued through the use of dynamic SQL statements using the command line processor (or similar tools), causing a result table to be displayed on the user's screen. In either case, the table specified by a *select-statement* is the result of the fullselect.

A *select-statement* that references a nickname cannot be specified directly in the DECLARE CURSOR statement.

## common-table-expression

```
►►──table-name─────────────────────────────────AS──(──fullselect──)──────────►◄
               │         ,                     │(1)
               │    ┌────◄────┐                │
               └─(──┴─────────┴──)─────────────┘
                     │          │
                     └column-name┘
```
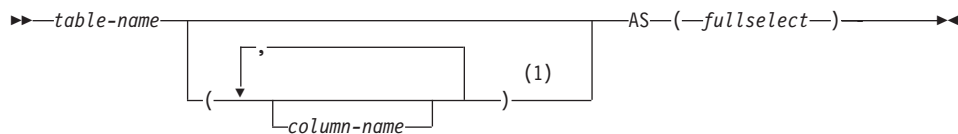
**Notes:**

**1**    If a common table expression is recursive, or if the fullselect results in duplicate column names, column names must be specified.

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. 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*.

The *common table expression* is also optional prior to the fullselect in the CREATE VIEW and INSERT statements.

A *common table expression* can be used:

## common-table-expression

- 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 desired result table is based on host variables
- When the same result table needs to be shared in a *fullselect*
- When the result needs to be derived using recursion.

If a *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. For an example, see "Appendix M. Recursion Example: Bill of Materials" on page 1169.

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 may differ. In this case, the column in the iterative fullselect must have a length that would always be assignable to the length determined from the initialization fullselect.
- Each fullselect that is part of the recursion cycle must not include any column 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).
- 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

terminate. 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).

## order-by-clause

```
           ┌─────,──────────┐
           ▼                │   ┌──ASC──┐
►►──ORDER BY───sort-key─────┴───┤       ├────────────────────►◄
                                └──DESC─┘
```

**sort-key:**

```
├──┬──simple-column-name──┬─────────────────────────────────┤
   ├──simple-integer──────┤
   └──sort-key-expression─┘
```

The ORDER BY clause specifies an ordering of the rows of the result table. 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. The length attribute of each *sort-key* must not be more
than 254 characters for a character column or 127 characters for a graphic
column (SQLSTATE 42907).

A named column in the select list may 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. [50]

Ordering is performed in accordance with the comparison rules described in
Chapter 3.. 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* may also identify a column name of a table, view
> or nested table identified in the FROM clause if the query is a subselect.
> An error occurs if the subselect:

---

50. The rules for determining the name of result columns for a fullselect that involves set operators (UNION,
   INTERSECT, or EXCEPT) can be found in "fullselect" on page 407.

- specifies DISTINCT in the select-clause (SQLSTATE 42822)
- 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 name in sort keys″ (see "Notes").

*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″ (see "Notes").

  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 a column function, constant or host variable.

**ASC**
  Uses the values of the column in ascending order. This is the default.

**DESC**
  Uses the values of the column in descending order.

**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.

  See "Column Name Qualifiers to Avoid Ambiguity" on page 126 for more information on qualified column names.

- **Limits**: The use of a *sort-key-expression* or a *simple-column-name* where the column is not in the select list may result in the addition of the column or expression to the temporary table used for sorting. This may 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.

## update-clause

```
►►──FOR UPDATE─────────────────────────────────────────────────────►◄
               │                    ┌─ , ◄──┐                 │
               │                    ▼        │                │
               └─OF───────column-name────────┘
```
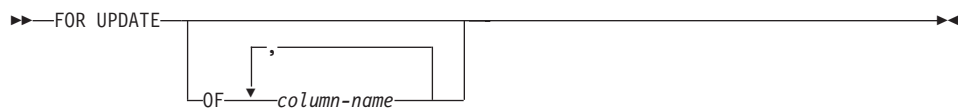
The FOR UPDATE clause identifies the columns that can be updated in a
subsequent Positioned UPDATE statement. Each *column-name* must be
unqualified and must identify a column of the table or view identified in the
first FROM clause of the fullselect. If the FOR UPDATE clause is specified
without column names, all 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 is true:
- The cursor associated with the select-statement is not deletable (see "Notes"
  on page 733).

- 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

```
►►──FOR──┬─READ──┬──ONLY────────────────────────────────────────►◄
         └─FETCH─┘
```

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 and avoid exclusive locks. 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 was 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 a 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). See "DECLARE CURSOR" on page 731 for more information about read-only and updatable cursors.

## fetch-first-clause

```
►►─FETCH FIRST─┬───1───┬─┬─ROW──┬─ONLY──────────────────────────►◄
               └─integer─┘ └─ROWS─┘
```
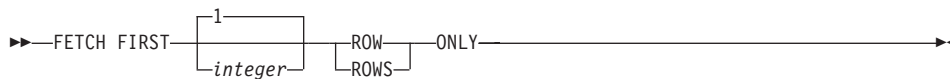
The *fetch-first-clause* sets a maximum number of rows that can be retrieved. It lets the database manager know that the application does not want to retrieve more than *integer* rows, regardless of how many rows there might be in the result table when this clause is not specified. An attempt to fetch beyond *integer* rows is handled the same way as normal end of data (SQLSTATE 02000, SQLCODE +100). The value of *integer* must be a positive integer (not zero).

Limiting the result table to the first *integer* rows can improve performance. The database manager will cease processing the query once it has determined the first *integer* rows. If both the *fetch-first-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.

Specification of the fetch-first-clause in a select-statement makes the cursor not deletable (read-only). This clause cannot be specified with the FOR UPDATE clause.

## optimize-for-clause

```
►►──OPTIMIZE FOR──integer──┬─ROWS─┬──────────────────────────────────────────►◄
                           └─ROW──┘
```

The OPTIMIZE FOR clause requests special processing of the *select statement*. If the 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 for *integer*. The value of *n* must be a positive integer. 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 (ie. no more than *n* rows will be returned in each block). If both the *fetch-first-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 the performance if no more than *n* rows are retrieved, but may 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[51] the OPTIMIZE FOR clause will have no impact on the data buffers.

---

51. The size of the communication buffer is defined by the RQRIOBLK or the ASLHEAPSZ configuration parameter. See the *Administration Guide* for details.

## Examples of a select-statement

*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 overhead 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 need to be considered by the view.

## Examples of a select-statement

```
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
```

# Chapter 6. Statements

This chapter contains syntax diagrams, semantic descriptions, rules, and examples of the use of the SQL statements.

*Table 18. SQL Statements*

| SQL Statement | Function | Page |
|---|---|---|
| ALTER BUFFERPOOL | Changes the definition of a buffer pool. | 435 |
| ALTER NICKNAME | Changes the definition of a nickname. | 437 |
| ALTER NODEGROUP | Changes the definition of a nodegroup. | 441 |
| ALTER SERVER | Changes the definition of a server. | 445 |
| ALTER TABLE | Changes the definition of a table. | 449 |
| ALTER TABLESPACE | Changes the definition of a table space. | 471 |
| ALTER TYPE (Structured) | Changes the definition of a structured type. | 475 |
| ALTER USER MAPPING | Changes the definition of a user authorization mapping. | 477 |
| ALTER VIEW | Changes the definition of a view by altering a reference type column to add a scope. | 480 |
| BEGIN DECLARE SECTION | Marks the beginning of a host variable declaration section. | 482 |
| CALL | Calls a stored procedure. | 484 |
| CLOSE | Closes a cursor. | 492 |
| COMMENT ON | Replaces or adds a comment to the description of an object. | 494 |
| COMMIT | Terminates a unit of work and commits the database changes made by that unit of work. | 505 |
| Compound SQL | Combines one or more other SQL statements into an executable block. | 507 |
| CONNECT (Type 1) | Connects to an application server according to the rules for remote unit of work. | 512 |
| CONNECT (Type 2) | Connects to an application server according to the rules for application-directed distributed unit of work. | 520 |
| CREATE ALIAS | Defines an alias for a table, view, or another alias. | 528 |
| CREATE BUFFERPOOL | Creates a new buffer pool. | 531 |
| CREATE DISTINCT TYPE | Defines a distinct data type. | 534 |
| CREATE EVENT MONITOR | Specifies events in the database to monitor. | 541 |
| CREATE FUNCTION | Registers a user-defined function. | 551 |
| CREATE FUNCTION (External Scalar) | Registers a user-defined external scalar function. | 552 |

*Table 18. SQL Statements (continued)*

| SQL Statement | Function | Page |
|---|---|---|
| GRANT (Index Privileges) | Grants the CONTROL privilege on indexes in the database. | 791 |
| GRANT (Package Privileges) | Grants privileges on packages in the database. | 793 |
| GRANT (Schema Privileges) | Grants privileges on a schema. | 796 |
| GRANT (Server Privileges) | Grants privileges to query a specific data source. | 799 |
| GRANT (Table, View, or Nickname Privileges) | Grants privileges on tables, views and nicknames. | 801 |
| INCLUDE | Inserts code or declarations into a source program. | 809 |
| INSERT | Inserts one or more rows into a table. | 811 |
| LOCK TABLE | Either prevents concurrent processes from changing a table or prevents concurrent processes from using a table. | 817 |
| OPEN | Prepares a cursor that will be used to retrieve values when the FETCH statement is issued. | 819 |
| PREPARE | Prepares an SQL statement (with optional parameters) for execution. | 824 |
| REFRESH TABLE | Refreshes the data in a summary table. | 834 |
| RELEASE | Places one or more connections in the release-pending state. | 835 |
| RENAME TABLE | Renames an existing table. | 838 |
| REVOKE (Database Authorities) | Revokes authorities from the entire database. | 840 |
| REVOKE (Index Privileges) | Revokes the CONTROL privilege on given indexes. | 843 |
| REVOKE (Package Privileges) | Revokes privileges from given packages in the database. | 845 |
| REVOKE (Schema Privileges) | Revokes privileges on a schema. | 848 |
| REVOKE (Server Privileges) | Revokes privileges to query a specific data source. | 850 |
| REVOKE (Table, View, or Nickname Privileges) | Revokes privileges from given tables, views or nicknames. | 852 |
| ROLLBACK | Terminates a unit of work and backs out the database changes made by that unit of work. | 858 |
| SELECT INTO | Specifies a result table of no more than one row and assigns the values to host variables. | 861 |
| SET CONNECTION | Changes the state of a connection from dormant to current, making the specified location the current server. | 863 |
| SET CURRENT DEGREE | Changes the value of the CURRENT DEGREE special register. | 865 |
| SET CURRENT EXPLAIN MODE | Changes the value of the CURRENT EXPLAIN MODE special register. | 867 |
| SET CURRENT EXPLAIN SNAPSHOT | Changes the value of the CURRENT EXPLAIN SNAPSHOT special register. | 870 |

*Table 18. SQL Statements (continued)*

| SQL Statement | Function | Page |
|---|---|---|
| SET CURRENT PACKAGESET | Sets the schema name for package selection. | 872 |
| SET CURRENT QUERY OPTIMIZATION | Changes the value of the CURRENT QUERY OPTIMIZATION special register. | 874 |
| SET CURRENT REFRESH AGE | Changes the value of the CURRENT REFRESH AGE special register. | 877 |
| SET EVENT MONITOR STATE | Activates or deactivates an event monitor. | 879 |
| SET INTEGRITY | Sets the check pending state and checks data for constraint violations. | 881 |
| SET PASSTHRU | Opens a session for submitting a data source's native SQL directly to the data source. | 890 |
| SET PATH | Changes the value of the CURRENT PATH special register. | 892 |
| SET SCHEMA | Changes the value of the CURRENT SCHEMA special register. | 895 |
| SET SERVER OPTION | Sets server option settings. | 897 |
| SET transition-variable | Assigns values to NEW transition variables. | 899 |
| SIGNAL SQLSTATE | Signals an error. | 903 |
| UPDATE | Updates the values of one or more columns in one or more rows of a table. | 905 |
| VALUES INTO | Specifies a result table of no more than one row and assigns the values to host variables. | 914 |
| WHENEVER | Defines actions to be taken on the basis of SQL return codes. | 916 |

## How SQL Statements Are Invoked

The SQL statements described in this chapter are classified as *executable* or *nonexecutable*. The *Invocation* section in the description of each statement indicates whether or not the statement is executable.

An *executable statement* can be invoked in three ways:
- Embedded in an application program
- Dynamically prepared and executed
- Issued interactively.

**Note:** Statements embedded in REXX are prepared and executed dynamically.

Depending on the statement, some or all of these methods can be used. The *Invocation* section in the description of each statement tells which methods can be used.

A *nonexecutable statement* can only be embedded in an application program.

In addition to the statements described in this chapter, there is one more SQL statement construct: the *select-statement*. (See "select-statement" on page 412.) It is not included in this chapter because it is used differently from other statements.

A *select-statement* can be invoked in three ways:

- Included in DECLARE CURSOR and implicitly executed by OPEN, FETCH and CLOSE
- Dynamically prepared, referenced in DECLARE CURSOR, and implicitly executed by OPEN, FETCH and CLOSE
- Issued interactively.

The first two methods are called, respectively, the *static* and the *dynamic* invocation of *select-statement*.

The different methods of invoking an SQL statement are discussed below in more detail. For each method, the discussion includes the mechanism of execution, interaction with host variables, and testing whether or not the execution was successful.

## Embedding a Statement in an Application Program

SQL statements can be included in a source program that will be submitted to the 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 and SQL.

### Executable statements

An executable statement embedded in an application program is executed every time a statement of the host language would be executed if 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:

- 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. The treatment of other references is described individually for each statement.

All executable statements should be followed by a test of an SQL return code. Alternatively, the WHENEVER statement (which is itself nonexecutable) can be used to change the flow of control immediately after the execution of an embedded statement.

All objects referenced in DML statements must exist when the statements are bound to a DB2 Universal Database.

### Nonexecutable statements

An embedded nonexecutable 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 an SQL return code.

## 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 (other than a select-statement) so constructed can be prepared for execution by means of the (embedded) statement PREPARE and executed by means of the (embedded) statement EXECUTE. Alternatively, the (embedded) statement EXECUTE IMMEDIATE can be used to prepare and execute a 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. (See "PREPARE" on page 824 for rules concerning the parameter markers.) When the prepared statement is executed, the parameter markers are effectively replaced by current values of the host variables specified in the EXECUTE statement. (See "EXECUTE" on page 770 for rules concerning this replacement.) Once prepared, a statement can be executed several times with different values of host variables. Parameter markers are not allowed in EXECUTE IMMEDIATE.

The 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. The SQL return code should be checked as described above. See "SQL Return Codes" on page 432 for more information.

## Static Invocation of a select-statement

A *select-statement* can be included as a part of the (nonexecutable) statement DECLARE CURSOR. Such a statement is executed every time the cursor is opened by means of the (embedded) statement OPEN. 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 at the moment of executing OPEN.

## 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) statement PREPARE, and referenced by a (nonexecutable) statement DECLARE CURSOR. The statement is then executed every time the cursor is opened by means of the (embedded) statement OPEN. 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. (See "PREPARE" on page 824 for rules concerning the parameter markers.) The parameter markers are effectively replaced by the values of the host variables specified in the OPEN statement. (See "OPEN" on page 819 for rules concerning this replacement.)

## 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.

A statement issued interactively 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 Return Codes

An application program containing executable SQL statements can use either the 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. An SQLCA is provided automatically in REXX. In other languages, an SQLCA can be obtained by using the INCLUDE SQLCA statement.

  The SQLCA includes an integer variable named SQLCODE and and a character string variable named SQLSTATE.

- When LANGLEVEL SQL92E is specified as a precompile option, a variable SQLCODE or SQLSTATE may 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 SQLCODE is declared elsewhere in the program. When using LANGLEVEL SQL92E, the program should not have an INCLUDE SQLCA statement.

Occasionally, warning conditions are mentioned in addition to error conditions with respect to return codes. A warning SQLCODE is a positive value and a warning SQLSTATE has the first two characters set to '01'.

### SQLCODE

An SQLCODE is set by the database manager after each SQL statement is executed. 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, however, one or more warning indicators were set. Refer to "Appendix B. SQL Communications (SQLCA)" on page 927 for more details.

- If SQLCODE < 0, execution was not successful.

The meaning of SQLCODE values other than 0 and 100 is product-specific. See the *Message Reference* for the product-specific meanings.

### SQLSTATE

SQLSTATE is also set by the database manager after execution of each SQL statement. Thus, application programs can check the execution of SQL statements by testing SQLSTATE instead of SQLCODE.

SQLSTATE provides application programs with common codes for common error conditions. Furthermore, SQLSTATE is designed so that 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. For a complete list of the possible values of SQLSTATE, see the *Message Reference*.

## SQL Comments

Static SQL statements can include host language or SQL comments. SQL comments are introduced by two hyphens.

These rules apply to the use of SQL comments:

- The two hyphens must be on the same line, not separated by a space.
- Comments can be started wherever a space is valid (except within a delimiter token or between 'EXEC' and 'SQL').
- Comments are terminated by the end of the line.
- Comments are not allowed within statements that are dynamically prepared (using PREPARE or EXECUTE IMMEDIATE).
- In COBOL, the hyphens must be preceded by a space.

*Example:*  This example shows how to include comments in an SQL statement within a C program:

```
EXEC SQL
  CREATE VIEW PRJ_MAXPER        -- projects with most support personnel
    AS SELECT PROJNO, PROJNAME  -- number and name of project
        FROM PROJECT
        WHEREDEPTNO = 'E21'     -- systems support dept code
        AND PRSTAFF > 1;
```

## ALTER BUFFERPOOL

The ALTER BUFFERPOOL statement is used to do the following:
- modify the size of the buffer pool on all partitions (or nodes) or on a single partition
- turn on or off the use of extended storage
- add this buffer pool definition to a new nodegroup.

### Invocation

This statement can be embedded in an application program or issued interactively. 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 authorization ID of the statement must have SYSCTRL or SYSADM authority.

### Syntax

```
►►──ALTER──BUFFERPOOL──bufferpool-name──────────────────────────────────►

►──┬──────────────────────┬──┬─SIZE─number-of-pages─┬──────────────────►◄
   │  ┌─NODE─node-number─┐ │  └──────────────────────┘
   │  ├─NOT EXTENDED STORAGE─┤
   │  └─EXTENDED STORAGE─────┘
   └─ADD NODEGROUP─nodegroup-name─┘
```

### 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.

**NODE** *node-number*
> Specifies the partition on which size of the buffer pool is modified. The partition must be in one of the nodegroups for the buffer pool (SQLSTATE 42729). If this clause is not specified, then the size of the buffer pool is modified on all partitions on which the buffer pool exists that used the default size for the buffer pool (did not have a size specified in the *except-on-nodes-clause* of the CREATE buffer pool statement).

## ALTER BUFFERPOOL

**SIZE** *number-of-pages*
> The size of the buffer pool specified as the number of pages.[52]

**EXTENDED STORAGE**
> If the extended storage configuration is turned on [53], pages that are being migrated out of this buffer pool, will be cached in the extended storage.

**NOT EXTENDED STORAGE**
> Even if the extended storage configuration is turned on, pages that are being migrated out of this buffer pool, will NOT be cached in the extended storage.

**ADD NODEGROUP** *nodegroup-name*
> Adds this nodegroup to the list of nodegroups to which the buffer pool definition is applicable. Tables spaces in *nodegroup-name* may specify this buffer pool. The nodegroup must currently exist in the database (SQLSTATE 42704).

### Notes

- 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 nodegroups will use the default buffer pool.

- 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.

---

52. The size can be specified with a value of (-1) which will indicate that the buffer pool size should be taken from the BUFFPAGE database configuration parameter.

53. Extended storage configuration is turned on by setting the database configuration parameters NUM_ESTORE_SEGS and ESTORE_SEG_SIZE to non-zero values. See *Administration Guide* for details.

## ALTER NICKNAME

The ALTER NICKNAME statement modifies the federated database's representation of a data source table or view by:
- Changing the local names of the table's or view's columns
- Changing the local data types of these columns
- Adding, changing, or deleting options for these 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. 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 at least one of the following:
- SYSADM or DBADM authority
- 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
- Definer of the nickname as recorded in the DEFINER column of the catalog view for the nickname

### Syntax

►►──ALTER NICKNAME──*nickname*────────────────────────────────────►

►──┬──────────────────────────────────────────────────────────────┬──►◄
   │      ┌─COLUMN─┐                   ┌─,──────────────────────────┐
   └─ALTER─┴────────┴──*column-name*──┬─LOCAL NAME──*column-name*───┤
                                      ├─LOCAL TYPE──*data-type*─────┤
                                      │                         (1) │
                                      └─┤ federated-column-options ├─┘
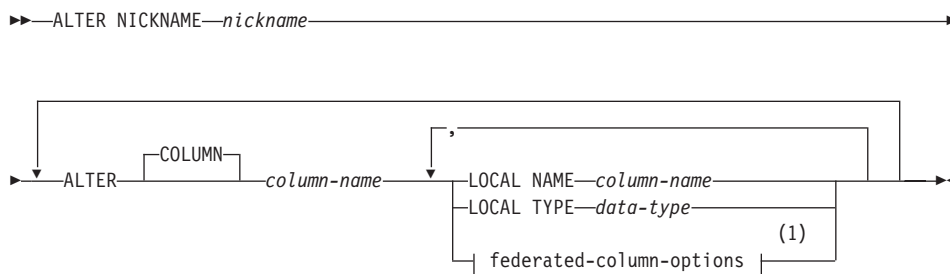
**federated-column-options:**

## ALTER NICKNAME
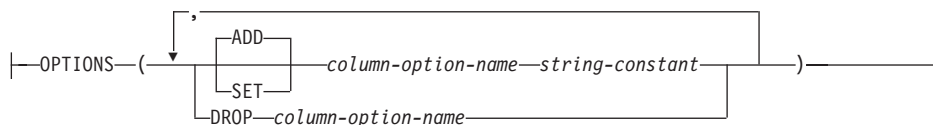
**Notes:**

**1** If the user needs to specify the federated-column-options clause in addition to the LOCAL NAME parameter, the LOCAL TYPE parameter, or both, the user must specify the federated-column-options clause last.

## Description

*nickname*
> Identifies the nickname for the data source table or view that contains the column specified after the COLUMN keyword. It must be a nickname described in the catalog.

**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 data source table or view referenced by *nickname*.

**LOCAL NAME** *column-name*
> Is the new name by which the federated server is to reference the column identified by the ALTER COLUMN *column-name* parameter. This new name must be a valid DB2 identifier.

**LOCAL TYPE** *data-type*
> Maps the specified column's data type to a local data type other than the one that it maps to now. The new type is denoted by *data-type*.
>
> The *data-type* cannot be LONG VARCHAR, LONG VARGRAPHIC, DATALINK, a large object (LOB) data type, or a user-defined type.

**OPTIONS**
> Indicates what column options are to be enabled, reset, or dropped for the column specified after the COLUMN keyword. Refer to "Column Options" on page 1047 for descriptions of *column-option-name*s and their settings.

> **ADD**
> > Enables a column option.

> **SET**
> > Changes the setting of a column option.

*column-option-name*
> Names a column option that is to be enabled or reset.

*string-constant*
> Specifies the setting for *column-option-name* as a character string constant.

**DROP** *column-option-name*
> Drops a column option.

## Rules

- If a view has been created on a nickname, the ALTER NICKNAME statement cannot be used to change the local names or data types for the columns in the table or view that the nickname references (SQLSTATE 42601). The statement can be used, however, to enable, reset, or drop column options for these columns.

## Notes

- If ALTER NICKNAME is used to change the local name for a column in a table or view that a nickname references, queries of the column must reference it by its new name.
- 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.
- 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.
- The ALTER NICKNAME statement cannot be processed within a unit of work (UOW) if the nickname referenced in this statement is already referenced by a SELECT statement in the same UOW (SQLSTATE 55007).

## Examples

*Example 1:* The nickname NICK1 references a DB2 Universal Database for AS/400 table called T1. Also, COL1 is the local name that references this table's first column, C1. Change the local name for C1 to NEWCOL.

```
ALTER NICKNAME NICK1
    ALTER COLUMN COL1
    LOCAL NAME NEWCOL
```

*Example 2:* The nickname EMPLOYEE references a DB2 Universal Database for OS/390 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

```
ALTER NICKNAME EMPLOYEE
   ALTER COLUMN SALARY
   LOCAL TYPE DECIMAL(10,5)
```

*Example 3:* Indicate that in an Oracle table, a column with the data type of
VARCHAR doesn't 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' )
```

## ALTER NODEGROUP

The ALTER NODEGROUP statement is used to:
- add one or more partitions or nodes to a nodegroup
- drop one or more partitions from a nodegroup.

### Invocation

This statement can be embedded in an application program or issued interactively. 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 authorization ID of the statement must have SYSCTRL or SYSADM authority.

### Syntax



**nodes-clause:**



### Description

*nodegroup-name*
    Names the nodegroup. This is a one-part name. It is an SQL identifier (either ordinary or delimited). It must be a nodegroup described in the catalog. IBMCATGROUP and IBMTEMPGROUP cannot be specified (SQLSTATE 42832).

**ADD NODE**
    Specifies the specific partition or partitions to add to the nodegroup.

**ALTER NODEGROUP**

> NODES is a synonym for NODE. Any specified partition must not already be defined in the nodegroup (SQLSTATE 42728).

**DROP NODE**
> Specifies the specific partition or partitions to drop from the nodegroup. NODES is a synonym for NODE. Any specified partition must already be defined in the nodegroup (SQLSTATE 42729).

*nodes-clause*
> Specifies the partition or partitions to be added or dropped.

> *node-number1*
>> Specify a specific partition number.

> **TO** *node-number2*
>> Specify a range of partition numbers. The value of *node-number2* must be greater than or equal to the value of *node-number1* (SQLSTATE 428A9).

**LIKE NODE** *node-number*
> Specifies that the containers for the existing table spaces in the nodegroup will be the same as the containers on the specified *node-number*. The partition specified must be a partition that existed in the nodegroup prior to this statement and is not included in a DROP NODE clause of the same statement.

**WITHOUT TABLESPACES**
> Specifies that the default table spaces are not created on the newly added partition or partitions. The ALTER TABLESPACE using the FOR NODE clause must be used to define containers for use with the table spaces that are defined on this nodegroup. If this option is not specified, the default containers are specified on newly added partitions for each table space defined on the nodegroup.

### Rules

- Each partition or node specified by number must be defined in the `db2nodes.cfg` file (SQLSTATE 42729). See "Data Partitioning Across Multiple Partitions" on page 57 for information about this file.
- Each *node-number* listed in the ON NODES clause must be for a unique partition (SQLSTATE 42728).
- A valid partition number is between 0 and 999 inclusive (SQLSTATE 42729).
- A partition cannot appear in both the ADD and DROP clauses (SQLSTATE 42728).
- There must be at least one partition remaining in the nodegroup. The last partition cannot be dropped from a nodegroup (SQLSTATE 428C0).
- If neither the LIKE NODE clause nor the WITHOUT TABLESPACES clause is specified when adding a partition, the default is to use the lowest partition number of the existing partitions in the nodegroup (say it is 2)

and proceed as if LIKE NODE 2 had been specified. For an existing partition to be used as the default it must have containers defined for all the table spaces in the nodegroup (column IN_USE of SYSCAT.NODEGROUPDEF is not 'T').

**Notes**

- When a partition or node is added to a nodegroup, a catalog entry is made for the partition (see SYSCAT.NODEGROUPDEF). The partitioning map is changed immediately to include the new partition along with an indicator (IN_USE) that the partition is in the partitioning map if either:
  - no table spaces are defined in the nodegroup or
  - no tables are defined in the table spaces defined in the nodegroup and the WITHOUT TABLESPACES clause was not specified.

  The partitioning map is not changed and the indicator (IN_USE) is set to indicate that the partition is not included in the partitioning map if either:
  - tables exist in table spaces in the nodegroup or
  - table spaces exist in the nodegroup and the WITHOUT TABLESPACES clause was specified.

  To change the partitioning map, the REDISTRIBUTE NODEGROUP command must be used. This redistributes any data, changes the partitioning 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 partition is dropped from a nodegroup, the catalog entry for the partition (see SYSCAT.NODEGROUPDEF) is updated. If there are no tables defined in the table spaces defined in the nodegroup, the partitioning map is changed immediately to exclude the dropped partition and the entry for the partition in the nodegroup is dropped. If tables exist, the partitioning map is not changed and the indicator (IN_USE) is set to indicate that the partition is waiting to be dropped. The REDISTRIBUTE NODEGROUP command must be used to redistribute the data and drop the entry for the partition from the nodegroup.

**Example**

Assume that you have a six-partition database that has the following partitions: 0, 1, 2, 5, 7, and 8. Two partitions are added to the system with partition numbers 3 and 6.

- Assume that you want to add both partitions or nodes 3 and 6 to a nodegroup called MAXGROUP and have the table space containers like those on partition 2. The statement is as follows:

```
ALTER  NODEGROUP MAXGROUP
ADD NODES (3,6) LIKE NODE 2
```

## ALTER NODEGROUP

- Assume that you want to drop partition 1 and add partition 6 to nodegroup
  MEDGROUP. You will define the table space containers separately for
  partition 6 using ALTER TABLESPACE. The statement is as follows:

```
ALTER NODEGROUP MEDGROUP
 ADD NODE(6) WITHOUT TABLESPACES
 DROP NODE(1)
```

## ALTER SERVER

The ALTER SERVER statement[54] is used to:

- 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.

### 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 authorization ID of the statement must include either SYSADM or DBADM authority on the federated database.

### Syntax

```
►►──ALTER SERVER──────────────────────────────────────────────►

  ►─┬─ server-name ─┬──────────────────────────────────────────┬─►
    │               └─VERSION─┤ server-version ├────────────┐   │
    └─TYPE─ server-type ─┬────────────────────────────────┬─┘
                         └─VERSION─┤ server-version ├──────────┐
                                                └─WRAPPER─ wrapper-name ─┘

  ►─OPTIONS──(─┬─▼─┬─ADD─┬──── server-option-name ─ string-constant ─┬─┬─)─►◄
              │   └─SET─┘                                            │
              └─DROP─ server-option-name ──────────────────────────┘
```
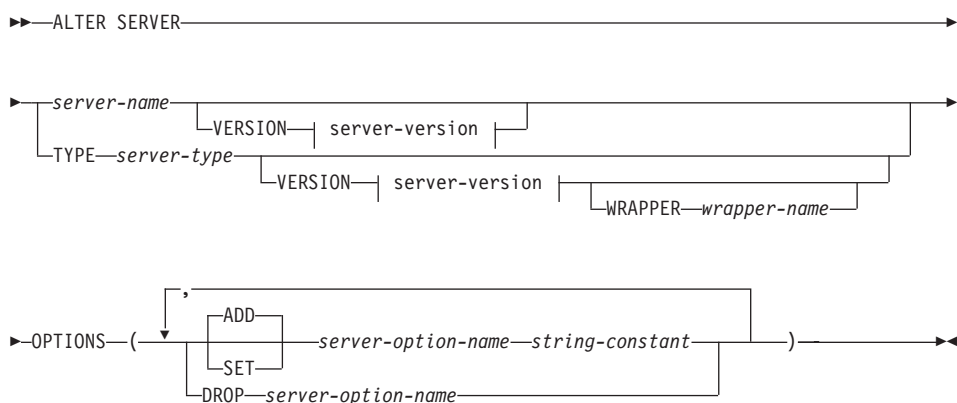
**server-version:**

---

54. 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. For information about federated systems, see "DB2 Federated Systems" on page 39. For information about DRDA application servers, see "Distributed Relational Database" on page 28.

Chapter 6. Statements   **445**

## ALTER SERVER

```
├──┬─version─┬──────────────────────────────────────┬─────────────────────────┤
│  │         └──.──release──┬──────────────┬─────────┘                         │
│  │                        └──.──mod──────┘                                   │
│  └─version-string-constant───────────────────────────────────────────────────┘
```
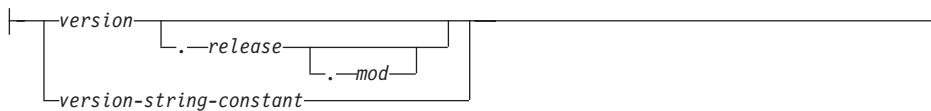
### Description

*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. *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').

**TYPE** *server-type*
  Specifies the type of data source to which the changes being requested are
  to apply. The server type must be one that is listed in the catalog.

**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. Refer to "Server Options" on page 1049 for descriptions of *server-option-name*s and their settings.

**ADD**
   Enables a server option.

**SET**
   Changes the setting of a server option.

*server-option-name*
   Names a server option that is to be enabled or reset.

*string-constant*
   Specifies the setting for *server-option-name* as a character string constant.

**DROP** *server-option-name*
   Drops a server option.

## Notes

- This statement does not support the DBNAME and NODE server options (SQLSTATE 428EE).
- 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 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 55007).
  - 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 55007).
- If the server option is set to one value for a type of data source, and set to another value for an instance of the 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.

## 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

**ALTER SERVER**

these data sources have started to run on an upgraded CPU that's half as fast
as your local 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 a DB2 Universal Database for AS/400 Version 3.0 data
source called SUNDIAL has been upgraded to Version 3.1.

```
ALTER SERVER SUNDIAL
   VERSION 3.1
```

## ALTER TABLE

The ALTER TABLE statement modifies existing tables by:
- Adding one or more columns to a table
- Adding or dropping a primary key
- Adding or dropping one or more unique or referential constraints
- Adding or dropping one or more check constraint definitions
- Altering the length of a VARCHAR column
- Altering a reference type column to add a scope
- Adding or dropping a partitioning key
- Changing table attributes such as the data capture option, pctfree, lock size, or append mode.
- Setting the table to not logged initially state.

### 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 at least one of the following:
- ALTER privilege on the table to be altered
- CONTROL privilege on the table to be altered
- ALTERIN privilege on the schema of the table
- SYSADM or DBADM authority.

To create or drop a foreign key, the privileges held by the authorization ID of the statement must include one of the following on the parent table:
- REFERENCES privilege on the table
- REFERENCES privilege on each column of the specified parent key
- CONTROL privilege on the table
- SYSADM or DBADM authority.

To drop a primary key or unique constraint of table T, the privileges held by the authorization ID of the statement must include at least one of the following on every table that is a dependent of this parent key of T:
- ALTER privilege on the table
- CONTROL privilege on the table
- ALTERIN privilege on the schema of the table
- SYSADM or DBADM authority.

## ALTER TABLE

### Syntax

```
►►──ALTER TABLE──table-name─────────────────────────────────────────►

         ┌──────────────────────────────────────────────────┐
         │          (1)    ┌─COLUMN─┐                         │
     ►──┬─ADD─────────────┴────────┴──column-definition──────┬──►◄
        │                         ├─unique-constraint─────────┤
        │                         ├─referential-constraint────┤
        │                         ├─check-constraint──────────┤
        │                         └─partitioning-key-definition┘
        │          ┌─COLUMN─┐
        ├─ALTER────┴────────┴──column-alteration───────────────┤
        ├─DROP──┬─PRIMARY KEY────────────────────────┬─────────┤
        │       ├─FOREIGN KEY──┬─constraint-name──────┤
        │       ├─UNIQUE───────┤
        │       ├─CHECK────────┤
        │       ├─CONSTRAINT───┘
        │       └─PARTITIONING KEY──────────────────────┘
        ├─DATA CAPTURE──┬─NONE──────────────────────────────────┤
        │               └─CHANGES──┬──────────────────────────┤
        │                          └─INCLUDE LONGVAR COLUMNS──┘
        ├─ACTIVATE NOT LOGGED INITIALLY──┬─────────────────────┤
        │                                └─WITH EMPTY TABLE───┘
        ├─PCTFREE──integer───────────────────────────────────────┤
        ├─LOCKSIZE──┬─ROW───┬────────────────────────────────────┤
        │           └─TABLE─┘
        ├─APPEND──┬─ON──┬─────────────────────────────────────────┤
        │         └─OFF─┘
        │                 ┌─CARDINALITY─┐
        └─┬─VOLATILE─────┬─┴─────────────┴──────────────────────┘
          └─NOT VOLATILE─┘
```

**column-alteration:**

```
├──column-name──┬─SET DATA TYPE──┬─VARCHAR───────────┬──(──integer──)──┬──┤
                │                 ├─CHARACTER VARYING─┤
                │                 └─CHAR VARYING──────┘
                └─ADD SCOPE──┬─typed-table-name─┬───────────────────┘
                             └─typed-view-name──┘
```

**Notes:**

1     For compatibility with Version 1, the ADD keyword is optional for:
   - -unnamed PRIMARY KEY constraints
   - -unnamed referential constraints
   - -referential constraints whose name follows the phrase FOREIGN KEY.

**column-definition:**

```
├──column-name──┤ data-type ├──────────────────────────────────────────────────────┤
                              └──┤ column-options ├──┘
```

**column-options:**

```
                ┌──────────────────────────────────────────────────────────────┐
├───────────────┴──┬──NOT NULL────────────────────────────────────────┬─────────┤
                   ├──┤ default-clause ├─────────────────────────────┤ │
                   │                         (1)                       │
                   ├──┤ lob-options ├────────────────────────────────┤ │
                   │                      (2)                          │
                   ├──┤ datalink-options ├───────────────────────────┤ │
                   │                            (3)                    │
                   ├──SCOPE──┬──typed-table-name2──┬──────────────────┤ │
                   │         └──typed-view-name2───┘                  │ │
                   │                                  ┌──PRIMARY KEY──┐│
                   │              (4)                 ├──UNIQUE───────┤│
                   └──CONSTRAINT──────constraint-name─┤ references-clause ├─┤
                                                      └──CHECK──(──check-condition──)──┘
```
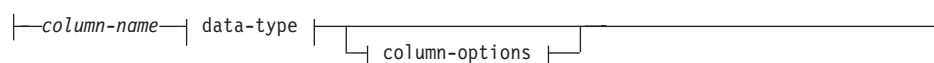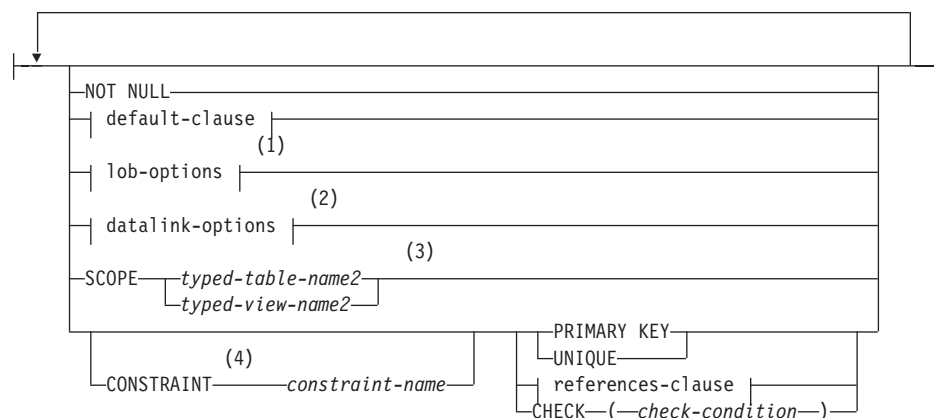
**Notes:**

**1** The lob-options clause only applies to large object types (BLOB, CLOB and DBCLOB) and distinct types based on large object types.

**2** The datalink-options clause only applies to the DATALINK type and distinct types based on the DATALINK type.

**3** The SCOPE clause only applies to the REF type.

**4** For compatibility with Version 1, the CONSTRAINT keyword may be omitted in a *-column-definition* defining a references-clause.

**default-clause:**
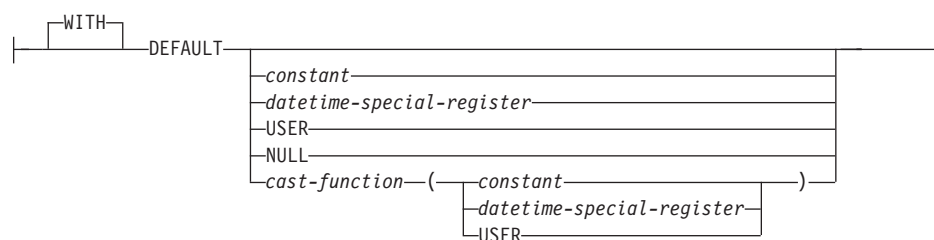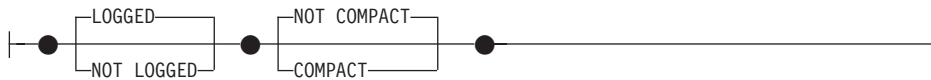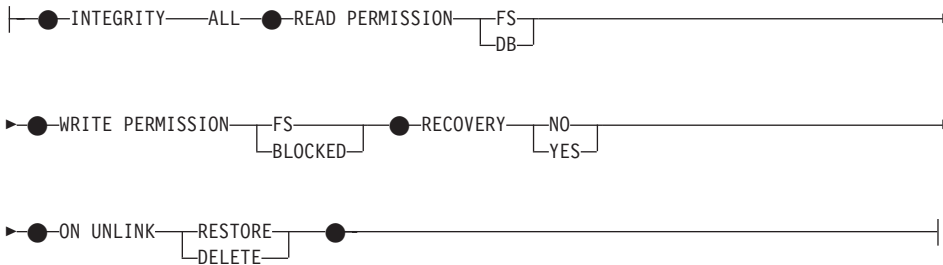
```
      ┌──WITH──┐
├─────┴────────┴──DEFAULT──┬─────────────────────────────────────────────────┬──┤
                           ├──constant──────────────────────────────────────┤
                           ├──datetime-special-register─────────────────────┤
                           ├──USER──────────────────────────────────────────┤
                           ├──NULL──────────────────────────────────────────┤
                           └──cast-function──(──┬──constant─────────────────┬──)──┘
                                                ├──datetime-special-register─┤
                                                └──USER─────────────────────┘
```

**lob-options:**

```
├──┬─────────────┬──●──┬──────────────┬──●──────────────────────────────────────────┤
   ├──LOGGED─────┤     ├──NOT COMPACT─┤  ●
   └──NOT LOGGED─┘     └──COMPACT─────┘
```

**datalink-options:**

```
├──LINKTYPE URL──┬──NO LINK CONTROL──────────────────────────────────────┬──────┤
                 └──FILE LINK CONTROL──┬──file-link-options──┬──────────┘
                                       └──MODE DB2OPTIONS────┘
```

**file-link-options:**

```
├──●──INTEGRITY──ALL──●──READ PERMISSION──┬──FS──┬─────────────────────────────►
                                          └──DB──┘

►──●──WRITE PERMISSION──┬──FS──────┬──●──RECOVERY──┬──NO───┬──────────────────►
                        └──BLOCKED─┘               └──YES──┘

►──●──ON UNLINK──┬──RESTORE─┬──●──────────────────────────────────────────────┤
                 └──DELETE──┘
```

**references-clause:**

```
├──REFERENCES──table-name──┬──────────────────────┬──┤ rule-clause ├──────────┤
                           │        ┌──,──────┐    │
                           └──(──▼──column-name──)─┘
```

**rule-clause:**

```
├──●──┬──ON DELETE NO ACTION──────────┬──●──┬──ON UPDATE NO ACTION──┬──●──────┤
      └──ON DELETE──┬──RESTRICT──┬─────┘     └──ON UPDATE RESTRICT───┘
                    ├──CASCADE───┤
                    └──SET NULL──┘
```
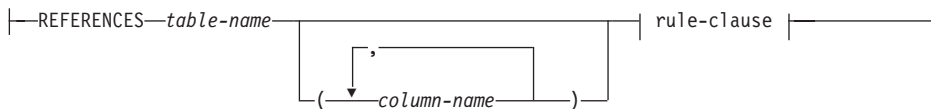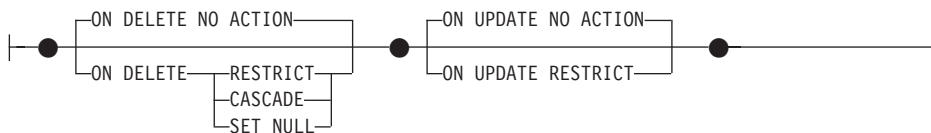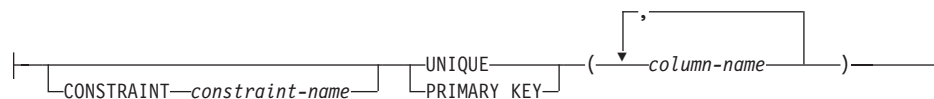
**unique-constraint:**

**referential-constraint:**
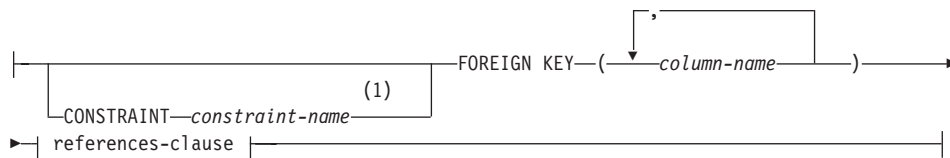


**check-constraint:**
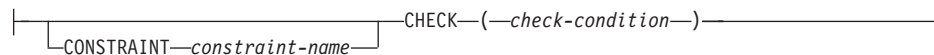


**partitioning-key-definition:**



**Notes:**

**1**  For compatibility with Version1, *-constraint-name* may be specified
     following FOREIGN KEY (without the CONSTRAINT keyword).

## Description

*table-name*
> Identifies the table to be changed. It must be a table described in the
> catalog and must not be a view or a catalog table. If *table-name* identifies a
> summary table, alterations are limited to activating not logged initially,
> changing pctfree, locksize, append, or volatile. The *table-name* cannot be a
> nickname (SQLSTATE 42809).

**ADD** *column-definition*
> Adds a column to the table. The table must not be a typed table
> (SQLSTATE 428DH). If the table has existing rows, every value of the
> newly added column is its default value. The new column is the 'last'
> column of the table. That is, if initially there are *n* columns, the added
> column is column *n*+1. The value of *n* cannot be greater than 499.

Adding the new column must not make the total byte count of all columns exceed the maximum record size as specified in Table 31 on page 924. See "Notes" on page 672 for more information.

*column-name*
> Is the name of the column to be added to the table. The name cannot be qualified. Existing column names in the table cannot be used (SQLSTATE 42711).

*data-type*
> Is one of the data types listed under "CREATE TABLE" on page 639.

**NOT NULL**
> Prevents the column from containing null values. The *default-clause* must also be specified (SQLSTATE 42601).

*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 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 19. If a column is defined as a DATALINK, then a specific default value cannot be specified.
> >
> > 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.

*Table 19. Default Values (when no value specified)*

| Data Type | Default Value |
|---|---|
| Numeric | 0 |
| Fixed-length character string | Blanks |
| Varying-length character string | A string of length 0 |
| Fixed-length graphic string | Double-byte blanks |
| Varying-length graphic 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. |

*Table 19. Default Values (when no value specified)  (continued)*

| Data Type | Default Value |
|---|---|
| Timestamp | For existing rows, a date corresponding to January 1, 0001, and a time corresponding to 0 hours, 0 minutes, 0 seconds and 0 microseconds. For added rows, the current timestamp. |
| Binary string (blob) | A string of length 0 |

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 could 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
> - not have non-zero 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 characters 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 or UPDATE 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 timestamp when the ALTER TABLE statement is processed.

**USER**
> Specifies the value of the USER special register at the time of INSERT or UPDATE as the default for the column. If USER is

specified, the data type of the column must be a character string with a length not less than the length attribute of USER. For existing rows, the value is the authorization ID of the ALTER TABLE statement.

**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 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 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. An example using the *cast-function* is given in Example 8 on page 469.

*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**
Specifies the USER special register. 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.
>
> If the value specified is not valid, an error (SQLSTATE 42894) is raised.

*lob-options*
> Specifies options for LOB data types. See *lob-options* in "CREATE TABLE" on page 639.

*datalink-options*
> Specifies options for DATALINK data types. See *datalink-options* in "CREATE TABLE" on page 639.

**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 actually 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 actually references 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-character identifier unique within the identifiers of the existing constraints defined on the table, is generated[55] by the system.
>
> When used with a PRIMARY KEY or UNIQUE constraint, the *constraint-name* may be used as the name of an index that is created to support the constraint. See "Notes" on page 465 for details on index names associated with unique constraints.

---

55. The identifier is formed of ″SQL″ followed by a sequence of 15 numeric characters generated by a timestamp-based function.

**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 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 description of the *unique-constraint* below.

**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 were specified as a separate clause.

See UNIQUE within the description of the *unique-constraint* below.

*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* on page 664 under CREATE TABLE.

**CHECK (***check-condition***)**

This provides a shorthand method of defining a check constraint that applies to a single column. See CHECK (*check-condition*) on page 666 under CREATE 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 information on *constraint-name*, see page 457.

**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 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 16 and the sum of their length attributes must not exceed 1 024. The length of any individual column

must not exceed 255 bytes. No LOB, LONG VARCHAR, LONG VARGRAPHIC or DATALINK column may be used as part of a unique key (even if the length attribute of the column is small enough to fit within the 255 byte limit) (SQLSTATE 42962).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). Any existing values in the set of identified columns must be unique (SQLSTATE 23515).

A check is performed to determine if 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) specifications. 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 (the selection is arbitrary). If no matching index is found, a unique index will automatically be created for the columns, as described in CREATE TABLE. See "Notes" on page 465 for details on index names associated with unique constraints.

**PRIMARY KEY** *(column-name,...)*
Defines a primary key 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 16 and the sum of their length attributes must not exceed 1 024. The length of any individual column must not exceed 255 bytes. The table must not have a primary key and the identified columns must be defined as NOT NULL. No LOB, LONG VARCHAR, LONG VARGRAPHIC or DATALINK column may be used as part of a primary key (even if the length attribute of the column is small enough to fit within the 255 byte limit) (SQLSTATE 42962). The set of columns in the primary key cannot be the same as the set of columns of a unique key (SQLSTATE 01543). [56] Any existing values in the set of identified columns must be unique (SQLSTATE 23515).

A check is performed to determine if 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) specifications. If a matching index definition is found, the description of the index is changed to indicate that it is the

---

56. If LANGLEVEL is SQL92E or MIA then an error is returned, SQLSTATE 42891.

primary index, as 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 (the selection is arbitrary). If no matching index is found, a unique index will automatically be created for the columns, as described in CREATE TABLE. See "Notes" on page 465 for details on index names associated with unique constraints.

Only one primary key can be defined on a table.

If the table has a partitioning key, the columns of a *unique-constraint* must be a superset of the partitioning key columns; column order is unimportant.

**ADD** *referential-constraint*
Defines a referential constraint. See *referential-constraint* on page 663 under CREATE TABLE.

**ADD** *check-constraint*
Defines a check constraint. See *check-constraint* in "CREATE TABLE" on page 639.

**ADD** *partitioning-key-definition*
Defines a partitioning key. The table must be defined in a table space on a single-partition nodegroup and must not already have a partitioning key. A partitioning key cannot be added to a table that is a subtable (SQLSTATE 428DH).

**PARTITIONING KEY (***column-name***...)**
Defines a partitioning key 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. No LONG VARCHAR, LONG VARGRAPHIC, or LOB column may be used as part of a partitioning key (SQLSTATE 42962).

For restrictions related to the partitioning key, see "Rules" on page 465.

**USING HASHING**
Specifies the use of the hashing function as the partitioning method for data distribution. This is the only partitioning method supported.

**ALTER** *column-alteration*
Alters the characteristics of a column.

*column-name*
Is the name of the column to be altered in the table. The *column-name* must identify an existing column of the table (SQLSTATE 42703). The name cannot be qualified.

**SET DATA TYPE VARCHAR (***integer***)**

Increase the length of an existing VARCHAR column. CHARACTER VARYING or CHAR VARYING can be used as synonyms for the VARCHAR keyword. The data type of *column-name* must be VARCHAR and the current maximum length defined for the column must not be greater than the value for *integer* (SQLSTATE 42837). The value for *integer* may range up to 32 672. The table must not be a typed table (SQLSTATE 428DH).

Altering the column must not make the total byte count of all columns exceed the maximum record size as specified in Table 31 on page 924 (SQLSTATE 54010). See "Notes" on page 672 for more information. If the column is used in a unique constraint or an index, the new length must not be greater than 255 bytes and must not cause the sum of the lengths of the columns for the unique constraint or index to exceed 1 024 (SQLSTATE 54008).

**ADD SCOPE**

Add a scope to an existing reference type column that does not already have a scope defined (SQLSTATE 428DK). If the table being altered is a typed table, the column must not be inherited from a supertable (SQLSTATE 428DJ). Refer to "ALTER TYPE (Structured)" on page 475 for examples.

*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 actually 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 actually reference existing rows in *typed-view-name*.

**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.

**DROP FOREIGN KEY** *constraint-name*

Drops the referential constraint *constraint-name*. The *constraint-name* must identify a referential constraint. For information on implications of dropping a referential constraint see "Notes" on page 465.

**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. For information on implications of dropping a unique constraint see "Notes" on page 465.

**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. For information on implications of dropping a constraint see "Notes" on page 465.

**DROP CHECK** *constraint-name*
Drops the check constraint *constraint-name*. The *constraint-name* must identify an existing check constraint defined on the table.

**DROP PARTITIONING KEY**
Drops the partitioning key. The table must have a partitioning key and must be in a table space defined on a single-partition nodegroup.

**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 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 defined to allow data on a partition other than the catalog partition (multiple partition nodegroup or nodegroup with partition other than the catalog partition), then this option is not supported (SQLSTATE 42997).

Further information about using replication can be found in the *Administration Guide* and the *Replication Guide and Reference*.

**INCLUDE LONGVAR COLUMNS**
Allows data replication utilities to capture changes made to LONG VARCHAR or LONG VARGRAPHIC columns. The clause may 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. The table must have been originally created with the NOT LOGGED INITIALLY attribute (SQLSTATE 429AA).

Any changes 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 made to the system catalog by the ALTER statement in which the NOT LOGGED INITIALLY attribute is activated are logged. Any subsequent changes 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 using this feature to avoid locks on the catalog tables while inserting data, it is important that only this clause be specified on the ALTER TABLE statement. Use of any other clause in the ALTER TABLE statement will result in catalog locks. If no other clauses are specified for the ALTER TABLE statement, then only a SHARE lock will be acquired on the system catalog tables. This can greatly reduce the possibility of concurrency conflicts for the duration of time between when this statement is executed and when the unit of work in which it was executed 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).

For more information on the NOT LOGGED INITIALLY attribute, see the description of this attribute in "CREATE TABLE" on page 639.

**Note:** An error in any operation in the unit of work in which the NOT LOGGED INITIALLY attribute is active will result in the entire unit of work being rolled back (SQLSTATE 40506). 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.

**WITH EMPTY TABLE**
Causes all data currently in table to be removed. Once the data has been 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 will NOT be returned to its original state.

When this action is requested, no DELETE triggers defined on the affected table are fired. Any indexes that exist on the table are also emptied.

**PCTFREE** *integer*
Indicates what percentage of each page to leave as free space during load or reorganization. The value of *integer* can range from 0 to 99. The first row on each page is added without restriction. When additional rows are added, at least *integer* percent of free space is left on each page. The PCTFREE value is considered only by the LOAD and REORGANIZE TABLE utilities.

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 will 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).

**ROW**
Indicates the use of row locks. This is the default lock size when a table is created.

**TABLE**
Indicates the use of table locks. This means that the appropriate share or exclusive lock is acquired on the table and intent locks (except intent none) are not used. Use of this value may improve the performance of queries by limiting the number of locks that need to be acquired. However, concurrency is also reduced since all locks are held over the complete table.

Further information about locking can be found in the *Administration Guide*.

**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 will be appended and information about free space on pages will not be kept. The table must not have a clustered index (SQLSTATE 428CA).

**OFF**

Indicates that table data will be placed where there is available space. This is the default when a table is created.

The table should be reorganized after setting APPEND OFF since the information about available free space is not accurate and may result in poor performance during insert.

**VOLATILE**

This indicates to the optimizer that the cardinality of table *table-name* can vary significantly at run time, from empty to quite large. To access *table-name* the optimizer will use an index scan rather than a table scan, regardless of the statistics, if that index is index-only (all columns referenced are in the index) or that index is able to apply a predicate in the index scan. 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**

This indicates to the optimizer that the cardinality of *table-name* is not volatile. Access Plans to this table will continue to be based on the existing statistics and on the optimization level in place.

**CARDINALITY**

An optional key word to indicate that it is the number of rows in the table that is volatile and not the table itself.

## Rules

- A partitioning key column of a table cannot be updated (SQLSTATE 42997).
- Any unique or primary key constraint defined on the table must be a superset of the partitioning key, if there is one (SQLSTATE 42997).
- A nullable column of a partitioning key cannot be included as a foreign key column when the relationship is defined with ON DELETE SET NULL (SQLSTATE 42997).
- A column can only be referenced in one ADD or ALTER COLUMN clause in a single ALTER TABLE statement (SQLSTATE 42711).
- A column length cannot be altered if the table has any summary tables that are dependent on the table (SQLSTATE 42997).

## Notes

- ADD column clauses are processed prior to all other clauses. Other clauses are processed in the order that they are specified.
- Any columns added via ALTER TABLE will not automatically be added to any existing view of the table.
- When an index is automatically created for a unique or primary key constraint, the database manager will try 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 formed of "SQL" followed by a sequence of 15 numeric characters generated by a timestamp based function.

- Any table that may 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 executed 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 executed by the package on behalf of one of its statements.

- Any changes to primary key, unique keys, or foreign keys may have the following effect on packages, indexes, and other foreign keys.
  - If a primary key or unique key is added:
    - There is no effect on packages, foreign keys, or existing unique keys.[57]
  - If a 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 used for the constraint. There is no effect on packages.
    - All dependent foreign keys are dropped. Further action is taken for each dependent foreign key, as specified in the next item.
  - If a foreign key is added or dropped:
    - 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.

- Adding a column to a table will result in invalidation of all packages with insert usage on the altered table.

---

57. If the primary or unique key uses an existing unique index that was created in a previous version and has not been converted to support deferred uniqueness, then the index is converted and packages with update usage on the associated table are invalidated.

- Adding a check or referential constraint to a table that already exists and that is not in check pending state (see "SET INTEGRITY" on page 881) will cause the existing rows in the table to be immediately evaluated against the constraint. If the verification fails, an error (SQLSTATE 23512) is raised. If a table is in check pending state, adding a check or referential constraint will not immediately lead to the enforcement of the constraint. Instead, the corresponding constraint type flags used in the check pending operation will be updated. To begin enforcing the constraint, the SET INTEGRITY statement will need to be issued.

- Adding or dropping a check constraint will result in invalidation of all packages with either an insert usage on the object table or an update usage on at least one of the columns involved in the constraint.

- Adding a partitioning key will result in invalidation of all packages with an update usage on at least one of the columns of the partitioning key.

- A partitioning key that was defined by default as the first column of the primary key is not affected by dropping the primary key and adding a different primary key.

- Altering a column to increase the length will invalidate all packages that reference the table (directly or indirectly through a referential constraint or trigger) with the altered column.

- Altering a column to increase the length will regenerate views (except typed views) that are dependent on the table. If an error occurs while regenerating a view, an error is returned (SQLSTATE 56098). Any typed views that are dependent on the table are marked inoperative.

- Altering a column to increase the length may cause errors in processing triggers when a statement that would involve the trigger is prepared or bound. This may occur when row length based on the sum of the lengths of the transition variables and transition table columns is too long. This error is reported using SQLCODE -670, SQLSTATE 54010. If such a trigger were dropped a subsequent attempt to create it would result in error SQLCODE -1424, SQLSTATE 54040.

- VARCHAR and VARGRAPHIC columns that have been altered to be greater than 4 000 and 2 000 respectively should not be used as input parameters in functions in the SYSFUN schema (SQLSTATE 22001).

- Changing the LOCKSIZE for a table will result in invalidation of all packages that have a dependency on the altered table. Further information about locking can be found in the *Administration Guide*.

- The ACTIVATE NOT LOGGED INITIALLY clause can not be used when DATALINK columns with the FILE LINK CONTROL attribute are being added to the table (SQLSTATE 42613).

- Changing VOLATILE or NOT VOLATILE CARDINALITY will result in invalidation of all packages that have a dependency on the altered table.

- Replication customers should take caution when increasing the length of VARCHAR columns. The change data table associated with an application table might already be at or near the DB2 rowsize limit. The change data table should be altered before the application table, or the two should be altered within the same unit of work, to ensure that the alteration can be completed for both tables. Consideration should be given for copies, which may also be at or near the rowsize limit, or reside on platforms which lack the feature 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 increased VARCHAR column length, the Capture program will likely fail. If a copy containing the VARCHAR column is not altered before the subscription maintaining the copy runs, the subscription will likely fail.

## Examples

*Example 1:* Add a new column named RATING, which is one character long, to the DEPARTMENT table.

```
ALTER TABLE DEPARTMENT
  ADD RATING CHAR(1)
```

*Example 2:* Add a new column named SITE_NOTES to the PROJECT table. Create SITE_NOTES as a varying-length column with a maximum length of 1000 characters. The values of the column do not have an associated character set and therefore should not be translated.

```
ALTER TABLE PROJECT
  ADD SITE_NOTES  VARCHAR(1000) FOR BIT DATA
```

*Example 3:* Assume a table 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 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 additional column is needed to allow the recording of the quantity 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
```

*Example 4:* Alter table EMPLOYEE. Add the check constraint 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)
```

*Example 5:* Alter table EMPLOYEE. Drop the constraint REVENUE which was previously defined.

```
ALTER TABLE EMPLOYEE
DROP CONSTRAINT REVENUE
```

*Example 6:* Alter a table to log SQL changes in the default format.

```
ALTER TABLE SALARY1
   DATA CAPTURE NONE
```

*Example 7:* Alter a table to log SQL changes in an expanded format.

```
ALTER TABLE SALARY2
   DATA CAPTURE CHANGES
```

*Example 8:* Alter the EMPLOYEE table to add 4 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 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 sourced on a BLOB or datetime data type.

**ALTER TABLE**

*Example 9:* Assume that you have a table called CUSTOMERS that 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. You want to partition the table, so you need to create a partitioning key for the table. The table must be defined in a table space on a single-node nodegroup. The primary key must be a superset of the partitioning columns: at least one of the columns of the primary key must be used as the partitioning key. Assume that you want to make BRANCH_NO the partitioning key. You would do this with the following statement:

```
ALTER TABLE CUSTOMERS
      ADD PARTITIONING KEY (BRANCH_NO)
```

## ALTER TABLESPACE

The ALTER TABLESPACE statement is used to modify an existing table space in the following ways.

- Add a container to a DMS table space (that is, one created with the MANAGED BY DATABASE option).
- Add a container to a SMS table space on a partition (or node) 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.

### Invocation

This statement can be embedded in an application program or issued interactively. 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 authorization ID of the statement must have SYSCTRL or SYSADM authority.

### Syntax

```
►►──ALTER──TABLESPACE──tablespace-name──────────────────────────────►

  ┌──────────────────────────────────────────────────────────────┐
  │        ┌─ADD─┬─ database-container-clause ─┬─┬─────────────────┐ │
►─┴────────┤     └─ system-container-clause ───┘ └─ on-nodes-clause ┤ ├─►◄
           │                                        └─ on-nodes-clause ┘
           ├─PREFETCHSIZE─┬─number-of-pages──────────┬─
           │              └─integer─┬─K─┬
           │                        ├─M─┤
           │                        └─G─┘
           ├─BUFFERPOOL──bufferpool-name─────────────
           ├─OVERHEAD──number-of-milliseconds────────
           ├─TRANSFERRATE──number-of-milliseconds────
           ├─DROPPED TABLE RECOVERY──┬─ON──┬─
           │                         └─OFF─┘
           └─SWITCH ONLINE───────────────────────────
```
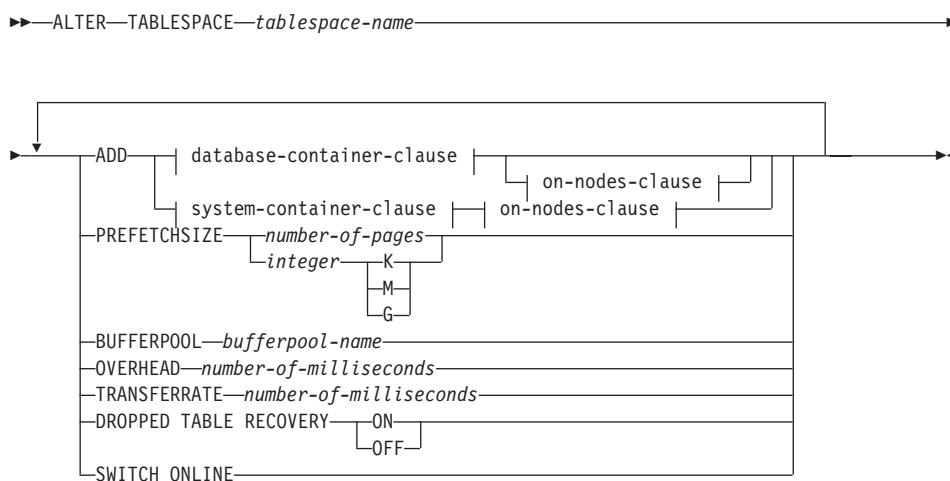
## ALTER TABLESPACE

**database-container-clause:**
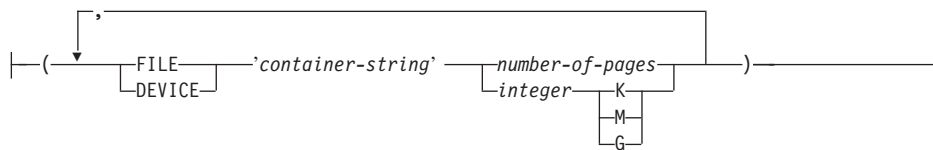


**system-container-clause:**



**on-nodes-clause:**
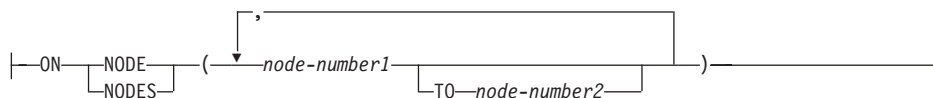


## Description

*tablespace-name*
> Names the table space. This is a one-part name. It is a long SQL identifier (either ordinary or delimited).

**ADD**
> ADD specifies that a new container is to be added to the table space.

*database-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. See the description of *container-clause* on page 684.

*system-container-clause*
> Adds one or more containers to an SMS table space on the specified partitions or nodes. 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 partitions for the table space. (SQLSTATE 42921). See the description of *system-containers* on page 683.

*on-nodes-clause*
> Specifies the partition or partitions for the added containers. See the description of *on-nodes-clause* on page 685.

**PREFETCHSIZE** *number-of-pages*
>    Specifies the number of PAGESIZE pages that will be read from the table space when data prefetching is being performed. The prefetch size value 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 value for prefetch size. Prefetching reads in data needed by a query prior to it being referenced by the query, so that the query need not wait for I/O to be performed.

**BUFFERPOOL** *bufferpool-name*
>    The name of the buffer pool used for tables in this table space. The buffer pool must currently exist in the database (SQLSTATE 42704). The nodegroup of the table space must be defined for the bufferpool (SQLSTATE 42735).

**OVERHEAD** *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. This value is used to determine the cost of I/O during query optimization.

**TRANSFERRATE** *number-of-milliseconds*
>    Any numeric literal (integer, decimal, or floating point) that specifies the time to read one page (4K or 8K) 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. This value is used to determine the cost of I/O during query optimization.

**DROPPED TABLE RECOVERY**
>    Dropped tables in the specified table space may be recovered using the RECOVER DROPPED TABLE ON option of the ROLLFORWARD command.

**SWITCH ONLINE**
>    Table spaces in OFFLINE state are brought online if the containers have become accessible. If the containers are not accessible an error is returned (SQLSTATE 57048).

## Notes

- Guidance on choosing optimal values for the PREFETCHSIZE, OVERHEAD, and TRANSFERRATE parameters, and information on rebalancing is provided in the *Administration Guide*.
- Once the new container has been added and the transaction is committed, the contents of the table space are automatically rebalanced across the containers. Access to the table space is not restricted during the rebalancing.

**ALTER TABLESPACE**

- If the table space is in OFFLINE state and the containers have become accessible, the user can disconnect all applications and connect to the database again to bring the table space out of OFFLINE state. Alternatively, 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 be 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 will result in an error (SQLSTATE 55041).

- In a partitioned database if more than one partition resides on the same physical node, then the same device or specific path cannot be specified for such partitions (SQLSTATE 42730). For this environment, either specify a unique *container-string* for each 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 in use, when the ALTER TABLESPACE statement was issued, will continue to be used in the interim.

## 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
```

## ALTER TYPE (Structured)

The ALTER TYPE statement is used to add or drop attributes 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. 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 as
least one of the following:
- SYSADM or DBADM authority
- ALTERIN privilege on the schema of the type.
- definer of the type as recorded in the DEFINER column of
  SYSCAT.DATATYPES

### Syntax

```
              ┌──────────,──────────────────────────────────────────┐
▶▶──ALTER TYPE──type-name──▼──┬─ADD ATTRIBUTE──── attribute-definition ─┬──────────▶◀
                             └─DROP ATTRIBUTE──attribute-name─────────┘
```
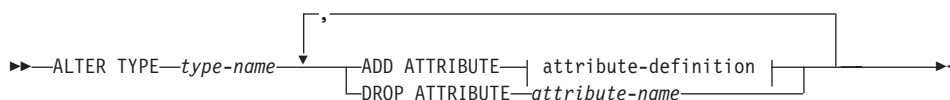
### Description

*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). The unqualified name must not be
> the same as the name of a built-in data type or BOOLEAN (SQLSTATE
> 42918). 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 characteristics of the new attribute. See *attribute-definition*
> in "CREATE TYPE (Structured)" on page 700.

**ALTER TYPE (Structured)**

> **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).

## Rules

- A type cannot be altered if it or one of its subtypes is the type of an existing table (SQLSTATE 55043).

## Notes

- When a type is altered, all packages are invalidated that depend on functions that use this type or a subtype of this type as a parameter or a result.

## 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)
```

## 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. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

### Authorization

If the authorization ID of the statement is different than the authorization name that is mapped to the data source, then the authorization ID of the statement must include SYSADM or DBADM authority. Otherwise, if the authorization ID and the authorization name match, then no privileges or authorities are required.

### Syntax

```
►►──ALTER USER MAPPING FOR──┬─authorization-name─┬──SERVER──server-name──────────►
                            └─USER───────────────┘

             ┌──────────────────,──────────────────────┐
             │         ┌─ADD─┐                          │
►──OPTIONS──(─▼─────────┼─────┼──user-option-name──string-constant─┴──)──────►◄
                       └─SET─┘
             └─DROP──user-option-name───────────────────┘
```
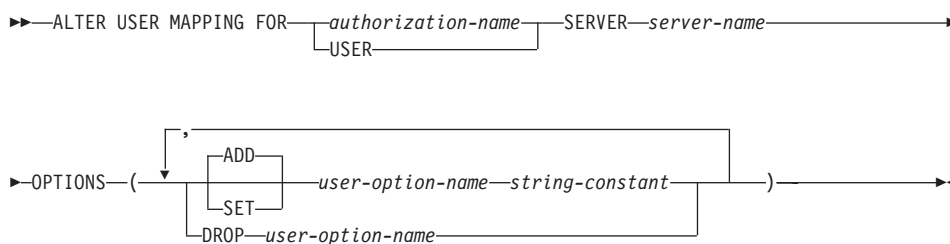
### Description

*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.

**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.

# ALTER USER MAPPING

OPTIONS
> Indicates what user options are to be enabled, reset, or dropped for the mapping that is being altered. Refer to "User Options" on page 1053 for descriptions of *user-option-name*s and their settings.

ADD
> Enables a user option.

SET
> Changes the setting of a user option.

*user-option-name*
> Names a user option that is to be enabled or reset.

*string-constant*
> Specifies the setting for *user-option-name* as a character string constant.

DROP *user-option-name*
> Drops a user 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.
- A user mapping cannot be altered in a given unit of work (UOW) if the UOW already includes a SELECT statement that references a nickname for a table or view at the data source that is to be included in the mapping.

## Examples

*Example 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' )
```

*Example 2:* Mary uses a federated database to connect to a DB2 Universal Database for OS/390 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

```
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.

### 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).
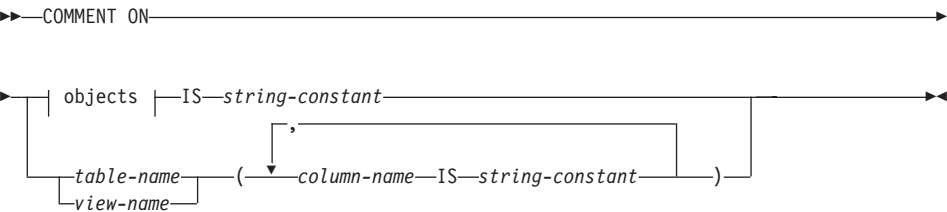
### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:
- SYSADM or DBADM authority
- ALTERIN privilege on the schema of the view
- Definer of the view to be altered
- CONTROL privilege on the view to be altered.

### Syntax

```
►►──ALTER VIEW──view-name──────────────────────────────────────────►

   ┌────────────────────────────────────────────────────────┐
   │        ┌─COLUMN─┐                      ┌─typed-table-name─┐ │
►──▼─ALTER──┼────────┼──column-name──ADD SCOPE──┤              ├─┴──►◄
                                      └─typed-view-name──┘
```
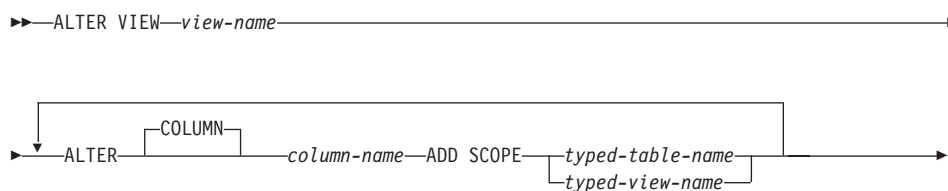
### Description

*view-name*
    Identifies the view to be changed. It must be a view described in the catalog.

**ALTER COLUMN** *column-name*
    Is the name of the column to be altered in the view. The *column-name* must identify an existing column of the view (SQLSTATE 42703). The name cannot be qualified.

**ADD SCOPE**
    Add 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*
        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*

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*.

## 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

```
►►──BEGIN DECLARE SECTION──────────────────────────────────────────►◄
```

### 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 (see "END DECLARE SECTION" on page 768).

### 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 must 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), and hv_blob_50k (blob(51200)) in a C
program.

```
EXEC SQL  BEGIN DECLARE SECTION;
  static short                        hv_smint;
  static struct {
          short hv_vchar24_len;
          char  hv_vchar24_value[24];
        }                             hv_vchar24;
  static double                       hv_double;
  static SQL TYPE IS BLOB(50K)        hv_blob_50k;
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 characters, 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

Invokes a procedure stored at the location of a database. A stored procedure, for example, executes at the location of the database, and returns data to the client application.

Programs using the SQL CALL statement are designed to run in two parts, one on the client and the other on the server. The server procedure at the database runs within the same transaction as the client application. If the client application and stored procedure are on the same partition, the stored procedure is executed locally.

### Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. However, the procedure name may be specified via a host variable and this, coupled with the use of the USING DESCRIPTOR clause, allows both the procedure name and the parameter list to be provided at run time; thus achieving the same effect as a dynamically prepared statement.

### Authorization

The authorization rules vary according to the server at which the procedure is stored.

**DB2 Universal Database:**
> The privileges held by the authorization ID of the CALL statement **at run time** statement must include at least one of the following:
> - EXECUTE privilege for the package associated with the stored procedure
> - CONTROL privilege for the package associated with the stored procedure
> - SYSADM or DBADM authority

**DB2 Universal Database for OS/390:**
> The privileges held by the authorization ID of the CALL statement **at bind time** must include at least one of the following:
> - EXECUTE privilege for the package associated with the stored procedure
> - Ownership of the package associated with the stored procedure
> - PACKADM authority for the package's collection
> - SYSADM authority

**DB2 for AS/400:**
> The privileges held by the authorization ID of the CALL statement **at bind time** must include at least one of the following:
> - If the stored procedure is written in REXX:

       – The system authorities *OBJOPR and *READ on the source file
         associated with the procedure
       – The system authority *EXECUTE on the library containing the
         source file and the system authority *USE to the CL command
    • If the stored procedure is not written in REXX:
       – The system authority *EXECUTE on both the program associated
         with the procedure and on the library containing that program
    • Administrative authority

## Syntax
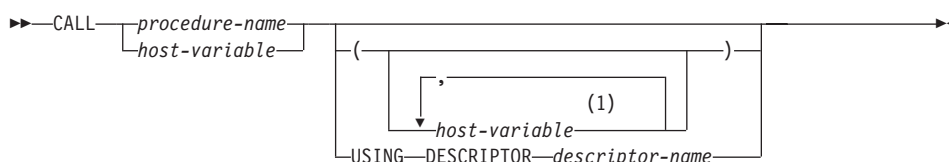


**Notes:**

**1** Stored procedures located at DB2 Universal Database for OS/390 and
    DB2 Universal Database for AS/400 servers and invoked by DB2
    Universal Database for OS/390 or DB2 Universal Database for AS/400
    clients support additional sources for procedure arguments (for example
    constant values). However, if the stored procedure is located on a DB2
    Universal Database or the procedure is invoked from a DB2 Universal
    Database client, all arguments must be provided via host variables.

## Description

*procedure-name* or *host-variable*
    Identifies the procedure to call. The procedure name may be specified
    either directly or within a host variable. The procedure identified must
    exist at the current server (SQLSTATE 42724).

    If *procedure-name* is specified it must be an ordinary identifier not greater
    than 254 bytes. Since this can only be an ordinary identifier, it cannot
    contain blanks or special characters and the value is converted to upper
    case. Thus, if it is necessary to use lower case names, blanks or special
    characters, the name must be specified via a *host-variable*.

    If *host-variable* is specified, it must be a character-string variable with a
    length attribute that is not greater than 254 bytes, and it must not include
    an indicator variable. Note that the value is **not** converted to upper case.
    *procedure-name* must be left-justified.

    The procedure name can take one of several forms. The forms supported
    vary according to the server at which the procedure is stored.

    **DB2 Universal Database:**

*procedure-name*  The name (with no extension) of the procedure to execute. The procedure invoked is determined as follows.

1. The *procedure-name* is used both as the name of the stored procedure library and the function name within that library. For example, if *procedure-name* is `proclib`, the DB2 server will load the stored procedure library named `proclib` and execute the function routine `proclib()` within that library.

   In UNIX-based systems, the DB2 server finds the stored procedure library in the default directory `sqllib/function`. Unfenced stored procedures are in the `sqllib/function/unfenced` directory.

   In OS/2, the location of the stored procedures is specified by the LIBPATH variable in the CONFIG.SYS file. Unfenced stored procedures are in the `sqllib\dll\unfenced` directory.

2. If the library or function could not be found, the *procedure-name* is used to search the defined procedures (in SYSCAT.PROCEDURES) for a matching procedure. A matching procedure is determined using the steps that follow.

   a. Find the procedures from the catalog (SYSCAT.PROCEDURES) where the PROCNAME matches the *procedure-name* specified and the PROCSCHEMA is a schema name in the SQL path (CURRENT PATH special register). If the schema name is explicitly specified, the SQL path is ignored and only procedures with the specified schema name are considered.

   b. Next, eliminate any of these procedures that do not have the same number of parameters as the number of arguments specified in the CALL statement.

   c. Chose the remaining procedure that is earliest in the SQL path.

   d. If there are no remaining procedures after step 2, an error is returned (SQLSTATE 42884).

>> Once the procedure is selected, DB2 will invoke the procedure defined by the external name.

*procedure-library!function-name*

>> The exclamation character (!), acts as a delimiter between the library name and the function name of the stored procedure. For example, if `proclib!func` was specified, then `proclib` would be loaded into memory and the function `func` from that library would be executed. This allows multiple functions to be placed in the same stored procedure library.

>> The stored procedure library is located in the directories or specified in the LIBPATH variable, as described in *procedure-name*.

*absolute-path!function-name*

>> The *absolute-path* specifies the complete path to the stored procedure library.

>> In a UNIX-based system, for example, if `/u/terry/proclib!func` was specified, then the stored procedure library `proclib` would be obtained from the directory `/u/terry` and the function `func` from that library would be executed.

>> In OS/2, if `d:\terry\proclib!func` was specified, then it would cause the database manager to load the func.dll file from the `d:\terry\proclib` directory.

In all these cases, the total length of the procedure name including its implicit or explicit full path must not be longer than 254 bytes.

**DB2 Universal Database for OS/390 (V4.1 or later) server:**
An implicit or explicit three part name. The parts are as follows.

**high order:** The location name of the server where the procedure is stored.

**middle:** SYSPROC

**middle:** Some value in the PROCEDURE column of the SYSIBM.SYSPROCEDURES catalog table.

**DB2 for OS/400 (V3.1 or later) server:**
The external program name is assumed to be the same as the *procedure-name*.

**CALL**

> For portability, *procedure-name* should be specified as a single token no larger than 8 bytes.

*(host-variable,...)*
> Each specification of *host-variable* is a parameter of the CALL. The nth parameter of the CALL corresponds to the nth parameter of the server's stored procedure.
>
> Each *host-variable* is assumed to be used for exchanging data in both directions between client and server. In order to avoid sending unnecessary data between client and server, the client application should provide an indicator variable with each parameter and set the indicator to -1 if the parameter is not used to transmit data to the stored procedure. The stored procedure should set the indicator variable to -128 for any parameter that is not used to return data to the client application.
>
> If the server is DB2 Universal Database the parameters must have matching data types in both the client and server program.[58]

**USING DESCRIPTOR** *descriptor-name*
> Identifies an SQLDA that must contain a valid description of host variables. The nth SQLVAR element corresponds to the nth parameter of the server's stored procedure.
>
> Before the CALL statement is processed, the application 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 following fields of each Base SQLVAR element passed must be initialized:
>   - SQLTYPE
>   - SQLLEN
>   - SQLDATA
>   - SQLIND
>
> The following fields of each Secondary SQLVAR element passed must be initialized:

---

58. DB2 Universal Database for OS/390 and DB2 Universal Database for AS/400 servers support conversions between compatible data types when invoking their stored procedures. For example, if the client program uses the INTEGER data type and the stored procedure expects FLOAT, the server will convert the INTEGER value to FLOAT before invoking the procedure.

- LEN.SQLLONGLEN
- SQLDATALEN
- SQLDATATYPE_NAME

Each SQLDA is assumed to be used for exchanging data in both directions between client and server. In order to avoid sending unnecessary data between client and server, the client application should set the SQLIND field to -1 if the parameter is not used to transmit data to the stored procedure. The stored procedure should set the SQLIND field -128 for any parameter that is not used to return data to the client application.

## Notes

- *Use of Large Object (LOB) data types*:

  If the client and server application needs to specify LOB data from an SQLDA, allocate double the number of SQLVAR entries.

  LOB data types are supported by stored procedures starting with DB2 Version 2. The LOB data types are not supported by all down level clients or servers.

- *Returning Result Sets from Stored Procedures*:

  If the client application program is written using CLI, result sets can be returned directly to the client application. The stored 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 that is invoked via CLI:

  - For every cursor that has been left open, a result set is returned to the application.
  - If more than one cursor is left open, the result sets are returned in the order in which their cursors were opened.
  - 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 stored procedure at the time the stored procedure is terminated, then rows 151 through 500 will be returned to the stored procedure.

  For additional information refer to the *Application Development Guide* and the *CLI Guide and Reference*.

- *Inter-operability between the CALL statement and the DARI API*:

  In general, the CALL statement will not work with existing DARI procedures. See the *Application Development Guide* for details.

## Examples

*Example 1:*

## CALL

In C, invoke a procedure called TEAMWINS in the ACHIEVE library passing it a parameter stored in the host variable HV_ARGUMENT.

```
strcpy(HV_PROCNAME, "ACHIEVE!TEAMWINS");
CALL :HV_PROCNAME (:HV_ARGUMENT);
```

*Example 2:*

In C, invoke a procedure called :SALARY_PROC using the SQLDA named INOUT_SQLDA.

```
struct sqlda *INOUT_SQLDA;

/* Setup code for SQLDA variables goes here */

CALL :SALARY_PROC
USING DESCRIPTOR :*INOUT_SQLDA;
```

*Example 3:*

A Java stored 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 stored 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, variable1 ) ;
stpCall.setBigDecimal( 2, variable2 ) ;
stpCall.setInt( 3, variable3 ) ;

stpCall.registerOutParameter( 2, Types.DECIMAL, 2 ) ;
stpCall.registerOutParameter( 3, Types.INTEGER ) ;

stpCall.execute() ;

variable2 = stpCall.getBigDecimal(2) ;
variable3 = stpCall.getInt(3) ;
...
```

This application code fragment will invoke the Java method *onhand* in class *parts* since the procedure-name specified on the CALL statement is found in

the database and has the external name 'parts!onhand'.

## 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.

### Authorization

None required. See "DECLARE CURSOR" on page 731 for the authorization required to use a cursor.

### Syntax

```
►►──CLOSE──cursor-name──────────────────────────────────►◄
                       └─WITH RELEASE─┘
```

### 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.

**WITH RELEASE**
> The release of all read locks that have been held for the cursor is attempted. Note that not all of the read 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.
- CLOSE does not cause a commit or rollback operation.
- 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. Refer to "Appendix H. Comparison of Isolation Levels" on page 1083 for more details.

If a cursor that was originally either RR or RS is reopened after being closed using the WITH RELEASE clause, then new read locks will be acquired.

- Special rules apply to cursors within a stored procedure that have not been closed before returning to the calling program. See "Notes" on page 489 for more information.

## 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 ON

The COMMENT ON 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. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

### Authorization

The privileges that must be held by the authorization ID of the COMMENT ON statement must include one of the following:
- SYSADM or DBADM
- definer of the object (underlying table for column or constraint) as recorded in the DEFINER column of the catalog view for the object (OWNER column for a schema)
- ALTERIN privilege on the schema (applicable only to objects allowing more than one-part names)
- CONTROL privilege on the object (applicable to index, package, table and view objects only)
- ALTER privilege on the object (applicable to table objects only)

Note that for table space or nodegroup the authorization ID must have SYSADM or SYSCTRL authority.

### Syntax

```
►►──COMMENT ON──────────────────────────────────────────────────►

►──┬─│ objects │──IS──string-constant────────────────────────────►◄
   │                                                           │
   └─┬─table-name─┬──(──┬◄─────,────────────────────┬──)──┘
     └─view-name──┘      └──column-name──IS──string-constant──┘
```

**objects:**

```
├──── ALIAS──alias-name──────────────────────────────────────────┤
    ├──COLUMN──┬──table-name.column-name──┬──────────────────────
    │          └──view-name.column-name───┘
    ├──CONSTRAINT──table-name.constraint-name──────────────────────
    ├──FUNCTION────function-name─────────────────────────────────
    │                          ┌──(──────────────────────────)──┐
    │                          │        ┌──,◄─────┐            │
    │                          └────────┴─data-type─┴───────────┘
    ├──SPECIFIC FUNCTION──specific-name───────────────────────────
    ├──FUNCTION MAPPING──function-mapping-name─────────────────────
    │                              (1)
    ├──INDEX──index-name──────────────────────────────────────────
    ├──NICKNAME──nickname─────────────────────────────────────────
    ├──NODEGROUP──nodegroup-name──────────────────────────────────
    ├──PACKAGE──package-name──────────────────────────────────────
    ├──PROCEDURE──procedure-name─────────────────────────────────
    │                          ┌──(──────────────────────────)──┐
    │                          │        ┌──,◄─────┐            │
    │                          └────────┴─data-type─┴───────────┘
    ├──SPECIFIC PROCEDURE──specific-name──────────────────────────
    ├──SCHEMA──schema-name────────────────────────────────────────
    ├──SERVER──server-name────────────────────────────────────────
    ├──SERVER OPTION──server-option-name──FOR──┤ remote-server ├──
    ├──TABLE──┬──table-name──┬────────────────────────────────────
    │         └──view-name───┘
    ├──TABLESPACE──tablespace-name────────────────────────────────
    ├──TRIGGER──trigger-name──────────────────────────────────────
    │  ┌──────────────────┐──TYPE──type-name──────────────────────
    │  │            (2)    │
    │  └──DISTINCT─────────┘
    ├──TYPE MAPPING──type-mapping-name────────────────────────────
    └──WRAPPER──wrapper-name──────────────────────────────────────
```

**remote-server:**

```
├──┬──SERVER──server-name─────────────────────────────────────────┤
   └──TYPE──server-type──┬───────────────────────────────────────┘
                         └──VERSION──┤ server-version ├──┬─────────────────────────┘
                                                         └──WRAPPER──wrapper-name──┘
```

**server-version:**

```
├──┬──version──┬──────────────────────────────────┬──────────────┤
   │           └──.──release──┬──────────┐         │
   │                          └──.──mod──┘         │
   └──version-string-constant────────────────────────────────────┘
```

# COMMENT ON

**Notes:**

**1** *-Index-name* can be the name of either an index or an index specification.

**2** The keyword DATA can be used as a synonym for DISTINCT.

## Description

**ALIAS** *alias-name*
> Indicates a comment will be added or replaced for an alias. The *alias-name* must identify an alias that is described in the catalog (SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.TABLES catalog view for the row that describes the alias.

**COLUMN** *table-name.column-name* or *view-name.column-name*
> Indicates a comment will be added or replaced for a column. The *table-name.column-name* or *view-name.column-name* combination must identify a column and table combination that is described in the catalog (SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.COLUMNS catalog view for the row that describes the column.
>
> A comment cannot be made on a column of an inoperative view. (SQLSTATE 51024).

**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 be described in the catalog (SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.TABCONST catalog view for the row that describes the constraint.

**FUNCTION**
> Indicates a comment will be added or replaced for a function. The function instance specified must be a user-defined function or function template described in the catalog.
>
> There are several different ways available to identify the function instance:
>
> **FUNCTION** *function-name*
> > Identifies the particular function, and is valid only if there is exactly one function with the *function-name*. The function thus identified may 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 specific instance of the function in the named or implied schema, an error (SQLSTATE 42854) is raised.

**FUNCTION** *function-name* **(***data-type***,...)**

> Provides the function signature, which uniquely identifies the function to be commented upon. The function selection algorithm is *not* used.

> *function-name*
>> Gives the function name of the function to be commented upon. 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***,...)**
>> Must match the data types that were specified on the CREATE FUNCTION statement in the corresponding position. The number of data types, and the logical concatenation of the data types is used to identify the specific function for which to add or replace the comment.

>> If the *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 may be coded to indicate that these attributes are to be ignored when looking for a data type match.

>> FLOAT() cannot be used (SQLSTATE 42601) since 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.

>> A type of FLOAT(n) does not need to match the defined value for n since 0 <n<25 means REAL and 24<n<54 means DOUBLE. Matching occurs based on whether the type is REAL or DOUBLE.

>> (Note that the FOR BIT DATA attribute is not considered part of the signature for matching purposes. So, for example, a CHAR FOR BIT DATA specified in the signature would match a function defined with CHAR only, and vice versa.)

> If no function with the specified signature exists in the named or implied schema, an error (SQLSTATE 42883) is raised.

**SPECIFIC FUNCTION** *specific-name*

> Indicates that comments will be added or replaced for a function (see FUNCTION for other methods of identifying a function). Identifies the particular user-defined function that is to be commented upon, 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/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.

It is not possible to comment on a function that is either in the SYSIBM
schema or the SYSFUN schema (SQLSTATE 42832).

The comment replaces the value of the REMARKS column of the
SYSCAT.FUNCTIONS 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 is
described in the catalog (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.

**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 is described in the catalog (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.

**NICKNAME** *nickname*
Indicates a comment will be added or replaced for a nickname. The
*nickname* must be a nickname that is described in the catalog (SQLSTATE
42704). The comment replaces the value for the REMARKS column of the
SYSCAT.TABLES catalog view for the row that describes the nickname.

**NODEGROUP** *nodegroup-name*
Indicates a comment will be added or replaced for a nodegroup. The
*nodegroup-name* must identify a distinct nodegroup that is described in the
catalog (SQLSTATE 42704). The comment replaces the value for the
REMARKS column of the SYSCAT.NODEGROUPS catalog view for the
row that describes the nodegroup.

**PACKAGE** *package-name*
Indicates a comment will be added or replaced for a package. The
*package-name* must identify a distinct package that is described in the

catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.PACKAGES catalog view for the row that describes the package.

**PROCEDURE**

Indicates a comment will be added or replaced for a procedure. The procedure instance specified must be a stored procedure described in the catalog.

There are several different ways available to identify the procedure instance:

**PROCEDURE** *procedure-name*

Identifies the particular procedure, and is valid only if there is exactly one procedure with the *procedure-name* in the schema. The procedure thus identified may 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 specific instance of the procedure in the named or implied schema, an error (SQLSTATE 42854) is raised.

**PROCEDURE** *procedure-name* **(***data-type,...***)**

This is used to provide the procedure signature, which uniquely identifies the procedure to be commented upon.

*procedure-name*

Gives the procedure name of the procedure to be commented upon. 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,...***)**

Must match the data types that were specified on the CREATE PROCEDURE statement in the corresponding position. The number of data types, and the logical concatenation of the data types is used to identify the specific procedure for which to add or replace the comment.

If the *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

may be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601) since 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 PROCEDURE statement.

A type of FLOAT(n) does not need to match the defined value for n since 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 raised.

**SPECIFIC PROCEDURE** *specific-name*
Indicates that comments will be added or replaced for a procedure (see PROCEDURE for other methods of identifying a procedure). Identifies the particular stored procedure that is to be commented upon, 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/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.

The comment replaces the value of the REMARKS column of the SYSCAT.PROCEDURES catalog view for the row that describes the procedure.

**SCHEMA** *schema-name*
Indicates a comment will be added or replaced for a schema. The *schema-name* must identify a schema that is described in the catalog (SQLSTATE 42704). The comment replaces the value of the REMARKS column of the SYSCAT.SCHEMATA catalog view for the row that describes the schema.

**SERVER** *server-name*
Indicates a comment will be added or replaced for a data source. The *server-name* must identify a data source that is described in the catalog (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 is described in the catalog (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 is described in the catalog.

**TYPE** *server-type*
Specifies the type of data source—for example, DB2 Universal Database for OS/390 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.*

**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 distinct table or view (not an alias) that is described in the catalog (SQLSTATE 42704). 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 is described in the catalog (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.

**TRIGGER** *trigger-name*

Indicates a comment will be added or replaced for a trigger. The *trigger-name* must identify a distinct trigger that is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.TRIGGERS catalog view for the row that describes the trigger.

**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 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). 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 is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.TYPEMAPPINGS catalog view for the row that describes the mapping.

**WRAPPER** *wrapper-name*

Indicates a comment will be added or replaced for a wrapper. The *wrapper-name* must identify a wrapper that is described in the catalog (SQLSTATE 42704). The comment replaces the value for the REMARKS column of the SYSCAT.WRAPPERS catalog view for the row that describes the wrapper.

**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 ON 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 base table or view, and the table or view must be described in the catalog.

    A comment cannot be made on a column of an inoperative view (SQLSTATE 51024).

## 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 CENTRE function, which he created in his PELLOW schema, using the signature to identify the specific function to be commented on.
```
COMMENT ON FUNCTION CENTRE (INT,FLOAT)
    IS 'Frank''s CENTRE fctn, uses Chebychev method'
```

*Example 6:* McBride wants to comment on another CENTRE 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 CENTRE 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'
```

# COMMENT ON

*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.'
```

## 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

```
         ┌─WORK─┐
►►──COMMIT─┴──────┴──────────────────────────────────────────►◄
```

### 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 ON, CREATE, DELETE, DROP, GRANT, INSERT, LOCK TABLE, REVOKE, SET INTEGRITY, SET transition-variable, and UPDATE.

The following statements, however, are not under transaction control and changes made by them are independent of issuing the COMMIT statement:
* SET CONNECTION,
* SET CURRENT DEGREE,
* SET CURRENT EXPLAIN MODE,
* SET CURRENT EXPLAIN SNAPSHOT,
* SET CURRENT PACKAGESET,
* SET CURRENT QUERY OPTIMIZATION,
* SET CURRENT REFRESH AGE,
* SET EVENT MONITOR STATE,
* SET PASSTHRU,
* SET PATH,
* SET SCHEMA,
* SET SERVER OPTION.

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

## COMMIT

WITH HOLD remain open, and the cursor is positioned before the next logical row of the result table.[59] 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.

### 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. Refer to *Application Development Guide* for implicitly ending a transaction in different application environments.

See the "Notes" on page 771 for impact of COMMIT on cached dynamic SQL statements.

### Example

Commit alterations to the database made since the last commit point.
```
COMMIT WORK
```

---

59. A FETCH must be performed before a Positioned UPDATE or DELETE statement is issued.

## Compound SQL

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 statement construct is an executable statement that cannot be dynamically prepared. The statement is not supported in REXX.

### Authorization

None for the Compound SQL statement itself. The authorization ID of the Compound SQL statement must have the appropriate authorization on all the individual statements that are contained within the Compound SQL statement.

### Syntax



### Description

**ATOMIC**
> Specifies that, if any of the sub-statements within the Compound SQL statement fails, 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 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 ...
> ```

# Compound SQL

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 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 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 a Compound SQL statement:

| | |
|---|---|
| CALL | FETCH |
| CLOSE | OPEN |
| CONNECT | PREPARE |
| Compound SQL | RELEASE |
| DESCRIBE | ROLLBACK |
| DISCONNECT | SET CONNECTION |
| EXECUTE IMMEDIATE | |

If a COMMIT statement is included, it must be the last sub-statement. If COMMIT is in this position (for example the last of a hundred statements), it will be issued even if the STOP AFTER FIRST *host-variable* STATEMENT option indicates that only the first fifty statements are to be executed. In this case, it will be treated as the fifty-first 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 statement; that is, no host language code is allowed between the sub-statements that make up the Compound SQL statement.
- Only NOT ATOMIC Compound SQL statements will be accepted by DDCS.
- Compound SQL statements cannot be nested.

### Notes

One SQLCA is returned for the entire Compound SQL 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, then that warning is given precedence over any other warning in order that a WHENEVER NOT FOUND exception can be acted upon.[60]
- 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 on up to a maximum of seven of these 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 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. [61] This field is left-justified and padded with blanks.

**X**      The token separator X'FF'.

---

60. 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'. If there is more than one 'no data found' warning within the Compound SQL statement, the fields for the last sub-statement will be the fields returned.

61. If the number would exceed 999, counting restarts at zero.

**sss** The ordinal position of the statement that caused the error. [61] For example, if the first statement failed, this field would contain the number one left-justified ('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 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 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;
```

## 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 TO statement. A connection lasts until a CONNECT RESET statement or a DISCONNECT statement is issued or until another CONNECT TO statement changes the application server.

See "Remote Unit of Work Connection Management" on page 30 for concepts and additional details on connection states. See "Options that Govern Distributed Unit of Work Semantics" on page 37 for the precompiler options that determine the framework for CONNECT behavior.

### 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

The authorization ID of the statement must be authorized to connect to the identified application server. Depending on the authentication setting for the database, the authorization check may be performed by either the client or the server. For a partitioned database, the user and group definitions must be identical across partitions or nodes. Refer to the AUTHENTICATION database manager configuration parameter in the *Administration Guide* for information about the authentication setting.

### Syntax

```
►►──CONNECT──────────────────────────────────────────────────►
```

```
►►──┬─TO──┬─server-name────┬──┬──────────────┬──┬───────────────┬─┬──►◄
    │     └─host-variable──┘  └─ lock-block ──┘  └─ authorization ┘ │
    ├─RESET─────────────────────────────────────────────────────────┤
    │                    (1)                                         │
    └─┤ authorization ├───────────────────────────────────────────┘
```

**authorization:**

```
├──USER──┬─authorization-name─┬──USING──┬─password───────┬──────────►
         └─host-variable──────┘         └─host-variable──┘
```

```
►──┬──────────────────────────────────────────────────┬──────────┤
   └─NEW──┬─password───────┬──CONFIRM──password────────┘
          └─host-variable──┘
```

**lock-block:**

```
├──┬─IN SHARE MODE──────────────────────────────┬───────────────────┤
   └─IN EXCLUSIVE MODE─┬──────────────────┬──────┘
                       └─ON SINGLE NODE───┘
```
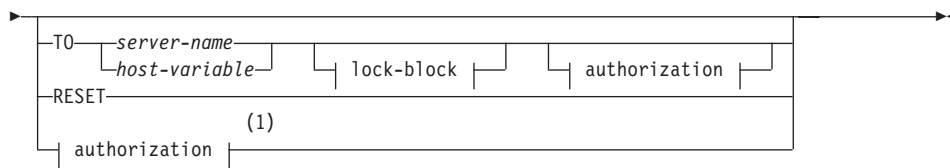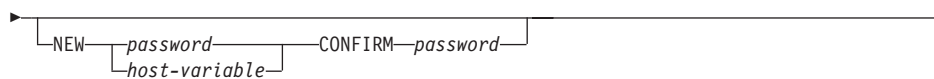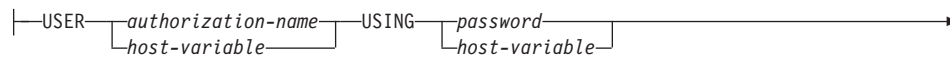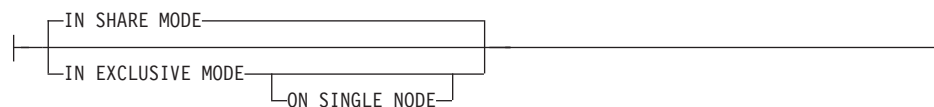
**Notes:**

1    This form is only valid if implicit connect is enabled.

## 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 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 country code and code page of the application server are placed in the SQLERRMC field (as they are with a successful CONNECT TO statement).

This form of CONNECT:
• Does not require the application process to be in the connectable state.

## CONNECT (Type 1)

- 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 code and code page of the application server are placed in the SQLERRMC field, like a successful CONNECT TO 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-justified 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.

**Note:** DB2 for MVS supports a 16 byte location-name and both SQL/DS and DB2/400 support a 18 byte target database name. DB2 Version 6 only supports the use of 8 byte database-alias name on the SQL CONNECT statement. However, the database-alias name can be mapped to an 18 byte database name through the Database Connection Service Directory.

When the CONNECT TO statement is executed, the application process must be in the connectable state (see "Remote Unit of Work Connection Management" on page 30 for information about connection states with Type 1 CONNECT).

*Successful Connection:*

If the CONNECT TO 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* identifies the product as follows:

```
DSN for DB2 for MVS
ARI for SQL/DS
QSQ for DB2/400
SQL for DB2 Universal Database
```

– *vv* is a two-digit version identifier such as '02'

– *rr* is a two-digit release identifier such as '01'

– *m* is a one-digit modification level identifier such as '0'.

For example, if the application server is Version 1 Release 1 of DB2 for OS/2, the value of SQLERRP is 'SQL01010'.[62]

- The SQLERRMC field of the SQLCA is set to contain the following values (separated by X'FF')

    1. the country code of the application server (or blanks if using DDCS),

    2. the code page of the application server (or CCSID if using DDCS),

    3. the authorization ID,

    4. the database alias,

    5. the platform type of the application server. Currently identified values are:

        | Token | Server |
        |---|---|
        | **QAS** | DB2/400 |
        | **QDB2** | DB2 for MVS |
        | **QDB2/2** | DB2 Universal Database for OS/2 |
        | **QDB2/6000** | DB2 Universal Database for AIX |
        | **QDB2/HPUX** | DB2 Universal Database for HP-UX |
        | **QDB2/NT** | DB2 Universal Database for NT |
        | **QDB2/SNI** | DB2 Universal Database for Siemens Nixdorf |
        | **QDB2/SUN** | DB2 Universal Database for Solaris |
        | **QOS/2 DBM** | ES 1.0 DBM (note that there is a single space between QOS/2 and DBM) |
        | **QSQLDS/VM** | SQL/DS for VM |
        | **QSQLDS/VSE** | SQL/DS 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.

---

62. This release of DB2 Universal Database Version 6 is 'SQL06010'.

8. Partition number. For a non-partitioned database, this is always 0, if present.
9. The code page of the application client.
10. Number of partitions in a partitioned database. If the database cannot be partitioned, the value is 0 (zero). Token is present only with Version 5 or later.

- 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. [63]

- 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. [63]

- 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 down-level client which 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 TP Monitor environment).
  - **3 -** two-phase commit.

- The SQLERRD(5) field of the SQLCA returns the authentication type of the connection. The value is one of:
  - **0 -** Authenticated on the server.
  - **1 -** Authenticated on the client.
  - **2 -** Authenticated using DB2 Connect.
  - **3 -** Authenticated using Distributed Computing Environment security services.
  - **255 -** Authentication not specified.

---

63. See the "Character Conversion Expansion Factor" section of the "Programming in Complex Environments" chapter in the *Application Development Guide* for details.

See "Controlling Database Access" in the *Administration Guide* for details on authentication types.
- The SQLERRD(6) field of the SQLCA returns the partition number of the partition to which the connection was made if the database is partitioned. Otherwise, a value of 0 is returned.

*Unsuccessful Connection:*

If the CONNECT TO 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. Note that the first three characters of the module name identifies the product. For example, if the application requester is on the OS/2 database manager, the first three characters are 'SQL'.
- If the CONNECT TO 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 TO 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 TO 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** [64]
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.

**ON SINGLE NODE**
Specifies that the coordinator partition is connected in exclusive mode and all other partitions are connected in share mode. This option is only effective in a partitioned database.

---

64. This option is not supported by DDCS.

# CONNECT (Type 1)

**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 userid trying to connect to the application server. 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 userid that is contained within the *host-variable* must be left justified and must not be delimited by quotation marks.

**USING** *password/host-variable*
Identifies the password of the userid trying to connect to the application server. *Password* or *host-variable* may be up to 18 characters. If a host variable is specified, it must be a character string variable with a length attribute not greater than 18 and it must not include an indicator variable.

**NEW** *password/host-variable* **CONFIRM** *password*
Identifies the new password that should be assigned to the userid identified by the USER option. *Password* or *host-variable* may be up to 18 characters. If a host variable is specified, it must be a character string variable with a length attribute not greater than 18 and it must not include an indicator variable. The system on which the password will be changed depends on how user authentication is set up.

## Notes

- It is good practice for the first SQL statement executed by an application process to be the CONNECT TO statement.
- If a CONNECT TO statement is issued to the current application server with a different userid 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 TO statement is issued to the current application server with the same userid and password then the conversation is not deallocated and reallocated. Cursors, in this case, are not closed.
- To use DB2 Universal Database Enterprise - Extended Edition, the user or application must connect to one of the partitions listed in the `db2nodes.cfg` file (see "Data Partitioning Across Multiple Partitions" on page 57 for information about this file). You should try to ensure that not all users use the same partition as the coordinator partition.

## Examples

*Example 1:* In a C program, connect to the application server TOROLAB3, where TOROLAB3 is a database alias of the same name, with the userid FERMAT and the password THEOREM.

```
EXEC SQL  CONNECT TO TOROLAB3 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);
```

## 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.

See "Application-Directed Distributed Unit of Work" on page 34 for concepts and additional details.

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

The invocation is the same as "Invocation" on page 512.

### Authorization

The authorization is the same as "Authorization" on page 512.

### Syntax

The syntax is the same as "Syntax" on page 512. The selection between Type 1 and Type 2 is determined by precompiler options. See "Options that Govern Distributed Unit of Work Semantics" on page 37 for an overview of these options. Further details are provided in the *Command Reference* and *Administrative API Reference* manuals.

### 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.

See "Options that Govern Distributed Unit of Work Semantics" on page 37 for information about the specification of SQLRULES.

*Successful Connection*

If the CONNECT TO 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 Type 1 CONNECT; see page 514.

*Unsuccessful Connection*

If the CONNECT TO 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 userid 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. See the *Quick Beginnings* for information on setting implicit connect on installation and environment variables, and the *Administration Guide* for information on authentication settings.

**Rules**

- As outlined in "Options that Govern Distributed Unit of Work Semantics" on page 37 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 raised. 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 on page 525 illustrates these rules.

- Although the CONNECT TO statement can be used to establish or switch connections, CONNECT TO 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.

### 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.

**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" on page 37). 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 below:

Use of **CONNECT TO**:

| 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. |

| Type 1 | Type 2 |
|---|---|
| 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. |

## CONNECT (Type 2)

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 also implicitly commits the current unit of work. | CONNECT RESET does not commit 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.<br><br>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. |

## Examples

*Example 1:* This example illustrates the use of multiple source programs (shown in the boxes), some preprocessed with different connection options (shown above 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;
   ...
```

1 Note: 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;
   ...
```

# CONNECT (Type 2)

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 |

## CONNECT (Type 2)

| Sequence | Statement | Current Server | Dormant Connections | Release Pending |
|---|---|---|---|---|
| 9 | DISCONNECT S1 SELECT * FROM TBLF | S2 S2 | S0 S0 | None None |

## CREATE ALIAS

The CREATE ALIAS statement defines an alias for a table, view, nickname, or another alias.

### 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 as least one of the following:
* SYSADM or DBADM authority
* IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the alias does not exist
* CREATEIN privilege on the schema, if the schema name of the alias refers to an existing schema.

To use the referenced object via the alias, the same privileges are required on that object as would be necessary if the object itself were used.

### Syntax

```
►►──CREATE──┬─ALIAS────────┬──alias-name──FOR──┬─table-name──┬──────────►◄
            │       (1)    │                   ├─view-name───┤
            └─SYNONYM──────┘                   ├─nickname────┤
                                               └─alias-name2─┘
```
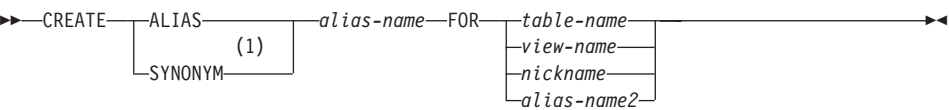
**Notes:**

**1**    CREATE SYNONYM is accepted as an alternative for CREATE ALIAS
       for  syntax toleration of existing CREATE SYNONYM statements of other
       SQL implementations.

### Description

*alias-name*
    Names the alias. The name must not identify a table, view, nickname, or alias that exists in the current database.

    If a two-part name is specified, the schema name cannot begin with "SYS" (SQLSTATE 42939).

    The rules for defining an alias name are the same as those used for defining a table name.

**FOR** *table-name, view-name, nickname,* or *alias-name2*
>  Identifies the table, view, nickname, or 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).

### Notes

- The definition of the newly created alias is stored in SYSCAT.TABLES.
- 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 view definition, a statement in a package, or a trigger points to an alias chain, then a dependency is recorded for the view, package, or trigger on each alias in the chain. Repetitive cycles in an alias chain are not allowed and are detected at alias definition time.
- 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.

### 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).

## CREATE ALIAS

*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
```

## CREATE BUFFERPOOL

The CREATE BUFFERPOOL statement creates a new buffer pool to be used by the database manager. Although the buffer pool definition is transactional and the entries will be reflected in the catalog tables on commit, the buffer pool will not become active until the next time the database is started.

In a partitioned database, a default buffer pool definition is specified for each partition or node, with the capability to override the size on specific partitions or nodes. Also, in a partitioned database, the buffer pool is defined on all partitions unless nodegroups are specified. If nodegroups are specified, the buffer pool will only be created on partitions that are in those nodegroups.

### Invocation

This statement can be embedded in an application program or issued interactively. 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 authorization ID of the statement must have SYSCTRL or SYSADM authority.

### Syntax

```
►►──CREATE──BUFFERPOOL──bufferpool-name───┬─ALL NODES──────────────────────────┬──►
                                          │                     ┌─,◄───────┐    │
                                          └─NODEGROUP──▼─nodegroup-name─┴──┘

►──SIZE──number-of-pages──┬──────────────────────┬──────●─────────────────────►
                          └─ except-on-nodes-clause ┘

►──┬─PAGESIZE──4096───────────┬──●──┬─NOT EXTENDED STORAGE─┬──●──►◄
   └─PAGESIZE──integer──┬───┬─┘     └─EXTENDED STORAGE──────┘
                        └─K─┘
```
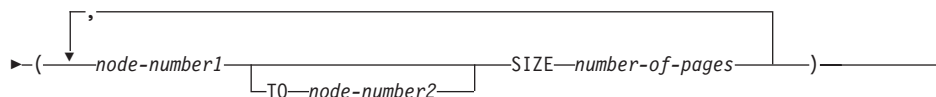
**except-on-nodes-clause:**

```
├──EXCEPT ON──┬─NODE──┬──────────────────────────────────────────────────────►
              └─NODES─┘
```

## CREATE BUFFERPOOL

```
►─(──┬──node-number1───┬──────────────────────SIZE──number-of-pages─────)──────────┤
      │                 │
      └─TO──node-number2─┘
```

### 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 a catalog (SQLSTATE 42710). The *bufferpool-name* must not begin with the characters ″SYS″ or ″IBM″ (SQLSTATE 42939).

**ALL NODES**
> This buffer pool will be created on all partitions in the database.

**NODEGROUP** *nodegroup-name, ...*
> Identifies the nodegroup or nodegroups to which the buffer pool definition is applicable. If this is specified, this buffer pool will only be created on partitions in these nodegroups. Each nodegroup must currently exist in the database (SQLSTATE 42704). If the NODEGROUP keyword is not specified, then this buffer pool will be created on all partitions (and any partitions subsequently added to the database).

**SIZE** *number-of-pages*
> The size of the buffer pool specified as the number of pages.[65] In a partitioned database, this will be the default size for all partitions where the buffer pool exists.

*except-on-nodes-clause*
> Specifies the partition or partitions for which the size of the buffer pool will be different than the default. If this clause is not specified, then all partitions will have the same size as specified for this buffer pool.

> **EXCEPT ON NODES**
>> Keywords that indicate that specific partitions are specified. NODE is a synonym for NODES.

>> *node-number1*
>>> Specifies a specific partition number that is included in the partitions for which the buffer pool is created.

>> **TO** *node-number2*
>>> Specify a range of partition numbers. The value of *node-number2* must be greater than or equal to the value of *node-number1*

---

65. The size can be specified with a value of (-1) which will indicate that the buffer pool size should be taken from the BUFFPAGE database configuration parameter.

(SQLSTATE 428A9). All partitions between and including the specified partition numbers must be included in the partitions for which the buffer pool is created (SQLSTATE 42729).

**SIZE** *number-of-pages*
The size of the buffer pool specified as the number of pages.

**PAGESIZE** *integer* **[K]**
Defines the size of pages used for the bufferpool. The valid values for *integer* without the suffix K are 4 096, 8 192, 16 384 or 32 768. The valid values for *integer* with the suffix K are 4, 8, 16 or 32. An error occurs if the page size is not one of these values (SQLSTATE 428DE). The default is 4 096 byte (4K) pages. Any number of spaces is allowed between *integer* and K, including no space.

**EXTENDED STORAGE**
If the extended storage configuration is turned on,[66] pages that are being migrated out of this buffer pool will be cached in the extended storage.

**NOT EXTENDED STORAGE**
Even if the database extended storage configuration is turned on, pages that are being migrated out of this buffer pool, will NOT be cached in the extended storage.

**Notes**

- Until the next time the database is started, any table space that is created will use an already active buffer pool of the same page size. The database has to be restarted for the table space assignment to the new buffer pool to take effect.

- 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 DB2 is unable to obtain the total memory for all buffer pools, it will attempt to start up only the default buffer pool. If this is unsuccessful, it will start up a minimal default buffer pool. In either of these cases, a warning will be returned to the user (SQLSTATE 01626) and the pages from all table spaces will use the default buffer pool.

---

66. Extended storage configuration is turned on by setting the database configuration parameters
NUM_ESTORE_SEGS and ESTORE_SEG_SIZE to non-zero values. See *Administration Guide* for details.

## CREATE DISTINCT TYPE

The CREATE DISTINCT TYPE statement defines a distinct type. The distinct type is always sourced on one of the built-in data types. Successful execution of the statement 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.

### 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 as least one of the following:
- SYSADM or DBADM authority
- 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.

### Syntax

```
►►──CREATE DISTINCT TYPE──distinct-type-name──AS──────────────────►

                                            (1)
►─┤ source-data-type ├──WITH COMPARISONS──────────────────────►◄
```

**source-data-type:**

```
├─┬─SMALLINT──────────────────────────────────────────┬─┤
  ├─INTEGER──┐                                         │
  │          ├────────────────────────────────────────┤
  │ └─INT────┘                                         │
  ├─BIGINT─────────────────────────────────────────────┤
  ├─FLOAT──┬────────────────┬──────────────────────────┤
  │        └─(─integer─)─────┘                          │
  ├─REAL───────────────────┐                            │
  │        ┌─PRECISION─┐    │                           │
  ├─DOUBLE─┴───────────┴────┘                           │
  ├─DECIMAL──┐                                          │
  ├─DEC──────┤    ┌──────────────────┐                  │
  ├─NUMERIC──┼──(─integer──┬─────────┬─)────────────────┤
  ├─NUM──────┘             └─,integer─┘                 │
  ├──┬─CHARACTER─┬──┬────────────┬──┬─FOR BIT DATA─┐    │
  │  └─CHAR──────┘  └─(integer)──┘                  │    │
  ├─VARCHAR(integer)───────────────────────────────────┤
  ├─LONG VARCHAR───────────────────────────────────────┤
  ├─GRAPHIC──┬────────────┬────────────────────────────┤
  │          └─(integer)──┘                             │
  ├─VARGRAPHIC(integer)────────────────────────────────┤
  ├─LONG VARGRAPHIC────────────────────────────────────┤
  ├─DATE───────────────────────────────────────────────┤
  ├─TIME───────────────────────────────────────────────┤
  ├─TIMESTAMP──────────────────────────────────────────┤
  ├─BLOB──┐                                             │
  ├─CLOB──┼──(─integer──┬───┬──)───────────────────────┤
  ├─DBCLOB─┘            ├─K─┤                            │
  │                     ├─M─┤                            │
  │                     └─G─┘                            │
  └─DATALINK──┬────────────┬───────────────────────────┘
              └─(integer)──┘
```
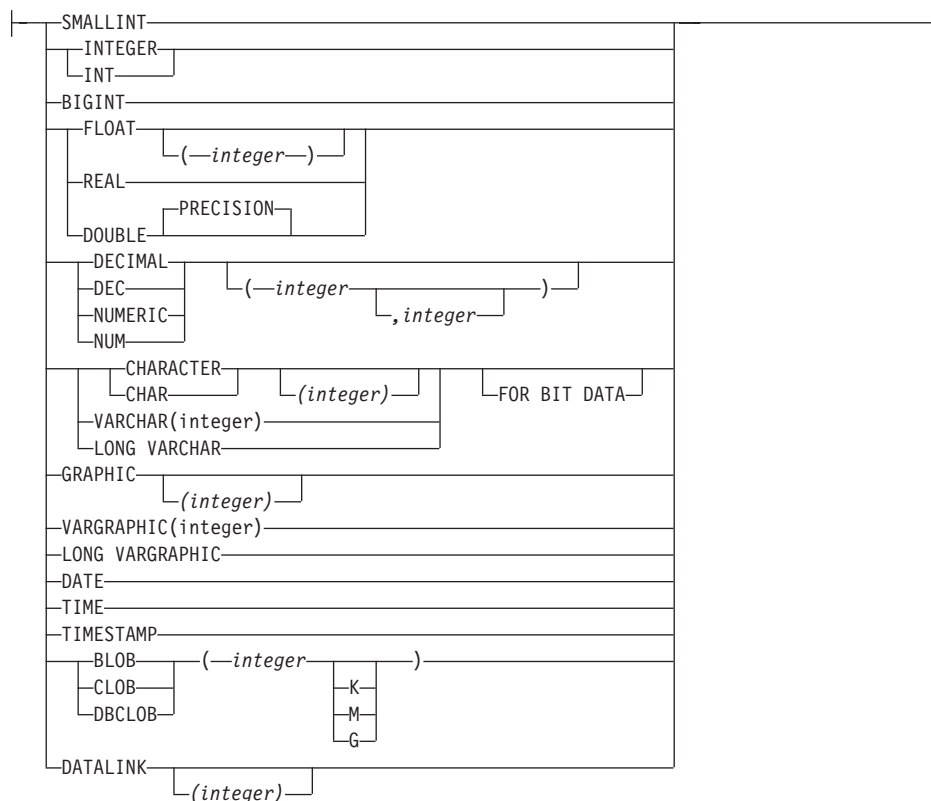
**Notes:**

1   Required for all source-data-types except LOBs, LONG VARCHAR, LONG VARGRAPHIC and DATALINK which are not supported.

## Description

*distinct-type-name*
> Names the distinct type. The name, including the implicit or explicit qualifier must not identify a distinct type described in the catalog. The unqualified name must not be the same as the name of a source-data-type or BOOLEAN (SQLSTATE 42918).
>
> 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.
>
> A number of names used as keywords in predicates are reserved for system use, and may not be used as a *distinct-type-name*. The names are

> SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH and the comparison operators as described in "Basic Predicate" on page 167. Failure to observe this rule will lead to an error (SQLSTATE 42939).
>
> If a two-part *distinct-type-name* is specified, the schema name cannot begin with ″SYS″; otherwise, an error (SQLSTATE 42939) is raised.

**source-data-type**
Specifies the data type used as the basis for the internal representation of the distinct type. For information about the association of distinct types with other data types, see "Distinct Types" on page 84. For information about data types, see "CREATE TABLE" on page 639.

**WITH COMPARISONS**
Specifies that system-generated comparison operators are to be created for comparing two instances of a distinct type. These keywords should not be specified if the source-data-type is BLOB, CLOB, DBCLOB, LONG VARCHAR, LONG VARGRAPHIC, or DATALINK, otherwise a warning will be returned (SQLSTATE 01596) and the comparison operators will not be generated. For all other source-data-types, the WITH COMPARISONS keywords are required.

### Notes

- 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.

- 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 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 20 for details).
The type of the return value of this function will include the parameters given
on the CREATE DISTINCT TYPE 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 DISTINCT TYPE T_SHOESIZE AS CHAR(2)
          WITH COMPARISONS

CREATE DISTINCT TYPE T_MILES AS DOUBLE
          WITH COMPARISONS
```

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 20. CAST functions on distinct types

| Source Type Name | Function Name | Parameter | Return-type |
| --- | --- | --- | --- |
| CHAR | <distinct> | CHAR (n) | <distinct> |
| | CHAR | <distinct> | CHAR (n) |
| | <distinct> | VARCHAR (n) | <distinct> |
| VARCHAR | <distinct> | VARCHAR (n) | <distinct> |
| | VARCHAR | <distinct> | VARCHAR (n) |
| LONG VARCHAR | <distinct> | LONG VARCHAR | <distinct> |
| | LONG_VARCHAR | <distinct> | LONG VARCHAR |
| CLOB | <distinct> | CLOB (n) | <distinct> |
| | CLOB | <distinct> | CLOB (n) |
| BLOB | <distinct> | BLOB (n) | <distinct> |
| | BLOB | <distinct> | BLOB (n) |

## CREATE DISTINCT TYPE

*Table 20. CAST functions on distinct types  (continued)*

| Source Type Name | Function Name | Parameter | Return-type |
|---|---|---|---|
| GRAPHIC | \<distinct\> | GRAPHIC (n) | \<distinct\> |
| | GRAPHIC | \<distinct\> | GRAPHIC (n) |
| | \<distinct\> | VARGRAPHIC (n) | \<distinct\> |
| VARGRAPHIC | \<distinct\> | VARGRAPHIC (n) | \<distinct\> |
| | VARGRAPHIC | \<distinct\> | VARGRAPHIC (n) |
| LONG VARGRAPHIC | \<distinct\> | LONG VARGRAPHIC | \<distinct\> |
| | LONG_VARGRAPHIC | \<distinct\> | LONG VARGRAPHIC |
| DBCLOB | \<distinct\> | DBCLOB (n) | \<distinct\> |
| | DBCLOB | \<distinct\> | DBCLOB (n) |
| SMALLINT | \<distinct\> | SMALLINT | \<distinct\> |
| | \<distinct\> | INTEGER | \<distinct\> |
| | SMALLINT | \<distinct\> | SMALLINT |
| INTEGER | \<distinct\> | INTEGER | \<distinct\> |
| | INTEGER | \<distinct\> | INTEGER |
| BIGINT | \<distinct\> | BIGINT | \<distinct\> |
| | BIGINT | \<distinct\> | BIGINT |
| DECIMAL | \<distinct\> | DECIMAL (p,s) | \<distinct\> |
| | DECIMAL | \<distinct\> | DECIMAL (p,s) |
| NUMERIC | \<distinct\> | DECIMAL (p,s) | \<distinct\> |
| | DECIMAL | \<distinct\> | DECIMAL (p,s) |
| REAL | \<distinct\> | REAL | \<distinct\> |
| | \<distinct\> | DOUBLE | \<distinct\> |
| | REAL | \<distinct\> | REAL |
| FLOAT($n$) where $n<=24$ | \<distinct\> | REAL | \<distinct\> |
| | \<distinct\> | DOUBLE | \<distinct\> |
| | REAL | \<distinct\> | REAL |
| FLOAT($n$) where $n>24$ | \<distinct\> | DOUBLE | \<distinct\> |
| | DOUBLE | \<distinct\> | DOUBLE |
| FLOAT | \<distinct\> | DOUBLE | \<distinct\> |
| | DOUBLE | \<distinct\> | DOUBLE |
| DOUBLE | \<distinct\> | DOUBLE | \<distinct\> |
| | DOUBLE | \<distinct\> | DOUBLE |

*Table 20. CAST functions on distinct types (continued)*

| Source Type Name | Function Name | Parameter | Return-type |
|---|---|---|---|
| DOUBLE PRECISION | <distinct> | DOUBLE | <distinct> |
| | DOUBLE | <distinct> | DOUBLE |
| DATE | <distinct> | DATE | <distinct> |
| | DATE | <distinct> | DATE |
| TIME | <distinct> | TIME | <distinct> |
| | TIME | <distinct> | TIME |
| TIMESTAMP | <distinct> | TIMESTAMP | <distinct> |
| | TIMESTAMP | <distinct> | TIMESTAMP |
| DATALINK | <distinct> | DATALINK | <distinct> |
| | DATALINK | <distinct> | DATALINK |

**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 above table are the only functions that are generated automatically when distinct types are defined. Consequently, none of the built-in functions (AVG, MAX, LENGTH, etc.) are supported on distinct types until the CREATE FUNCTION statement (see "CREATE FUNCTION" on page 551) is used to register user-defined functions for the distinct type, where 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 distinct type is created using the WITH COMPARISONS clause, system-generated comparison operators are created. Creation of these comparison operators will generate entries in the SYSCAT.FUNCTIONS catalog view for the new functions.

The schema name of the distinct type must be included in the SQL path (see "SET PATH" on page 892 or the FUNCPATH BIND option as described in the *Application Development Guide*) for successful use of these operators and cast functions in SQL statements.

## Examples

*Example 1:* Create a distinct type named SHOESIZE that is based on an INTEGER data type.

```
CREATE DISTINCT TYPE SHOESIZE AS INTEGER WITH COMPARISONS
```

# CREATE DISTINCT TYPE

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 distinct type named MILES that is based on a DOUBLE data type.

```
CREATE DISTINCT TYPE MILES AS DOUBLE WITH COMPARISONS
```

This will also result in the creation of comparison operators (=, <>, <, =, >, >=) and cast functions DOUBLE(MILES) returning DOUBLE and MILES(DOUBLE) returning MILES.

## CREATE EVENT MONITOR

The CREATE EVENT MONITOR statement defines a monitor that will record certain events that occur when using the database. The definition of each 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. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

### Authorization

The privileges held by the authorization ID must include either SYSADM or DBADM authority (SQLSTATE 42502).

### Syntax
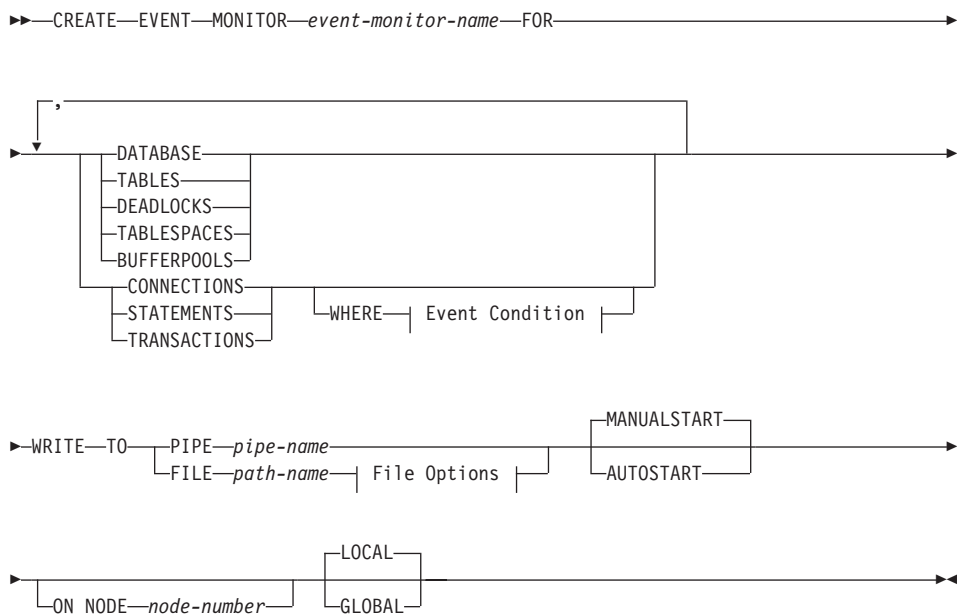


**Event Condition:**

# CREATE EVENT MONITOR

```
        ┌─AND │ OR───────────────────────────────────────────────┐
        │                                                         │
├───┬────────────────────────┬─────────────────────────────────────────────────────┤
    │            ┌─APPL_ID───┐  ┌─=──────────┐  ┌─comparison-string─┐
    └─NOT─┘      ├─AUTH_ID───┤  │      (1)   │
                └─APPL_NAME─┘  ├─<>─────────┤
                               ├─>──────────┤
                               │      (1)   │
                               ├─>=─────────┤
                               ├─<──────────┤
                               │      (1)   │
                               ├─<=─────────┤
                               ├─LIKE───────┤
                               └─NOT──LIKE──┘
             └─(Event Condition)──────────────────────────────────┘
```

**File Options:**

```
├──┬──────────────────────────────────┬──┬──────────────────────────┬──────────────────►
   │          ┌─NONE──────────┐        │  │            ┌─pages─┐      │
   └─MAXFILES─┴─number-of-files─┘       └─MAXFILESIZE──┴─NONE──┘
```
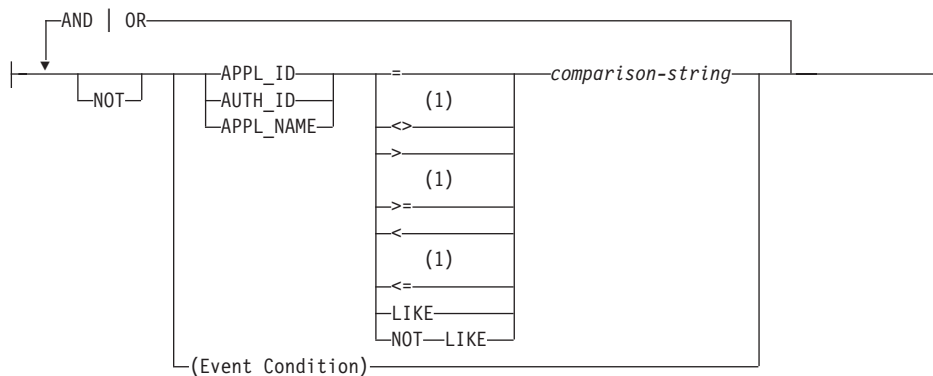
```
►──┬─────────────────────┬──┬─BLOCKED────┬──┬─APPEND──┬───────────────────────────────┤
   │                      │  └─NONBLOCKED─┘  └─REPLACE─┘
   └─BUFFERSIZE──pages────┘
```
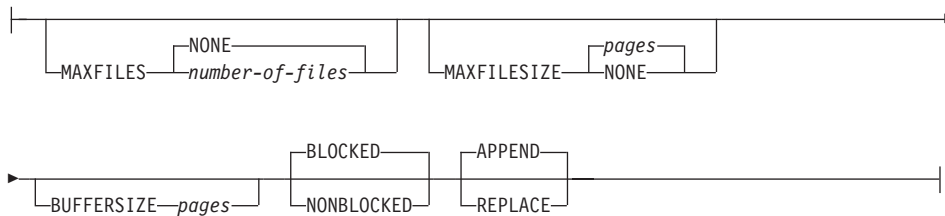
**Notes:**

**1**   Other forms of these operators are also supported. See "Basic Predicate" on page 167 for more details.

## Description

*event-monitor-name*
> Names 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.

> **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. An active table is a table that has changed since the first connection to the database.

**DEADLOCKS**
Specifies that the event monitor records a deadlock event whenever a deadlock occurs.

**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 a SQL statement finishes executing.

**TRANSACTIONS**
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 will generate 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 if an application will generate events for a particular event monitor, the WHERE clause is evaluated:

1. For each active connection when an event monitor is first turned on.
2. Subsequently 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 then all events of the specified event type will be monitored.

**CREATE EVENT MONITOR**

        **APPL_ID**

            Specifies that the application ID of each connection should be compared with the *comparison-string* in order to determine if the connection should generate CONNECTION, STATEMENT or TRANSACTION events (whichever was specified).

        **AUTH_ID**

            Specifies that the authorization ID of each connection should be compared with the *comparison-string* in order to determine if the connection should generate CONNECTION, STATEMENT or TRANSACTION events (whichever was specified).

        **APPL_NAME**

            Specifies that the application program name of each connection should be compared with the *comparison-string* in order to determine if the connection should generate 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, host variables and other string expressions are not permitted).

    **WRITE TO**

      Introduces the target for the data.

      **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. On UNIX operating systems pipe names are treated like file names. As a result, relative pipe names are permitted, and are treated like relative path-names (see *path-name* below). However, on OS/2, Windows 95 and Windows NT, there is a special syntax for a pipe

name. As a result, on OS/2, Windows 95 and Windows NT 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, 00000002.evt, etc). 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 partition or node (for example, in a UNIX-based system, this might be 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 (a path that starts with the root directory on AIX, or a disk identifier on OS/2, Windows 95 and Windows NT) is specified, then the specified path will be the one used. 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.

When a relative path is specified, the DB2EVENT directory is used to convert it into an absolute path. Thereafter, no distinction is made between absolute and relative paths. The absolute path is stored in the SYSCAT.EVENTMONITORS catalog view.

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:
- OS/2, Windows 95 and Windows NT - 200 4K pages
- UNIX - 1000 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 minimum and default size of each buffer (if this option is not specified) is 1 page (that is, 2 buffers, each 4 K in size). The maximum size of the buffers is limited by the size of the database heap (DBHEAP) since the buffers are allocated from the heap. If using a lot of event monitors at the same time, increase the size of the DBHEAP database 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 database heap (DBHEAP). For each active event monitor that has a pipe target, increase the size of the database heap by 2 pages.

**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.

**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 re-activated, 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 not be started automatically each time the database is started. Event monitors with the MANUALSTART option must be activated manually using the SET EVENT MONITOR STATE statement. This is the default option.

**AUTOSTART**

Specifies that the event monitor be started automatically each time the database is started.

**ON NODE**

Keyword that indicates that specific partitions are specified.

*node-number*

Specifies a partition number where the event monitor runs and write the events. With the monitoring scope defined as GLOBAL, all partitions report to the specified partition number. The I/O component will physically run on the specified partition, writing its records to /tmp/dlocks direcotry on that partition.

**GLOBAL**

Event monitor reports from all partitions. For a partitioned database in DB2 Universal Database Version 6, only deadlock event monitors can be defined as GLOBAL. The global event monitor will report deadlocks for all nodes in the system.

**LOCAL**

Event monitor reports only on the partition that is running. It gives a partial trace of the database activity. This is the default.

## Rules

- Each of the event types (DATABASE, TABLES, DEADLOCKs,...) can only be specified once 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.
- For detailed information on using the database monitor and on interpreting data from pipes and files, see the *System Monitor Guide and Reference*.

## Examples

*Example 1:* The following example creates an event monitor called SMITHPAY. This event monitor, will collect event data for the database as well as for the SQL statements performed by the PAYROLL application owned by the JSMITH authorization ID. The data will be appended to the absolute path /home/jsmith/event/smithpay/. A maximum of 25 files will be created. Each file will be a maximum of 1 024 4K pages long. The file I/O will be 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 will collect deadlock events and will write them to the relative path DLOCKS. One file will be written, and there is no maximum file size. Each time the event monitor is activated, it will append the event data to the file 00000000.evt if it exists. The event monitor will be started each time the database is started. The I/0 will be blocked by default.

```
CREATE EVENT MONITOR DEADLOCK_EVTS
   FOR DEADLOCKS
   WRITE TO FILE 'DLOCKS'
   MAXFILES 1
   MAXFILESIZE NONE
   AUTOSTART
```

## CREATE EVENT MONITOR

*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'
```

## CREATE FUNCTION

This statement is used to register a user-defined function or function template with an application server.

There are four 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. See "CREATE FUNCTION (External Scalar)" on page 552.

- 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. See "CREATE FUNCTION (External Table)" on page 570.

- 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. See "CREATE FUNCTION (OLE DB External Table)" on page 585.

- Source or template

  A source function is implemented by invoking another function (either built-in, external or source) that is already registered in the database. See "CREATE FUNCTION (Source or Template)" on page 593.

  It is possible to create a partial function, called a *function template*, that defines what types of values are to be returned but 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.

## CREATE FUNCTION (External Scalar)

This statement is used to register a user-defined external scalar function with an application 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. 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 at least one of the following:
- SYSADM or DBADM authority
- 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.

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:
- CREATE_NOT_FENCED authority on the database
- SYSADM or DBADM authority.

To create a fenced function, no additional authorities or privileges are required.

If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

### Syntax

```
►►──CREATE FUNCTION──function-name──(──┬─────────────────────────────┬──)──────►
                                        │      ┌─,──────────────┐     │
                                        └──▼──data-type1──┬──────────┘
                                                           └─AS LOCATOR─┘
```

```
►─●──RETURNS───data-type2──────────────────────────────────────────────●───────────►
         │                 └─AS LOCATOR─┘                               │
         └─data-type3──CAST FROM──data-type4───────────────────────────┘
                                        └─AS LOCATOR─┘

►──────────────────────────────●──EXTERNAL────────────────────────●─────────────────►
   └─SPECIFIC──specific-name─┘            └─NAME──┬─'string'──┬─┘
                                                 └─identifier─┘

                                                         ┌─NOT DETERMINISTIC─┐
►──LANGUAGE──┬─C────┬──●──PARAMETER STYLE──┬─DB2SQL─────┬──●──┤                ├──────►
             ├─JAVA─┤                      ├─DB2GENERAL─┤          └─DETERMINISTIC─┘ (1)
             └─OLE──┘                      └─JAVA───────┘

        ┌─FENCED─────┐  ┌─RETURNS NULL ON NULL INPUT─┐ (2)
►──●──┤            ├──┤                            ├──●──NO SQL──●─────────────────►
        └─NOT FENCED─┘  └─CALLED ON NULL INPUT───────┘

   ┌─EXTERNAL ACTION────┐   ┌─NO SCRATCHPAD─┐   ┌─NO FINAL CALL─┐
►──┤                  ├──●──┤             ├──●──┤             ├──●────────────────►
   └─NO EXTERNAL ACTION─┘   └─SCRATCHPAD────┘   └─FINAL CALL────┘

   ┌─ALLOW PARALLEL────┐   ┌─NO DBINFO─┐
►──┤                 ├──●──┤         ├──●────────────────────────────────────────►◄
   └─DISALLOW PARALLEL─┘   └─DBINFO────┘
```

**Notes:**

**1** NOT VARIANT may be specified in place of DETERMINISTIC and VARIANT may be specified in place of NOT DETERMINISTIC.

**2** NULL CALL may be specified in place of CALLED ON NULL INPUT and NOT NULL CALL may be specified in place of RETURNS NULL ON NULL INPUT.

## 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 (with a maximum length of 18). 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.

## CREATE FUNCTION (External Scalar)

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". Otherwise, an error (SQLSTATE 42939) is raised.

A number of names used as keywords in predicates are reserved for system use, and may not 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 as described in "Basic Predicate" on page 167. 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 column 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.

*(data-type1,...)*
Identifies the number of input parameters of the function, and specifies the data type 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. If this limit is exceeded, an error (SQLSTATE 54023) is raised.

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). 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 raises an SQL error (SQLSTATE 42723).

For example, given the statements:

```
CREATE FUNCTION PART (INT, CHAR(15)) ...
CREATE FUNCTION PART (INTEGER, CHAR(40)) ...

CREATE FUNCTION ANGLE (DECIMAL(12,2)) ...
CREATE FUNCTION ANGLE (DEC(10,7)) ...
```

the second and fourth statements would fail because they are considered to be duplicate functions.

*data-type1*
Specifies the data type of the parameter.

- SQL data type specifications and abbreviations which may be specified in the *data-type1* definition of a CREATE TABLE statement and have a correspondence in the language that is being used to write the function may be specified. See the language-specific sections of the *Application Development Guide* for details on the mapping between the SQL data types and host language data types with respect to user-defined functions.

- DECIMAL (and NUMERIC) are invalid with LANGUAGE C and OLE (SQLSTATE 42815). For alternatives to using DECIMAL refer to *Application Development Guide*.

- REF(*type-name*) cannot be specified as the data type of a parameter (SQLSTATE 42997).

**AS LOCATOR**
For the 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 UDF instead of the actual value. This saves greatly in the number of bytes passed to the UDF, 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 UDF. Use of LOB locators in UDFs are described in *Application Development Guide*.

Here is an example which illustrates the use of the AS LOCATOR clause in parameter definitions:

```
CREATE FUNCTION foo (CLOB(10M) AS LOCATOR, IMAGE AS LOCATOR)
            ...
```

which assumes that IMAGE is a distinct type based on one of the LOB types.

Note also that for argument promotion purposes, the AS LOCATOR clause has no effect. In the example the types are considered to be CLOB and IMAGE respectively, which would mean that a CHAR or VARCHAR argument could be passed to the function as the first argument. Likewise, the AS LOCATOR has no effect on the function signature, which is used in matching the function (a) when referenced in DML, by a process called "function resolution", and (b) when referenced in a DDL statement such as COMMENT ON or DROP. In fact the clause may or may not be used in COMMENT ON or DROP with no significance.

An error (SQLSTATE 42601) is raised if AS LOCATOR is specified for a type other than a LOB or a distinct type based on a LOB.

If the function is FENCED, 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 above under *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 (for the definition of castable, see "Casting Between Data Types" on page 88).

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 paramaterized 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).

A distinct type is not valid as the type specified in *data-type4* (SQLSTATE 42815).

The cast operation is also subject to run-time 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. Use of LOB locators in UDFs are described in *Application Development Guide*.

**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 (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 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, SQLyymmddhhmmsshhn.

**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.

## CREATE FUNCTION (External Scalar)

**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 characters. 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 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 performed. 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 (SQLSTATE 42724) is raised.

```
►►─'─┬─library_id──────┬─┬──────────┬─'─────────────────►◄
     └─absolute_path_id─┘ └─!─func_id─┘
```

>    Extraneous blanks are not permitted within the single quotes.
>
>    *library_id*
>    >    Identifies the library name containing the function. The database manager will look for the library in the .../sqllib/function directory (UNIX-based systems), or ...\\*instance_name*\function directory (OS/2, Windows 95 and Windows NT as specified by the DB2INSTPROF registry variable), where the database manager will locate the controlling sqllib directory which is being used to run the database manager. For example, the controlling sqllib directory in UNIX-based systems is /u/$DB2INSTANCE/sqllib.
>    >
>    >    'myfunc' were the *library_id* in a UNIX-based system it would cause the database manager to look for the function in library /u/production/sqllib/function/myfunc, provided the database manager is being run from /u/production.
>    >
>    >    For OS/2, Windows 95 and Windows NT, the database manager will look in the LIBPATH or PATH if the *library_id* is not located in the function directory.
>    >
>    >    In OS/2 the library_id should not contain more than 8 characters.
>
>    *absolute_path_id*
>    >    Identifies the full path name of the file containing the function.

## CREATE FUNCTION (External Scalar)

> In a UNIX-based system, for example, '/u/jchui/mylib/myfunc' would cause the database manager to look in /u/jchui/mylib for the myfunc shared library.
>
> In OS/2, Windows 95 and Windows NT, 'd:\mylib\myfunc' would cause the database manager to load dynamic link library, myfunc.dll file, from the d:\mylib directory.
>
> In OS/2 the last part of this specification (i.e. the name of the dll), should not contain more than 8 characters.

*! 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. If *! func_id* is omitted, the database manager will use the default entry point established when the library was linked.
>
> In a UNIX-based system, for example, '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.
>
> In OS/2, Windows 95 and Windows NT, 'mymod!func8' would direct the database manager to load the mymod.dll file and call the func8() function in the dynamic link library (DLL).

If the string is not properly formed, an error (SQLSTATE 42878) is raised.

The body of every external function should be in a directory which is mounted and available on every partition of the database.

- 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 performed. 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 (SQLSTATE 42724) is raised.

```
►►─'─┬──────────────┬─class_id─┬───┬─method_id─'──────────────►◄
     └─jar_id :─────┘          ├─.─┤
                               └─!─┘
```

Extraneous blanks are not permitted within the single quotes.

## CREATE FUNCTION (External Scalar)

*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 Java virtual machine will look in directory '.../myPacks/UserFuncs/' for the classes. In OS/2 and Windows 95 and Windows NT, the Java virtual machine will look in directory '...\myPacks\UserFuncs\'.

*method_id*
> Identifies the method name of the Java object to be invoked.

- 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 performed. 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 (SQLSTATE 42724) is raised.

```
►►─' ──┬─progid─┬──!─method_id─' ─────────────────────────────►◄
        └─clsid──┘
```

  Extraneous blanks are not permitted within the single quotes.

*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.

**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 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.

> **OLE**    This means the database manager will call 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 only supported for user-defined functions stored in DB2 for Windows 95 and Windows NT.

**PARAMETER STYLE**
> This clause is used to specify the conventions used for passing parameters to and returning the value from functions.

> **DB2SQL**
> > 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 or methods exposed by OLE automation objects. This must be specified when LANGUAGE C or LANGUAGE OLE is used.

> **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. PARAMETER STYLE JAVA functions do not support the
FINAL CALL, SCRATCHPAD or DBINFO clauses.

Refer to *Application Development Guide* for details on passing parameters.

**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 (NOT FENCED), or not (FENCED).

If a function is registered as FENCED, the database manager insulates its
internal resources (e.g. 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.

**Warning:** Use of NOT FENCED for functions not adequately checked out
can compromise the integrity of DB2. DB2 takes 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.

Most user-defined functions should be able to run either as FENCED or
NOT FENCED. Only FENCED can be specified for a function with
LANGUAGE OLE (SQLSTATE 42613).

If the function is FENCED, the AS LOCATOR clause cannot be specified
(SQLSTATE 42613).

To change from FENCED to NOT FENCED, the function must be
re-registered (by first dropping it and then re-creating it). Either SYSADM
authority, DBADM authority or a special authority
(CREATE_NOT_FENCED) is required to register a user-defined function
as NOT FENCED.

**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 execution time
any one of the function's arguments is null, 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 compatability. Similarly, NOT
NULL CALL may be used as a synonym for RETURNS NULL ON NULL
INPUT.

**NO SQL**
This mandatory clauses indicates that the function cannot issue any SQL
statements. If it does, an error (SQLSTATE 38502) is raised at run time.

**NO EXTERNAL ACTION** or **EXTERNAL ACTION**
This optional clause specifies whether or not the function takes some
action that changes the state of an object not managed by the database
manager. Optimizations that assume functions 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.

**NO SCRATCHPAD** or **SCRATCHPAD**
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. This scratchpad has the following characteristics:

- It is 100 bytes in size.
- It is initialized to all X'00''s.

## CREATE FUNCTION (External Scalar)

- 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 above. If the function is executed in multiple partitions, a scratchpad would be assigned in each partition where the function is processed, for each reference to the function in the SQL statement.

- 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 SCRATCHPAD is specified, then on each invocation of the user-defined function an additional argument is passed to the external function which addresses the scratchpad.

If NO SCRATCHPAD is specified then no scratchpad is allocated or passed to the external function.

SCRATCHPAD is not supported for PARAMETER STYLE JAVA functions.

**NO FINAL CALL** or **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:

CREATE FUNCTION (External Scalar)

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-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 function, and no final call is made.

A description of the scalar UDF processing of these calls when errors occur is included in the *Application Development Guide*.

FINAL CALL is not supported for PARAMETER STYLE JAVA functions.

**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 DB2 will accept this specification. 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.

**CREATE FUNCTION (External Scalar)**

- Is there some external action performed by the UDF which should happen only on one 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 function should be in a directory that is mounted and available on every partition of the database.

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 DB2 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 LANGUAGE OLE (SQLSTATE 42613) or 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 run-time 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-hand 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.

Please see the *Application Development Guide* for detailed information on the structure and how it is passed to the user-defined function.

**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" on page 87). 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.

- For information on writing, compiling, and linking an external user-defined function, see the *Application Development Guide*.

- 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.

**Examples**

*Example 1:* Pellow is registering the CENTRE function in his PELLOW schema. Let those keywords that will default default, and let the system provide a function specific name:

## CREATE FUNCTION (External Scalar)

```
CREATE FUNCTION CENTRE (INT,FLOAT)
    RETURNS FLOAT
    EXTERNAL NAME 'mod!middle'
    LANGUAGE C
    PARAMETER STYLE DB2SQL
    DETERMINISTIC
    NO SQL
    NO EXTERNAL ACTION
```

*Example 2:* Now, McBride (who has DBADM authority) is registering another CENTRE 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.CENTRE (FLOAT, FLOAT, FLOAT)
    RETURNS DECIMAL(8,4) CAST FROM FLOAT
    SPECIFIC FOCUS92
    EXTERNAL NAME 'effects!focalpt'
    LANGUAGE C   PARAMETER STYLE DB2SQL
    DETERMINISTIC   FENCED   NOT NULL CALL   NO SQL   NO EXTERNAL ACTION
    SCRATCHPAD   NO FINAL CALL
    DISALLOW PARALLEL
```

*Example 3:* The following 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 the null checking), if the CREATE FUNCTION statement had used NOT NULL CALL. Further examples of user-defined function programs can be found in the *Application Development Guide*. The CREATE FUNCTION statement:

```
CREATE FUNCTION ntest1 (SMALLINT)
    RETURNS SMALLINT
    EXTERNAL NAME 'ntest1!nudft1'
    LANGUAGE C   PARAMETER STYLE DB2SQL
    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 */
```

*Example 4:* The following 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
```

## CREATE FUNCTION (External Table)

This statement is used to register a user-defined external table function with an application server.

A *table function* may 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. 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 at least one of the following:
- SYSADM or DBADM authority
- 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.

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:
- CREATE_NOT_FENCED authority on the database
- SYSADM or DBADM authority.

To create a fenced function, no additional authorities or privileges are required.

If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

### Syntax

```
►►──CREATE FUNCTION──function-name──(─────────────────────────────────)────────►
                                    │      ┌─,──────────────────┐      │
                                    └──▼──data-type1──────────┬──┘
                                                  └─AS LOCATOR─┘
```

```
         ┌─,──────────────────────────────┐
►►─●─RETURNS TABLE─(─▼─column-name──data-type2─┬──────────────┬─)─●──────►
                                               └─AS LOCATOR───┘

►─┬────────────────────────┬─●─EXTERNAL─┬─────────────────────┬─●────────►
  └─SPECIFIC──specific-name─┘           └─NAME─┬─'string'────┬─┘
                                               └─identifier──┘

                          (1)
►─LANGUAGE─┬─C────┬──●─PARAMETER STYLE─┬─DB2SQL──────┬─●──────────────────►
           ├─JAVA─┤                    └─DB2GENERAL──┘
           └─OLE──┘

  ┌─NOT DETERMINISTIC──┐     ┌─FENCED──────┐
►─┼────────────────────┼─●───┼─────────────┼─●───────────────────────────►
  │          (2)       │     └─NOT FENCED──┘
  └─DETERMINISTIC──────┘

  ┌─RETURNS NULL ON NULL INPUT─┐  (3)          ┌─EXTERNAL ACTION─────┐
►─┼────────────────────────────┼───●─NO SQL─●──┼─────────────────────┼─●─►
  └─CALLED ON NULL INPUT───────┘               └─NO EXTERNAL ACTION──┘

  ┌─NO SCRATCHPAD─┐     ┌─NO FINAL CALL─┐
►─┼───────────────┼─●───┼───────────────┼─●─DISALLOW PARALLEL─●───────────►
  └─SCRATCHPAD────┘     └─FINAL CALL────┘

  ┌─NO DBINFO─┐
►─┼───────────┼─●─┬──────────────────────┬─●─────────────────────────────►◄
  └─DBINFO────┘   └─CARDINALITY──integer──┘
```
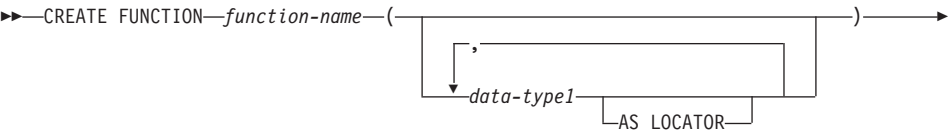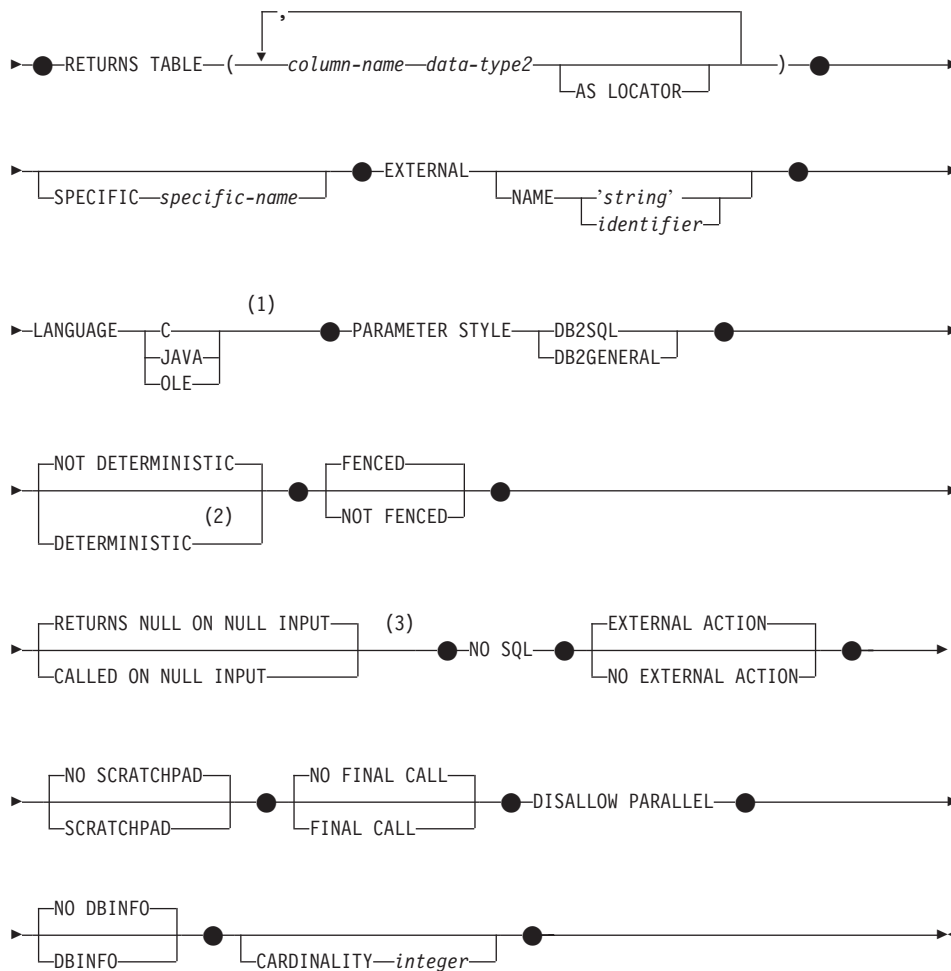
**Notes:**

**1**    Refer to "CREATE FUNCTION (OLE DB External Table)" on page 585 for creating LANGUAGE OLE DB external table functions.

**2**    NOT VARIANT may be specified in place of DETERMINISTIC and VARIANT may be specified in place of NOT DETERMINISTIC.

**3**    NULL CALL may be specified in place of CALLED ON NULL INPUT and NOT NULL CALL may be specified in place of RETURNS NULL ON NULL INPUT.

## CREATE FUNCTION (External Table)

### 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 (with a maximum length of 18). 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". Otherwise, an error (SQLSTATE 42939) is raised.

A number of names used as keywords in predicates are reserved for system use, and may not 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 as described in "Basic Predicate" on page 167. Failure to observe this rule will lead to an error (SQLSTATE 42939).

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.

*(data-type1,...)*

Identifies the number of input parameters of the function, and specifies the data type 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. If this limit is exceeded, an error (SQLSTATE 54023) is raised.

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). 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 raises an SQL error (SQLSTATE 42723).

For example, given the statements:

```
CREATE FUNCTION PART (INT, CHAR(15)) ...
CREATE FUNCTION PART (INTEGER, CHAR(40)) ...

CREATE FUNCTION ANGLE (DECIMAL(12,2)) ...
CREATE FUNCTION ANGLE (DEC(10,7)) ...
```

the second and fourth statements would fail because they are considered to be a duplicate functions.

*data-type1*
>    Specifies the data type of the parameter.
>
>    - SQL data type specifications and abbreviations which may be specified in the *data-type* definition of a CREATE TABLE statement and have a correspondence in the language that is being used to write the function may be specified. See the language-specific sections of the *Application Development Guide* for details on the mapping between the SQL data types and host language data types with respect to user-defined functions.
>
>    - DECIMAL (and NUMERIC) are invalid with LANGUAGE C and OLE (SQLSTATE 42815). For alternatives to using DECIMAL refer to *Application Development Guide*.
>
>    - REF(*type-name*) cannot be specified as the data type of a parameter (SQLSTATE 42997).

**AS LOCATOR**
>    For the 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 UDF instead of the actual value. This saves greatly in the number of bytes passed to the UDF, 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 UDF. Use of LOB locators in UDFs are described in *Application Development Guide*.
>
>    Here is an example which illustrates the use of the AS LOCATOR clause in parameter definitions:
>
>    ```
>    CREATE FUNCTION foo ( CLOB(10M) AS LOCATOR, IMAGE AS LOCATOR)
>             ...
>    ```

which assumes that IMAGE is a distinct type based on one of the LOB types.

Note also that for argument promotion purposes, the AS LOCATOR clause has no effect. In the example the types are considered to be CLOB and IMAGE respectively, which would mean that a CHAR or VARCHAR argument could be passed to the function as the first argument. Likewise, the AS LOCATOR has no effect on the function signature, which is used in matching the function (a) when referenced in DML, by a process called "function resolution", and (b) when referenced in a DDL statement such as COMMENT ON or DROP. In fact the clause may or may not be used in COMMENT ON or DROP with no significance.

An error (SQLSTATE 42601) is raised if AS LOCATOR is specified for a type other than a LOB or a distinct type based on a LOB.

If the function is FENCED, the AS LOCATOR clause cannot be specified (SQLSTATE 42613).

**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 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.

**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 on page 554.

**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 (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 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, SQLyymmddhhmmsshhn.

**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 which implements the function being defined.

The 'string' option is a string constant with a maximum of 254 characters. 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 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 performed. 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.

    ```
    ►►—'——┬─library_id─────┬──┬───────────┬—'——————————►◄
           └─absolute_path_id─┘  └─!—func_id─┘
    ```

    Extraneous blanks are not permitted within the single quotes.

    *library_id*
        Identifies the library name containing the function. The database manager will look for the library in the .../sqllib/function directory (UNIX-based systems), or ...\\*instance_name*\\function directory (OS/2, Windows 95 and Windows NT as specified by the DB2INSTPROF registry

variable), where the database manager will locate the controlling sqllib directory which is being used to run the database manager. For example, the controlling sqllib directory in UNIX-based systems is /u/$DB2INSTANCE/sqllib.

If 'myfunc' were the *library_id* in a UNIX-based system it would cause the database manager to look for the function in library /u/production/sqllib/function/myfunc, provided the database manager is being run from /u/production.

For OS/2, Windows 95 and Windows NT, the database manager will look in the LIBPATH or PATH if the *library_id* is not located in the function directory.

In OS/2 the library_id should not contain more than 8 characters.

*absolute_path_id*
Identifies the full path name of the function.

In a UNIX-based system, for example, '/u/jchui/mylib/myfunc' would cause the database manager to look in /u/jchui/mylib for the myfunc function.

In OS/2, Windows 95 and Windows NT 'd:\mylib\myfunc' would cause the database manager to load the myfunc.dll file from the d:\mylib directory.

In OS/2 the last part of this specification (i.e. the name of the dll), should not contain more than 8 characters.

*! 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. If *! func_id* is omitted, the database manager will use the default entry point established when the library was linked.

In a UNIX-based system, for example, '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.

In OS/2, Windows 95 and Windows NT, 'mymod!func8' would direct the database manager to load the mymod.dll file and call the func8() function in the dynamic link library (DLL).

If the string is not properly formed, an error (SQLSTATE 42878) is raised.

In any case, the body of every external function should be in a directory which is mounted and available on every partition of the database.

- 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 performed. 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.



Extraneous blanks are not permitted within the single quotes.

*jar_name*
    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 Java virtual machine will look in directory '.../myPacks/UserFuncs/' for the classes. In OS/2 and Windows 95 and Windows NT, the Java virtual machine will look in directory '...\myPacks\UserFuncs\'.

*method_id*
    Identifies the method name of the Java object to be invoked.

- 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 performed. 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 (SQLSTATE 42724) is raised.

## CREATE FUNCTION (External Table)

```
►►──'──┬─progid─┬──!──method_id──'──────────────────────►◄
        └─clsid──┘
```

Extraneous blanks are not permitted within the single quotes.

*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.

**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.

> **OLE**    This means the database manager will call the user-defined function as if it were a method exposed by an OLE automation

578

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 only supported for user-defined functions stored in DB2 for Windows 95 and Windows NT.

Refer to "CREATE FUNCTION (OLE DB External Table)" on page 585 for creating LANGUAGE OLEDB external table functions.

**PARAMETER STYLE**
This clause is used to specify the conventions used for passing parameters to and returning the value from functions.

**DB2SQL**
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. This must be specified when LANGUAGE C or LANGUAGE OLE is used.

**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.

The value DB2GENRL may be used as a synonym for DB2GENERAL.

**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 NOT DETERMINISTIC table function would be a function that retrieves data from a data source such as a file.

**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 (NOT FENCED), or not (FENCED).

If a function is registered as FENCED, the database manager insulates its internal resources (e.g. 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.

## CREATE FUNCTION (External Table)

> **Warning:** Use of NOT FENCED for functions not adequately checked out can compromise the integrity of DB2. DB2 takes 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.

Most user-defined functions should be able to run either as FENCED or NOT FENCED. Only FENCED can be specified for a function with LANGUAGE OLE (SQLSTATE 42613).

To change from FENCED to NOT FENCED, the function must be re-registered (by first dropping it and then re-creating it). Either SYSADM authority, DBADM authority or a special authority (CREATE_NOT_FENCED) is required to register a user-defined function as NOT FENCED.

If the function is FENCED, the AS LOCATOR clause 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 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, i.e. 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 compatability. Similarly, NOT NULL CALL may be used as a synonym for RETURNS NULL ON NULL INPUT.

**NO SQL**
This mandatory clauses indicates that the function cannot issue any SQL statements. If it does, an error (SQLSTATE 38502) is raised at run time.

**NO EXTERNAL ACTION** or **EXTERNAL ACTION**
This optional clause specifies whether or not the function takes some action that changes the state of an object not managed by the database manager. Optimizations that assume functions 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.

**NO SCRATCHPAD** or **SCRATCHPAD**

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. This scratchpad has the following characteristics:

- It is 100 bytes in size.
- 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 by DB2, 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 DB2 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 outlined above, 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 SCRATCHPAD is specified, then on each invocation of the user-defined function an additional argument is passed to the external function which addresses the scratchpad.

## CREATE FUNCTION (External Table)

> If NO SCRATCHPAD is specified then no scratchpad is allocated or passed to the external function.

**NO FINAL CALL or 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-initalized. If NO FINAL CALL is specified, then DB2 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. See *Application Development Guide* for more information about this argument and its values.

> A description of the table UDF processing of these calls when errors occur is included in the *Application Development Guide*.

**DISALLOW PARALLEL**
> This clause specifies that, for a single reference to the function, the invocation of the function can not be parallelized. Table functions are always run on a single partition.

**NO DBINFO** or **DBINFO**
> This optional clause specifies whether certain specific information known by DB2 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 LANGUAGE OLE (SQLSTATE 42613).

> 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 run-time authorization ID, regardless of the nested UDFs in between this UDF and the application.
> - Code page - identifies 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/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 - an array of the numbers of the table function result columns actually needed for the particular

statement referencing the function. Only provided for table functions, it enables the UDF to optimize by only returning the required column values instead of all column values.

Please see the *Application Development Guide* for detailed information on the structure and how it is passed to the UDF.

**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, DB2 will assume a finite value as a default- the same value assumed for tables for which the RUNSTATS utility has not gathered statistics.

Warning: if a function does in fact have infinite cardinality, i.e. it returns a row every time it is called to do so, never returning the "end-of-table" condition, then queries which require the "end-of-table" condition to correctly function will be infinite, and will have to be interrupted. Examples of such queries are those involving GROUP BY and ORDER BY. The user is advised to not write such UDFs.

## 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 (see "Promotion of Data Types" on page 87). 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.

- For information on writing, compiling, and linking an external user-defined function, see the *Application Development Guide*.

- 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.

## CREATE FUNCTION (External Table)

### Examples

*Example 1:* The following 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 DB2 optimizer.

```
CREATE FUNCTION DOCMATCH (VARCHAR(30), VARCHAR(255))
    RETURNS TABLE (DOC_ID CHAR(16))
    EXTERNAL NAME '/common/docfuncs/rajiv/udfmatch'
    LANGUAGE C
    PARAMETER STYLE DB2SQL
    NO SQL
    DETERMINISTIC
    NO EXTERNAL ACTION
    NOT FENCED
    SCRATCHPAD
    FINAL CALL
    DISALLOW PARALLEL
    CARDINALITY 20
```

*Example 2:* The following registers an OLE table function that is used to retrieve message header information and the partial message text of messages in Microsoft Exchange. For an example of the code that implements this table function, see the *Application Development Guide*.

```
CREATE FUNCTION MAIL()
    RETURNS TABLE (TIMERECIEVED DATE,
                  SUBJECT VARCHAR(15),
                  SIZE INTEGER,
                  TEXT VARCHAR(30))
    EXTERNAL NAME 'tfmail.header!list'
    LANGUAGE OLE
    PARAMETER STYLE DB2SQL
    NOT DETERMINISTIC
    FENCED
    CALLED ON NULL INPUT
    SCRATCHPAD
    FINAL CALL
    NO SQL
    EXTERNAL ACTION
    DISALLOW PARALLEL
```

## CREATE FUNCTION (OLE DB External Table)

This statement is used to register a user-defined OLE DB external table function to access data from an OLE DB provider.

A *table function* may 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. 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 at least one of the following:
- SYSADM or DBADM authority
- 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.

If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

### Syntax

```
►►──CREATE FUNCTION──function-name──(────────────────)──●──────────────►
                                        └─data-type1─┘


                           ┌─────────,────────────┐
►──RETURNS TABLE──(──▼──column-name──data-type2──┴──)──●───────────────►


►──────────────────────────────●──EXTERNAL──NAME──'string'──●──────────►
   └─SPECIFIC──specific-name─┘


                              ┌─NOT DETERMINISTIC─┐
►──LANGUAGE──OLEDB──●──────────┼───────────────────┼──●─────────────────►
                              │              (1)   │
                              └─DETERMINISTIC──────┘
```

## CREATE FUNCTION (OLE DB External Table)

```
      ┌─RETURNS NULL ON NULL INPUT─┐  (2)  ┌─NO EXTERNAL ACTION─┐
▶──────┤                            ├───●───┤                    ├───●──────▶
       └─CALLED ON NULL INPUT───────┘       └─EXTERNAL ACTION────┘

▶──────┬──────────────────────────┬───●──────────────────────────────────◀
       └─CARDINALITY──integer──────┘
```

### Notes:

**1**  NOT VARIANT may be specified in place of DETERMINISTIC and
VARIANT may be specified in place of NOT DETERMINISTIC.

**2**  NULL CALL may be specified in place of CALLED ON NULL INPUT
and NOT NULL CALL may be specified in place of RETURNS NULL
ON NULL INPUT.

## 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 (with a maximum length of 18). 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".
Otherwise, an error (SQLSTATE 42939) is raised.

A number of names used as keywords in predicates are reserved for
system use, and may not 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
as described in "Basic Predicate" on page 167. Failure to observe this rule
will lead to an error (SQLSTATE 42939).

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

## CREATE FUNCTION (OLE DB External Table)

prohibition against it, an external user-defined table function should not be given the same name as a built-in function.

*data-type1*

Identifies the input parameter of the function, and specifies the data type of the parameter. If no input parameter is specified, then data is retrieved from the external source possibly subsetted through query optimization. The input parameter can be any character or graphic string data type and it 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. Length is not considered in this type comparison. Therefore CHAR(8) and CHAR(35) are considered to be the same type. A duplicate signature raises an SQL error (SQLSTATE 42723).

**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 (see language-specific sections of *Application Development Guide* for details on the mapping between the SQL data types and OLE DB 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 (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 function instance that exists at the application server; otherwise an error (SQLSTATE 42710) is raised.

## CREATE FUNCTION (OLE DB External Table)

> 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, SQLyymmddhhmmsshhn.

**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 characters.
>
> The string specified is used to establish a connection and session with a 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 performed. See OLE DB Table Functions in *Application Development Guide* for more details.
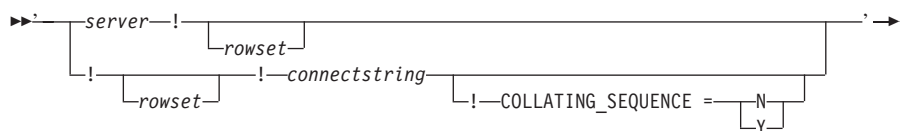


> *server*
>> Identifies the local name of a data source as defined by "CREATE SERVER" on page 635.
>
> *rowset*
>> Identifies the rowset (table) exposed by the OLE DB provider. Fully qualified table names must be provided for OLE DB providers that support catalog or schema names.
>
> *connectstring*
>> 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. Refer to the language-specific sections of *Application Development Guide* for details.
>
> **COLLATING_SEQUENCE**
>> Specifies whether the data source uses the same collating sequence as DB2 Universal Database. See "CREATE SERVER" on page 635 for details. Valid values are as follows:

CREATE FUNCTION (OLE DB External Table)

Y = Same collating sequence

N = Different collating sequence

If COLLATING_SEQUENCE is not specified, then the data source is assumed to have a different collating sequence from DB2 Universal Database.

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.

**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, i.e. 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.

**CREATE FUNCTION (OLE DB External Table)**

The value NULL CALL may be used as a synonym for CALLED ON NULL INPUT for backwards and family compatability. Similarly, NOT NULL CALL may be used as a synonym for RETURNS NULL ON NULL INPUT.

**NO EXTERNAL ACTION** or **EXTERNAL ACTION**
This optional clause specifies whether or not the function takes some action that changes the state of an object not managed by the database manager. Optimizations that assume functions with no external impacts are prevented by specifying EXTERNAL ACTION. For example: sending a message, ringing a bell, or writing a record to a file.

**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, DB2 will assume a finite value as a default- the same value assumed for tables for which the RUNSTATS utility has not gathered statistics.

Warning: if a function does in fact have infinite cardinality, i.e. it returns a row every time it is called to do so, never returning the "end-of-table" condition, then queries which require the "end-of-table" condition to correctly function will be infinite, and will have to be interrupted. Examples of such queries are those involving GROUP BY and ORDER BY. The user is advised to not write such UDFs.

**Notes**

- FENCED, FINAL CALL, SCRATCHPAD, PARAMETER STYLE DB2SQL, DISALLOW PARALLEL, NO DBINFO, and NO SQL are implicit in the statement and can be specified. Refer to "CREATE FUNCTION (External Table)" on page 570 for specific descriptions.
- 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" on page 87). 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:

  – 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.

590

- For information on creating a user-defined OLE DB external table function, see the *Application Development Guide*.
- 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.

## Examples

*Example 1:* The following registers 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';
```

*Example 2:* The following registers 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 userid in the connection string. Refer to "CREATE SERVER" on page 635, "CREATE WRAPPER" on page 729 and "CREATE USER MAPPING" on page 712 for details on the statements.

```
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),
```

## CREATE FUNCTION (OLE DB External Table)

```
                    zip_code INTEGER)
  LANGUAGE OLEDB
  EXTERNAL NAME 'spirit!demo.customer';
```

*Example 3:*  The following registers 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 a SQL comand text to retrieve the top 3 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, ' || '
                    stores.stor_name as name, ' || '
                    sum(sales. qty) as sales  ' || '
                  from sales, stores ' || '
                  where sales.stor_id = stores.stor_id ' || '
                  group by sales.stor_id, stores.stor_name' || '
                  order by sum(sales.qty) desc')) as f;
```

## CREATE FUNCTION (Source or Template)

This statement is used to:

- Register a user-defined function, based on another existing scalar or column function, with an application server.
- Register 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. Refer to "Function Mappings, Function Templates, and Function Mapping Options" on page 46 for more information.

### 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 at least one of the following:
- SYSADM or DBADM authority
- 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.

If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

No authority is required on a function referenced in the SOURCE clause.

### Syntax

## CREATE FUNCTION (Source or Template)

```
►──RETURNS──data-type2──●──────────────────────────────────────●──────────►
                          └─SPECIFIC──specific-name─┘


►──┬─SOURCE──┬─function-name─────────────────────────────┬──●──────────────►◄
   │         ├─SPECIFIC──specific-name───────────────────┤
   │         └─function-name──(──────────────────────)─┘
   │                            │    ┌──,──┐            │
   │                            └──▼─data-type─┘
   └─AS TEMPLATE──────────────────────────────────────────
```
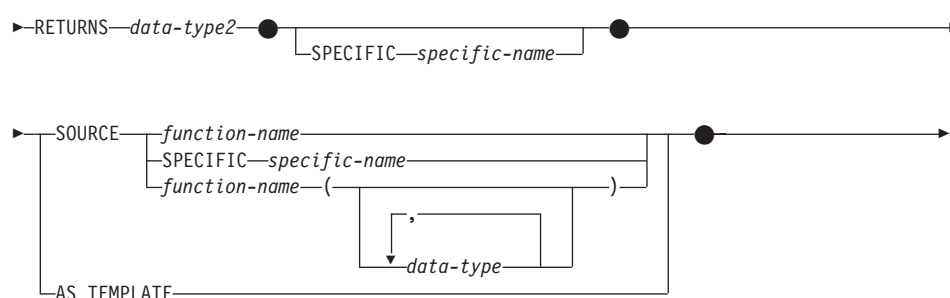
### Description

*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 (with a maximum length of 18). 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". Otherwise, an error (SQLSTATE 42939) is raised.
>
>    A number of names used as keywords in predicates are reserved for system use, and may not 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 as described in "Basic Predicate" on page 167. Failure to observe this rule will lead to an error (SQLSTATE 42939).
>
>    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.

## CREATE FUNCTION (Source or Template)

*(data-type,...)*

Identifies the number of input parameters of the function or function template, and specifies the data type 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. If this limit is exceeded, an error (SQLSTATE 54023) is raised.

It is possible to register a function or function template 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 or function templates within a schema are permitted to have exactly the same type for all corresponding parameters. (This restriction applies also to a function and function template within a schema that have the same name.) 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). 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 raises an SQL error (SQLSTATE 42723).

For example, given the statements:

```
CREATE FUNCTION PART (INT, CHAR(15)) ...
CREATE FUNCTION PART (INTEGER, CHAR(40)) ...

CREATE FUNCTION ANGLE (DECIMAL(12,2)) ...
CREATE FUNCTION ANGLE (DEC(10,7)) ...
```

the second and fourth statements would fail because they are considered to be a duplicate functions.

*data-type1*

Specifies the data type of the parameter.

With a sourced scalar function any valid SQL data type may be used provided it is castable to the type of the corresponding parameter of the function identified in the SOURCE clause (for the definition of castable, see "Casting Between Data Types" on page 88). A REF(*type-name*) data type cannot be specified as the data type of a parameter (SQLSTATE 42997).

Since the function is sourced, it is not necessary (but still permitted) to specify length, precision, or scale for the parameterized data types. Instead, empty parentheses may be used (for example CHAR() may be used). 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.

**RETURNS**

This mandatory clause identifies the output of the function or function template.

*data-type2*

Specifies the data type of the output.

Any valid SQL data type is valid, as is a distinct type, provided it is castable from the result type of the source function (for the definition of castable, see "Casting Between Data Types" on page 88).

The parameter of a parameterized type need not be specified, as above for parameters of a sourced function. Instead, empty parentheses may be used, for example, VARCHAR().

Also see page 598 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.

**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 (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 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, SQLyymmddhhmmsshhn.

**SOURCE**

Specifies that the function being created is to be implemented by another function (the source function) already known to the database manager. The source function can be either a built-in function[67]or a previously created user-defined scalar function (including a sourced function).

---

67. With the exception of COALESCE, NODENUMBER, NULLIF, PARTITION, TYPE_ID, TYPE_NAME, TYPE_SCHEMA and VALUE.

## CREATE FUNCTION (Source or Template)

The SOURCE clause may be specified only for scalar or column functions; it may not be specified for table functions.

The SOURCE clause provides the identity of the other function.

*function-name*
> Identifies the particular function that is to be used as the source and is valid only if there is exactly one specific function in the schema with this *function-name*. This syntax variant is not valid for a source function that is a built-in function.
>
> If an unqualified name is provided, then the authorization ID's current SQL path (the value of the CURRENT PATH special register) is used to locate the function. The first schema in the function path that has a function with this name is selected.
>
> If no function by this name exists in the named schema or if the name is not qualified and there is no function with this name in the function path, an error (SQLSTATE 42704) is raised. If there is more than one specific instance of the function in the named or located schema, an error (SQLSTATE 42725) is raised.

**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, then the authorization ID's current SQL path is used to locate the function. The first schema in the function path that has a function with this specific name 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 raised.

*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 (as described in "Function Resolution" on page 139) 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.

## CREATE FUNCTION (Source or Template)

*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) since 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 exact intended function will be used. Also note 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 since 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 raised.

**AS TEMPLATE**
> Indicates that this statement will be used to create a function template, not a function with executable code.

## Rules

- For convenience, in this section we will call the function being created CF and the function identified in the SOURCE clause 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 raises an error.

**CREATE FUNCTION (Source or Template)**

- – 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
  SCRATCHPAD / NO SCRATCHPAD
  FINAL CALL / NO FINAL CALL
  RETURNS TABLE (...)
  CARDINALITY ...
  ALLOW PARALLEL / DISALLOW PARALLEL
  DBINFO / NO DBINFO

  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 raised.

- 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 below 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. For the definition of castable, see "Casting Between Data Types" on page 88. If any parameter is not the same type or castable, an error (SQLSTATE 42879) is raised.

  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

**CREATE FUNCTION (Source or Template)**

> 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 raised.

  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.

### 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" on page 87). 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, DB2 (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, DB2 invokes the data source function that it maps to, provided that an access plan for invoking this function exists. Refer to the *Application Development Guide* for more information about controlling the overhead of invoking functions in a federated system.

## Examples

*Example 1:* Some time after the creation of Pellow's original CENTRE 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 MYCENTRE (INTEGER, INTEGER)
    RETURNS FLOAT
    SOURCE PELLOW.CENTRE (INTEGER, FLOAT)
```

*Example 2:* You have created a distinct type HATSIZE which is based on the built-in INTEGER data type, and now would find it 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; for the user's code for the mapping, refer to "Examples" on page 606.

```
CREATE FUNCTION NOVA.STATS (DOUBLE, DOUBLE)
    RETURNS DOUBLE
    AS TEMPLATE
```

*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

## CREATE FUNCTION (Source or Template)

template. The user uses the following code to register this template with the federated server; for the user's code for the mapping, refer to "Examples" on page 606.

```
CREATE FUNCTION BONUS ()
     RETURNS DECIMAL (8,2)
     AS TEMPLATE
```

## CREATE FUNCTION MAPPING

The CREATE FUNCTION MAPPING statement is used to:

- Create a mapping between a federated database function or function template and a data source function. The mapping can associate the federated database function or template with a function at either (1) a specified data source or (2) a range of data sources; for example, all data sources of a particular type and version.
- Disable a default mapping between a federated database function and a data source 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. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

### Authorization

The authorization ID of the statement must have SYSADM or DBADM authority.

### Syntax

```
►►──CREATE FUNCTION MAPPING──────────────────────FOR──────────────────────►
                            └─function-mapping-name─┘


                              ┌─,──────────┐
►──┬─function-name──(─▼──────────────┬──)─────┬─────────────────────────────►
   │                     └─data-type─┘        │
   └─SPECIFIC──specific-name──────────────────┘


►──┬─SERVER──server-name──────────────────────────────────────────────────►
   └─SERVER TYPE──server-type─┬─────────────────────────────────────────┬──
                              └─VERSION──┤ server-version ├──┬─────────────────────────┬─┘
                                                            └─WRAPPER──wrapper-name─┘


►──┤ function-options ├──┬───────────────┬──────────────────────────────►◄
                         └─WITH INFIX─┘
```
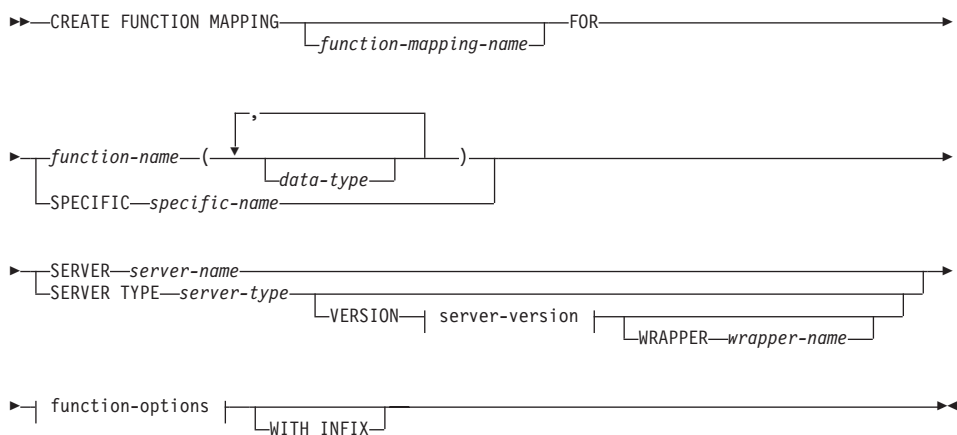
## CREATE FUNCTION MAPPING

**server-version:**
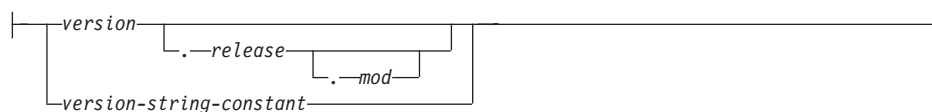


**function-options:**
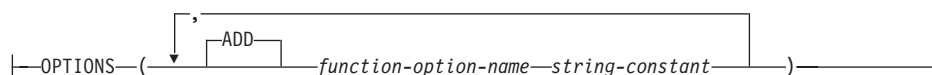


## Description

*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*
    Is the qualified or unqualified name of the function or function template to map from.

*data-type*
    For a function or function template that has any input parameters, *data-type* specifies the data type of such a parameter. The *data type* cannot be LONG VARCHAR, LONG VARGRAPHIC, DATALINK, a large object (LOB) type, or a user-defined type.

**SPECIFIC** *specific-name*
    Identifies the function or function template to map from. Specify *specific-name* if the function or function template does not have a unique *function-name* in the federated database.

**SERVER** *server-name*
    Names the data source that contains the function that is being mapped to.

**TYPE** *server-type*
    Identifies the type of data source that contains the function that is being mapped to.

**VERSION**
    Identifies the version of the data source denoted by *server-type*.

    *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*
> 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**
> Indicates what function mapping options are to be enabled. Refer to
> "Function Mapping Options" on page 1048 for descriptions of
> *function-option-name*s and their settings.

> **ADD**
> > Enables one or more function mapping options.

> *function-option-name*
> > Names a function mapping option that applies either to the function
> > mapping or to the data source function included in the mapping.

> *string-constant*
> > Specifies the setting for *function-option-name* as a character string
> > constant.

**WITH INFIX**
> Specifies that the data source function be generated in infix format.

## 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 that are defined for the
    data source function.
- If a distributed request references a DB2 function that maps to a data
  source function, the optimizer develops strategies for invoking either
  function when the request is processed. The DB2 function is invoked if

doing so requires less overhead than invoking the data source function. Otherwise, if invoking the DB2 function requires more overhead, then the data source function is invoked.

- If a distributed request references a DB2 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 DB2 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* with qualifier SYSIBM and function name (such as LENGTH).

- A CREATE FUNCTION MAPPING statement 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 a SELECT statement that references 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 a SELECT statement that references a nickname for a table or view within one of these data sources.

## 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)
   TYPE ORACLE
   OPTIONS ( DISABLE 'Y' )
```

*Example 4:* Map the local function UCASE(CHAR) to a UDF that's 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 INDEX

The CREATE INDEX statement is used to create:

- An index on a DB2 table
- 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. 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 at least one of the following:

- SYSADM or DBADM authority.
- One of:
  - CONTROL privilege on the table
  - INDEX privilege on the table

  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.
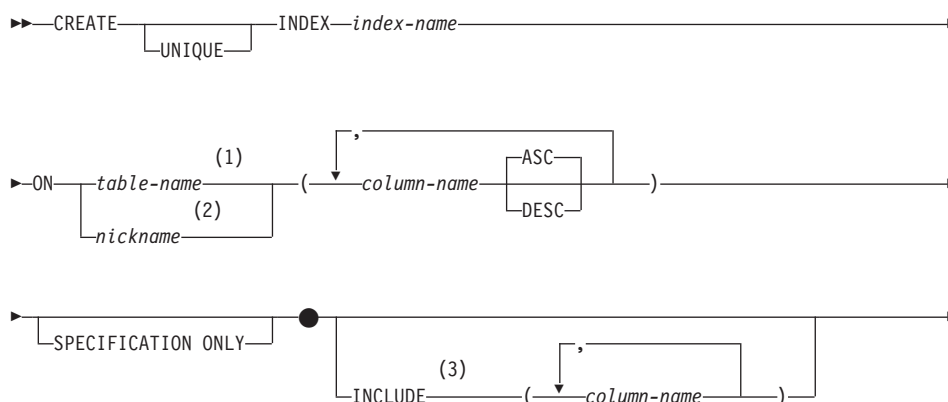
### Syntax
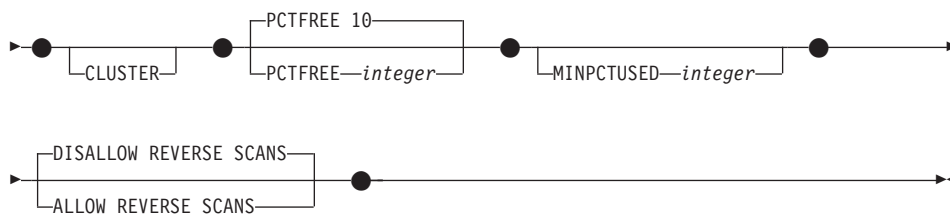
**Notes:**

**1**    In a federated system, the *-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 will create an index specification. INCLUDE, CLUSTER, PCTFREE, MNPCTUSED, DISALLOW REVERSE SCANS, and ALLOW REVERSE SCANS will be disallowed.

**3**    The INCLUDE clause may only be specified if UNIQUE is specified.

## 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. For details refer to "Appendix I. Interaction of Triggers and Constraints" on page 1085.

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.

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 partitioning key, the columns in the index key must be a superset of the partitioning key. That is, the columns specified for a unique index key must include all the columns of the partitioning key (SQLSTATE 42997).

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.

**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. The qualifier must not be SYSIBM, SYSCAT, SYSFUN, or SYSSTAT (SQLSTATE 42939)

**ON** *table-name* or *nickname*

The *table-name* names a table on which an index is to be created. The table must be a base table (not a view) or a summary table described in the catalog. It must *not* name a catalog table (SQLSTATE 42832). If UNIQUE is specified and *table-name* is a typed table, it must not be a subtable (SQLSTATE 429B3). If UNIQUE is specified, the *table-name* cannot be a summary table (SQLSTATE 42809).

*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.

*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.

Each *column-name* must be an unqualified name that identifies a column of the table. 16 columns or less may be specified. If *table-name* is a typed table, 15 columns or less may be specified. If *table-name* is a subtable, at least one *column-name* must be introduced in the subtable (not inherited from a supertable) (SQLSTATE 428DS). No *column-name* may be repeated (SQLSTATE 42711).

The sum of the length attributes of the specified columns must not be greater than 1024 bytes. If *table-name* is a typed table, the sum of the length attributes of the specified columns must not be greater than 1020 bytes. Note that this figure can be reduced by system overhead which varies according to the data type of the column and whether it is nullable. See "Byte Counts" on page 676 for more information on overhead affecting this limit. No LONG VARCHAR, LONG VARGRAPHIC, LOB or DATALINK column may be used as part of an index (even if the length attribute of the column is small enough to fit within the 255 byte limit (SQLSTATE 42962)).

**ASC**

Puts the index entries in ascending order by the column. This is the default.

**DESC**

Puts the index entries in descending order by the column.

**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).

**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 may improve the performance of some queries through index only access. The columns must be distinct from the columns used to enforce uniqueness (SQLSTATE 42711). 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.

*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 or DESC may be specified following the column-name but have no effect on the order.

INCLUDE is disallowed if *nickname* is specified.

**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.

**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. However, if a value greater than 10 is specified, only 10 percent free space will be left in non-leaf pages. The default is 10.

PCTFREE is disallowed if *nickname* is specified.

**MINPCTUSED** *integer*
Indicates whether indexes are reorganized online and the threshold for the minimum percentage of space used on an index leaf page If after a key is deleted from an index leaf page, the percentage of space used on the page

is at or below *integer* percentage, an attempt is made to merge the remaining keys on this page with those of a neighbouring 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. However, a value of 50 or below is recommended for performance reasons.

MINPCTUSED is disallowed if *nickname* is specified.

**DISALLOW REVERSE SCANS**
Specifies that an index only supports forward scans or scanning of the index in the order defined at INDEX CREATE time. This is the default.

DISALLOW REVERSE SCANS is disallowed if *nickname* is specified.

**ALLOW REVERSE SCANS**
Specifies that an index can support both forward and reverse scans; that is, in the order defined at INDEX CREATE time and in the opposite (or reverse) order.

ALLOW REVERSE SCANS is disallowed if *nickname* is specified.

## Rules

- The CREATE INDEX statement will fail (SQLSTATE 01550) if attempting to create an index that matches an existing index. Two index descriptions are considered duplicates if:
  - the set of columns (both key and include columns) and their order in the index is the same as that of an existing index AND
  - the ordering attributes are the same AND
  - both the previously existing index and the one being created are non-unique OR the previously existing index is unique AND
  - if both the previously existing index and the one being created are unique, the key columns of the index being created are the same or a superset of key columns of the previously existing index.

## Notes

- 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. (See *Command Reference* for information about RUNSTATS.) 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.
- 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 prior to creating the actual index. Refer to "SET CURRENT EXPLAIN MODE" on page 867 for more details.
- 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 optimizer uses index specifications to improve access to the data source tables that the specifications apply to.
- For more information about index specifications, see "Index Specifications" on page 48.

## 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
```

## CREATE NICKNAME

The CREATE NICKNAME statement creates a nickname for a data source
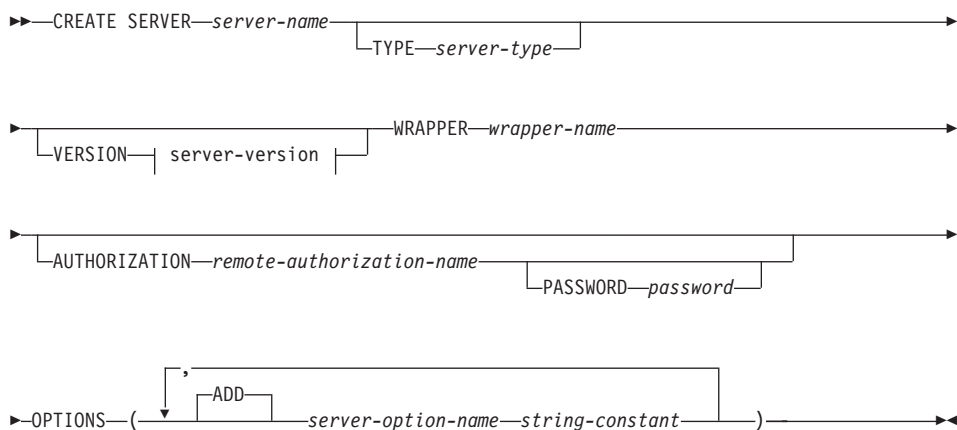table or view.

### 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
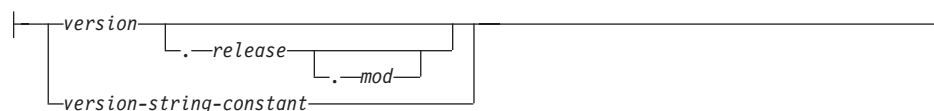applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

### Authorization

The privileges held by the authorization ID of the statement must include at
least one of the following:

- SYSADM or DBADM authority
- 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
  exists

In addition, the user's authorization ID at the data source must hold the
privilege to select from the data source catalog the metadata about the table
or view for which the nickname is being created.

### Syntax

```
►►──CREATE NICKNAME──nickname──FOR──remote-object-name──────────────────────►◄
```

### Description

*nickname*
   Names the federated server's identifier for the table or view that is
   referenced by *remote-object-name*. The nickname, including the implicit or
   explicit qualifier, must not identify a table, view, alias, or nickname
   described in the catalog. The schema name must not begin with SYS
   (SQLSTATE 42939).

*remote-object-name*
   Names a three-part identifier with this format:

   *data-source-name.remote-schema-name.remote-table-name*

   where:

   *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 data source in the CREATE SERVER statement.

*remote-schema-name*
> Names the schema to which the table or view belongs.

*remote-table-name*
> Names either of the following identifiers:
> - The name or an alias of a DB2 family table or view
> - The name of an Oracle table or view

## Notes

- The table or view that the nickname references must already exist at the data source denoted by the first qualifier in *remote-object-name*.
- The federated server does not support those data source data types that correspond to the following DB2 data types: LONG VARCHAR, LONG VARGRAPHIC, DATALINK, large object (LOB) types, and user-defined types. When a nickname is defined for a data source table or view, only those columns in the table or view that have supported data types will be defined to, and can be queried from, the federated database. When the CREATE NICKNAME statement is run against a table or view that has columns with unsupported data types, an error is issued.
- Because data types might be incompatible between data sources, the federated server makes minor adjustments to store remote catalog data locally as needed. Refer to the *Application Development Guide* for details.
- The maximum allowable length of DB2 index names is 18 characters. If a nickname is being created for a table that has an index whose name exceeds this length, the entire name is not cataloged. Rather, DB2 truncates it to 18 characters. If the string formed by these characters is not unique within the schema to which the index belongs, DB2 attempts to make it unique by replacing the last character with 0. If the result is still not unique, DB2 changes the last character to 1. DB2 repeats this process with numbers 2 through 9, and if necessary, with numbers 0 through 9 for the name's seventeenth character, sixteenth character, and so on, until a unique name is generated. To illustrate: The index of a data source table is named ABCDEFGHIJKLMNOPQRSTUVWXYZ. The names ABCDEFGHIJKLMNOPQR and ABCDEFGHIJKLMNOPQ0 already exist in the schema to which this index belongs. The new name is over 18 characters; therefore, DB2 truncates it to ABCDEFGHIJKLMNOPQR. Because this name already exists in the schema, DB2 changes the truncated version to ABCDEFGHIJKLMNOPQ0. Because this latter name exists, too, DB2 changes the truncated version to ABCDEFGHIJKLMNOPQ1. This name does not already exist in the schema, so DB2 now accepts it as a new name.
- When a nickname is created for a table or view, DB2 stores the names of the table's or view's columns in the catalog. If a name exceeds the

maximum allowable length for DB2 column names—128 characters—DB2 truncates the name to this length. If the truncated version is not unique among the other names of the table's or view's columns, DB2 makes it unique by following the procedure described in the preceding paragraph.

## Examples

*Example 1:* Create a nickname for a view, DEPARTMENT, that is in a schema called HEDGES. This view is stored in a DB2 Universal Database for OS/390 data source called OS390A.

```
CREATE NICKNAME DEPT FOR OS390A.HEDGES.DEPARTMENT
```

*Example 2:* Select all records from the view for which a nickname was created in Example 1. The view must be referenced by its nickname. (It can be referenced it by its own name only in pass-through sessions.)

```
SELECT * FROM OS390A.HEDGES.DEPARTMENT   Invalid
SELECT * FROM DEPT                       Valid after nickname DEPT is created
```

## CREATE NODEGROUP

The CREATE NODEGROUP statement creates a new nodegroup within the database and assigns partitions or nodes to the nodegroup, and records the nodegroup definition in the catalog.

### Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be prepared dynamically. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

### Authorization

The authorization ID of the statement must have SYSCTRL or SYSADM or authority.

### Syntax
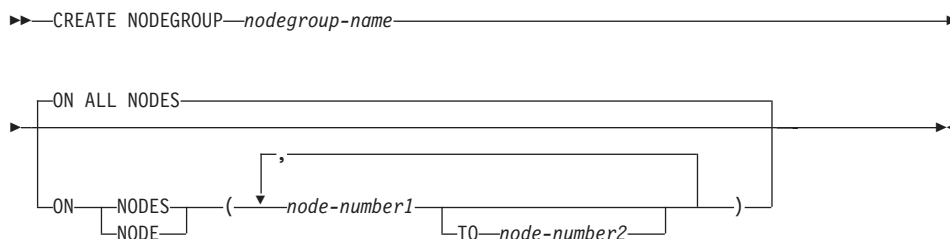
```
►►──CREATE NODEGROUP──nodegroup-name──────────────────────────────►

      ┌─ON ALL NODES───────────────────────────────────────┐
 ►────┤                                                    ├──►◄
      │                       ┌──────,──────┐               │
      └─ON──┬─NODES─┬──(──▼──node-number1──────────────┐──)─┘
            └─NODE──┘              └─TO──node-number2─┘
```

### Description

*nodegroup-name*
> Names the nodegroup. This is a one-part name. It is an SQL identifier (either ordinary or delimited). The *nodegroup-name* must not identify a nodegroup that already exists in the catalog (SQLSTATE 42710). The *nodegroup-name* must not begin with the characters "SYS" or "IBM" (SQLSTATE 42939).

**ON ALL NODES**
> Specifies that the nodegroup is defined over all partitions defined to the database (db2nodes.cfg file) at the time the nodegroup is created.
>
> If a partition is added to the database system, the ALTER NODEGROUP statement should be issued to include this new partition in a nodegroup (including IBMDEFAULTGROUP). Furthermore, the REDISTRIBUTE NODEGROUP command must be issued to move data to the partition. Refer to the *Administrative API Reference* or the *Command Reference* for more information.

## CREATE NODEGROUP

**ON NODES**
> Specifies the specific partitions that are in the nodegroup. NODE is a synonym for NODES.

> *node-number1*
>> Specify a specific partition number.[68]

> **TO** *node-number2*
>> Specify a range of partition numbers. The value of *node-number2* must be greater than or equal to the value of *node-number1* (SQLSTATE 428A9). All partitions between and including the specified partition numbers are included in the nodegroup.

### Rules

- Each partition or node specified by number must be defined in the `db2nodes.cfg` file (SQLSTATE 42729).
- Each *node-number* listed in the ON NODES clause must be appear at most once (SQLSTATE 42728).
- A valid *node-number* is between 0 and 999 inclusive (SQLSTATE 42729).

### Notes

- This statement creates a partitioning map for the nodegroup (Refer to "Data Partitioning Across Multiple Partitions" on page 57 for more information) . A partitioning map identifier (PMAP_ID) is generated for each partitioning map. This information is recorded in the catalog and can be retrieved from SYSCAT.NODEGROUPS and SYSCAT.PARTITIONMAPS. Each entry in the partitioning map specifies the target partition on which all rows that are hashed reside. For a single-partition nodegroup, the corresponding partitioning map has only one entry. For a multiple partition nodegroup, the corresponding partitioning map has 4 096 entries, where the partition numbers are assigned to the map entries in a round-robin fashion, by default.

### Example

Assume that you have a partitioned database with six partitions defined as: 0, 1, 2, 5, 7, and 8.

- Assume that you want to create a nodegroup call MAXGROUP on all six partitions. The statement is as follows:

```
CREATE NODEGROUP MAXGROUP
ON ALL NODES
```

- Assume that you want to create a nodegroup MEDGROUP on partitions 0, 1, 2, 5, 8. The statement is as follows:

---

68. node-name of the form 'NODEnnnnn' may be specified for compatibility with the previous version.

618

```
CREATE NODEGROUP MEDGROUP
 ON NODES (0 TO 2, 5, 8)
```

- Assume that you want to create a single-partition nodegroup MINGROUP on partition (or node) 7. The statement is as follows:

```
CREATE NODEGROUP MINGROUP
 ON NODE (7)
```

**Note:** The singular form of the keyword NODES is also accepted.

## CREATE PROCEDURE

This statement is used to register a stored procedure with an 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. 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 as least one of the following:

- SYSADM or DBADM authority
- IMPLICIT_SCHEMA authority on the database, if the implicit or explicit schema name of the procedure does not exist
- CREATEIN privilege on the schema, if the schema name of the procedure refers to an existing schema.

To create a not-fenced stored procedure, the privileges held by the authorization ID of the statement must also include at least one of the following:

- CREATE_NOT_FENCED authority on the database
- SYSADM or DBADM authority.

To create a fenced stored procedure, no additional authorities or privileges are required.

If the authorization ID has insufficient authority to perform the operation, an error (SQLSTATE 42502) is raised.

### Syntax

```
►►──CREATE PROCEDURE───────────────────────────────────────────►
```

```
►►─procedure-name─(─┬──────────────────────────────────────┬─)─●────────►
                    │    ┌─────────────,───────────────┐    │
                    │    │  ┌─IN────┐                   │    │
                    └────▼──┼───────┼──parameter-name─data-type─┘
                           ├─OUT───┤
                           └─INOUT─┘

                                      ┌─DYNAMIC RESULT SETS 0──────┐        (1)
►─┬───────────────────────┬─●─────────┼────────────────────────────┼─●───────►
  └─SPECIFIC─specific-name─┘           └─DYNAMIC RESULT SETS─integer─┘

►─EXTERNAL─┬─────────────────────────┬─●─LANGUAGE─┬─C─────┬─────────────────►
           │        ┌─'string'─────┐ │            ├─JAVA──┤
           └─NAME───┴─identifier───┘ │            └─COBOL─┘

                                          ┌─NOT DETERMINISTIC─┐
►─●─PARAMETER STYLE─┬─DB2DARI─────────┬─●─┼───────────────────┼─●───────────►
                    │             (2) │   └─DETERMINISTIC─────┘
                    ├─DB2GENERAL──────┤
                    ├─GENERAL─────────┤
                    ├─GENERAL WITH NULLS─┤
                    ├─DB2SQL──────────┤
                    └─JAVA────────────┘

   ┌─FENCED─────┐      ┌─NO DBINFO─┐       ┌─CALLED ON NULL INPUT──┐  (3)
►─┬┼────────────┼─●───┬┼───────────┼─●────┼───────────────────────┼─●────────►
  │└─NOT FENCED─┘     │└─DBINFO────┘       └───────────────────────┘
  
►─┬────────────────────────────┬─►◄
  └─PROGRAM TYPE─┬─SUB──┬───────┘
                 └─MAIN─┘
```
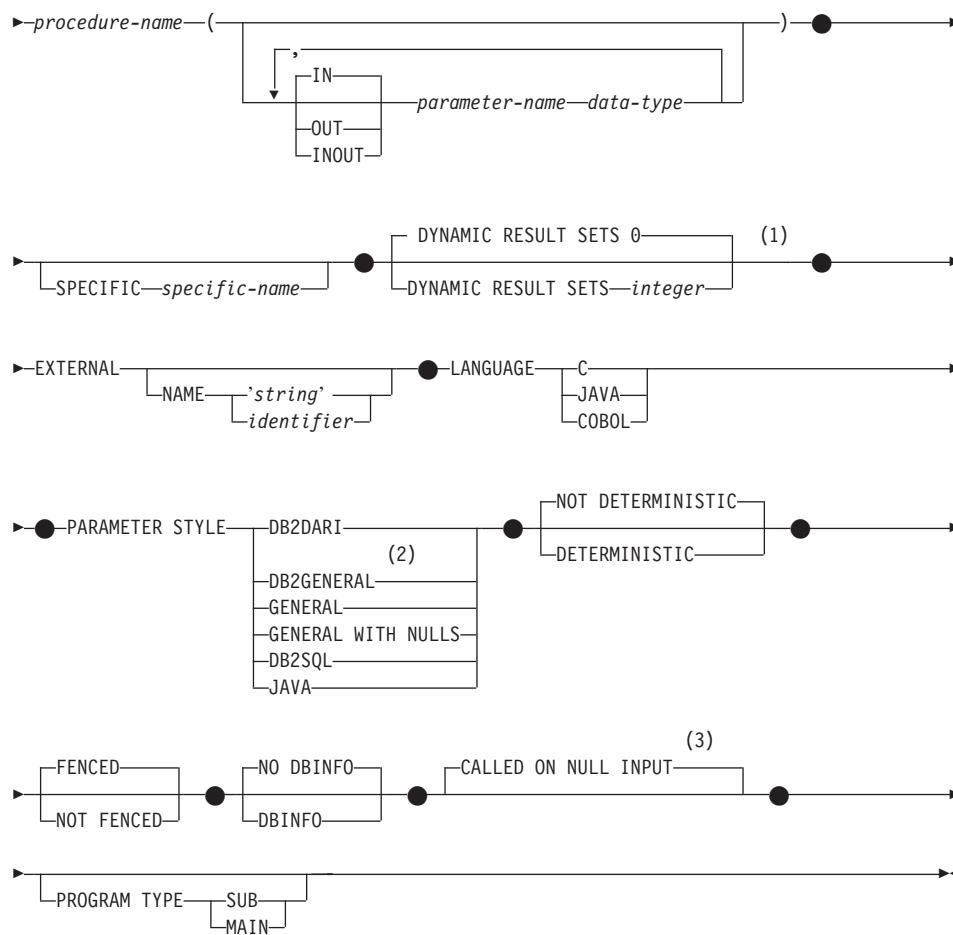
**Notes:**

**1**     RESULT SETS may be specified in place of DYNAMIC RESULT SETS.

**2**     DB2GENRL may be specified in place of DB2GENERAL, SIMPLE CALL may be specified in place of GENERAL and SIMPLE CALL WITH NULLS may be specified in place of GENERAL WITH NULLS.

**3**     NULL CALL may be specified in place of CALLED ON NULL INPUT.

## Description

*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 (with a maximum length of 18). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier

## CREATE PROCEDURE

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, while of course unique within its schema, need not be unique across schemas.

The a two-part name is specified, the *schema-name* cannot begin with "SYS". Otherwise, an error (SQLSTATE 42939) is raised.

**( IN | OUT | INOUT** *parameter-name data-type,...)*
Identifies the parameters of the procedure, and specifies the mode, name and data type 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. 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). 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 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 of the procedure are the same even if the data types are not.

**IN | OUT | INOUT**
Specifies the mode of the parameter.

- IN - parameter is input only
- OUT - parameter is output only
- INOUT - parameter is both input and output

*parameter-name*
Specifies the name of the parameter.

*data-type*
> Specifies the data type of the parameter.
>
> - SQL data type specifications and abbreviations which may be specified in the *data-type* definition of a CREATE TABLE statement and have a correspondence in the language that is being used to write the procedure may be specified. See the language-specific sections of the *Application Development Guide* for details on the mapping between the SQL data types and host language data types with respect to stored procedures.
> - User-defined data types are not supported (SQLSTATE 42601).

**SPECIFIC** *specific-name*
> Provides a unique name for the instance of the procedure that is being defined. This specific name can be used when dropping the procedure or commenting on the procedure. It can never be used to invoke the procedure. 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 procedure 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, SQLyymmddhhmmsshhn.

**DYNAMIC RESULT SETS** *integer*
> Indicates the estimated upper bound of returned result sets for the stored procedure. Refer to "Returning Result Sets from Stored Procedures" on page 489 for more information.
>
> The value RESULT SETS may be used as a synonym for DYNAMIC RESULT SETS for backwards and family compatability.

**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 NAME clause is not specified "NAME *procedure-name*" is assumed.
>
> **NAME** *'string'*
>> This clause identifies the name of the user-written code which implements the procedure being defined.

# CREATE PROCEDURE

The 'string' option is a string constant with a maximum of 254 characters. 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 stored 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.

  ```
  ►►─'─┬─library_id────────┬─────────────────'───────────────►◄
       └─absolute_path_id──┘  └─!─proc_id──┘
  ```

  The name must be enclosed in single quotes. Extraneous blanks are not permitted within the single quotes.

  *library_id*
  Identifies the library name containing the procedure. The database manager will look for the library in the .../sqllib/function/unfenced directory and the .../sqllib/function directory (UNIX-based systems), or ...\\*instance_name*\\function\\unfenced directory and the ...\\*instance_name*\\function directory (OS/2, Windows 95 and Windows NT as specified by the DB2INSTPROF registry variable), where the database manager will locate the controlling sqllib directory which is being used to run the database manager. For example, the controlling sqllib directory in UNIX-based systems is /u/$DB2INSTANCE/sqllib.

  If 'myproc' were the *library_id* in a UNIX-based system it would cause the database manager to look for the procedure in library /u/production/sqllib/function/unfenced/myfunc and /u/production/sqllib/function/myfunc, provided the database manager is being run from /u/production.

  For OS/2, Windows 95 and Windows NT, the database manager will look in the LIBPATH or PATH if the *library_id* is not found in the function directory, and will be run as fenced.

  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.

In a UNIX-based system, for example,
'/u/jchui/mylib/myproc' would cause the database manager to
look in /u/jchui/mylib for the myproc procedure.

In OS/2, Windows 95 and Windows NT 'd:\mylib\myproc'
would cause the database manager to load the myproc.dll file
from the d:\mylib directory.

If absolute path is specified, the procedure will run as fenced,
ignoring the FENCED or NOT FENCED attribute.

*! proc_id*

Identifies the entry point name of the procedure to be invoked.
The ! serves as a delimiter between the library id and the
procedure id. If *! proc_id* is omitted, the database manager will
use the default entry point established when the library was
linked.

In a UNIX-based system, for example, 'mymod!proc8' would
direct the database manager to look for the library
$inst_home_dir/sqllib/function/mymod and to use entry point
proc8 within that library.

In OS/2, Windows 95 and Windows NT 'mymod!proc8' would
direct the database manager to load the mymod.dll file and call
the proc8() procedure in the dynamic link library (DLL).

If the string is not properly formed, an error (SQLSTATE 42878) is
raised.

The body of every stored procedure should be in a directory which
is mounted and available on every partition of the database.

- 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 stored 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 (SQLSTATE
42884) is raised.

```
►►─'──────────────────class_id──┬───┬──method_id─'────────────►◄
       └─jar_id :─┘              └─!─┘
```

The name must be enclosed in single quotes. Extraneous blanks are

not permitted within the single quotes.

*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 Java virtual machine will look in directory '../myPacks/StoredProcs/' for the classes. In OS/2 and Windows 95 and Windows NT, the Java virtual machine will look in directory '..\myPacks\StoredProcs\'.

*method_id*
> Identifies the method name with the Java class 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.

**LANGUAGE**
This mandatory clause is used to specify the language interface convention to which the stored procedure body is written.

**C**
> This means the database manager will call the stored procedure as if it were a C procedure. The stored 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 stored 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.

**PARAMETER STYLE**
This clause is used to specify the conventions used for passing parameters to and returning the value from stored procedures.

**DB2DARI**
> This means that the stored procedure will use a parameter passing

convention that conforms to C language calling and linkage conventions. This can only be specified when LANGUAGE C is used.

**DB2GENERAL**

This means that the stored 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.

The value DB2GENRL may be used as a synonym for DB2GENERAL.

**GENERAL**

This means that the stored procedure will use a parameter passing mechanism where the stored 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 or COBOL is used.

Null indicators are NOT directly passed to the program.

The value SIMPLE CALL may be used as a synonym for GENERAL.

**GENERAL WITH NULLS**

In addition to the parameters on the CALL statement as specified in GENERAL, another argument is passed to the stored procedure. This additional argument contains a vector of null indicators for each of the parameters on the CALL statement. In C, this would be an array of short ints. This can only be specified when LANGUAGE C or COBOL is used.

The value SIMPLE CALL WITH NULLS may be used as a synonym for GENERAL WITH NULLLS.

**DB2SQL**

In addition to the parameters on the CALL statement, the following arguments are passed to the stored procedure:

- a NULL indicator for each parameter on the CALL statement
- the SQLSTATE to be returned to DB2
- the qualified name of the stored procedure
- the specific name of the stored procedure
- the SQL diagnostic string to be returned to DB2

This can only be specified when LANGUAGE C or COBOL is used.

**JAVA** This means that the stored 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.

Refer to *Application Development Guide* for details on passing parameters.

**DETERMINISTIC** or **NOT DETERMINISTIC**
This 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.

This clause is not used by DB2 Universal Database.

**FENCED** or **NOT FENCED**
This clause specifies whether or not the stored procedure is considered "safe" to run in the database manager operating environment's process or address space (NOT FENCED), or not (FENCED).

If a stored procedure is registered as FENCED, the database manager insulates its internal resources (e.g. 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.

If the stored procedure is located in .../sqllib/function/unfenced directory and the .../sqllib/function directory (UNIX-based systems), or ...\\*instance_name*\function\unfenced directory and the ...\\*instance_name*\function directory (OS/2, Windows 95 and Windows NT), then the FENCED or NOT FENCED registered attribute (and every other registered attribute) will be ignored.

**Note:** Use of NOT FENCED for procedures not adequately checked out can compromise the integrity of DB2. DB2 takes 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.

To change from FENCED to NOT FENCED, the procedure must be re-registered (by first dropping it and then re-creating it). Either SYSADM authority, DBADM authority or a special authority (CREATE_NOT_FENCED) is required to register a stored procedures as NOT FENCED.

**NO DBINFO or DBINFO**

Specifies whether specific information known by DB2 is passed to the stored procedure when it is invoked as an additional invocation-time argument (DBINFO) or not (NO DBINFO). NO DBINFO is the default.

If DBINFO is specified, then a structure is passed to the stored procedure 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 run-time authorization ID.
- Code page - identifies the database code page.
- Schema name - not applicable to stored procedures.
- Table name - not applicable to stored procedures.
- Column name - not applicable to stored procedures.
- Database version/release - identifies the version, release and modification level of the database server invoking the stored procedure.
- Platform - contains the server's platform type.
- Table function result column numbers - not applicable to stored procedures.

DBINFO is not supported for PARAMETER STYLE JAVA.

Please see the *Application Development Guide* for detailed information on the structure and how it is passed to the stored procedure.

**CALLED ON NULL INPUT**

CALLED ON NULL INPUT always applies to stored procedures. This means that regardless if any arguments are null, the stored procedure is called. It can return a null value or a normal (non-null) value. Responsibility for testing for null argument values lies with the stored procedure.

The value NULL CALL may be used as a synonym for CALLED ON NULL INPUT for backwards and family compatability.

**PROGRAM TYPE**

Specifies whether the stored procedure expects parameters in the style of a main routine or a subroutine.

**SUB**

The stored procedure expects the parameters to be passed as separate arguments.

## CREATE PROCEDURE

**MAIN**

The stored procedure expects the parameters to be passed as an argument counter, and a vector of arguments (argc, argv). The name of the stored 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 as opposed to a standalone executable.

The default for PROGRAM TYPE on is SUB. PROGRAM TYPE MAIN is only valid for LANGUAGE C and PROGRAM STYLE GENERAL, GENERAL WITH NULLS and DB2SQL.

### Notes

- For information on creating the programs for a stored procedure, see the *Application Development Guide*.
- Creating a procedure 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.

### Examples

*Example 1:* Create the procedure definition for a stored procedure, written in Java, that is passed a part number and returns the cost of the part and the quantity that are 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 stored 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'
               RESULT SETS 1 NOT FENCED
               LANGUAGE C PARAMETER STYLE GENERAL
```

## 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. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

### Authorization

An authorization ID that holds SYSADM or DBADM authority can create a schema with any valid *schema-name* or *authorization-name*.

An authorization ID that does not hold SYSADM or 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 any *schema-SQL-statement*s the privileges held by the *authorization-name* (if not specified, it defaults to the authorization ID of the statement) must include at least one of the following:
* The privileges required to perform each of the *schema-SQL-statement*s
* SYSADM or DBADM authority.

### Syntax

```
►►──CREATE SCHEMA──┬─schema-name──────────────────────────────────┬──►
                   ├─AUTHORIZATION──authorization-name─────────────┤
                   └─schema-name──AUTHORIZATION──authorization-name─┘

►──┬──────────────────────────────┬──►◄
   │   ┌──────────────────────┐    │
   └───▼─schema-SQL-statement──┴────┘
```
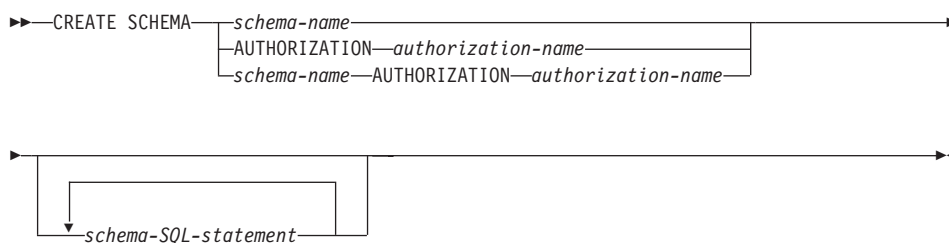
### Description

*schema-name*
> 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.

## CREATE SCHEMA

**AUTHORIZATION** *authorization-name*
> Identifies the user that is the owner of the schema. The value *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* with the user called *authorization-name* as the schema owner. The *schema-name* must not identify a schema-name for a schema already described in the catalog (SQLSTATE 42710). The *schema-name* cannot begin with "SYS" (SQLSTATE 42939).

*schema-SQL-statement*
> SQL statements that can be included as part of the CREATE SCHEMA statement are:
> - CREATE TABLE statement excluding typed tables and summary tables (see "CREATE TABLE" on page 639)
>
> - CREATE VIEW statement excluding typed views(see "CREATE VIEW" on page 714)
>
> - CREATE INDEX statement (see "CREATE INDEX" on page 608)
>
> - COMMENT ON statement (see "COMMENT ON" on page 494)
>
> - GRANT statement (see "GRANT (Table, View, or Nickname Privileges)" on page 801).

### 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.

- 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.
- If the AUTHORIZATION clause is specified and DCE authentication is
  used, the group membership of the *authorization-name* specified will not be
  considered in evaluating the authorizations required to perform the
  statements that follow the clause. If the *authorization-name* specified is
  different than the authorization id creating the schema, an authorization
  failure may result during the execution of the CREATE SCHEMA statement.

## 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,
```

## CREATE SCHEMA

```
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) )
```

## CREATE SERVER

The CREATE SERVER statement[69] defines a data source to a federated database.

### 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 authorization ID of the statement must have SYSADM or DBADM authority on the federated database.

### Syntax

```
►►──CREATE SERVER──server-name──────────────────────────────────────►
                              └─TYPE──server-type─┘

►──────────────────────────────────WRAPPER──wrapper-name────────────►
      └─VERSION─┤ server-version ├─┘

►────────────────────────────────────────────────────────────────────►
      └─AUTHORIZATION──remote-authorization-name──────────────────┘
                                           └─PASSWORD──password─┘

              ┌─────────────,───────────────────────────────────┐
►─OPTIONS──(──▼──┬─ADD─┬──server-option-name──string-constant──┴──)──►◄
                 └─ADD─┘
```

**server-version:**

```
├──┬─version──────────────────────────────────────────┬──┤
   │         └─.──release──────────────┘               │
   │                     └─.──mod─┘                    │
   └─version-string-constant───────────────────────────┘
```

---

69. 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. For information about federated systems, see "DB2 Federated Systems" on page 39. For information about DRDA application servers, see "Distributed Relational Database" on page 28.

## CREATE SERVER

### Description

*server-name*
> Names 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.

**TYPE** *server-type*
> Specifies the type of the data source denoted by *server-name*. Refer to "Appendix E. Federated Systems: Server Types; Options; Default Type Mappings; Pass-Through Guidelines" on page 1047 for a list of supported data source types.

**VERSION**
> Specifies the version of the data source denoted 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*
> Names the wrapper that the federated server uses to interact with data sources of the type and version denoted by *server-type* and *'server-version'*.

**AUTHORIZATION** *remote-authorization-name*
> Specifies the authorization ID under which any necessary actions are performed at the data source when the CREATE SERVER statement is processed. This ID must hold the authority (BINDADD or its equivalent) that the necessary actions require.

**PASSWORD** *password*
> Specifies the password associated with the authorization ID represented by *remote-authorization-name.*

**OPTIONS**
> Indicates what server options are to be enabled. Refer to "Server Options" on page 1049 for descriptions of *server-option-name*s and their settings.

> **ADD**
> > Enables one or more server options.

*server-option-name*
> Names a server option that will be used to either configure or provide information about the data source denoted by *server-name*.

*string-constant*
> Specifies the setting for *server-option-name* as a character string constant.

## Notes

- If *remote-authorization-name* is not specified, the authorization ID for the federated database will be used.

- The *password* should be specified in the case required by the data source; if any letters in *password* must be in lowercase, enclose *password* in quotation marks. If an *identifier* is specified but not *password*, the authentication type of the data source denoted by *server-name* is assumed to be CLIENT.

- If a server option is set to one value for a type of data source, and this same option set to another value for an instance of this type, the second value overrides the first for the instance. For example, suppose that PASSWORD is set to 'Y' (yes, validate passwords at the data source) for a federated system's DB2 Universal Database for OS/390 data sources. Then later, this option's default ('N') is used for a specific DB2 Universal Database for OS/390 data source named SIBYL. As a result, passwords will be validated at all of the DB2 Universal Database for OS/390 data sources except SIBYL.

## Examples

*Example 1:* Define a DB2 for MVS/ESA 4.1 data source that is accessible through a wrapper called DB2WRAP. Call the data source CRANDALL. In addition, specify that:

- MURROW and DROWSSAP will be the authorization ID and password under which packages are bound at CRANDALL when this statement is processed.

- CRANDALL is defined to the DB2 RDBMS as an instance called MYNODE.

- When the federated server accesses CRANDALL, it will be connected to a database called MYDB.

- The authorization IDs and passwords under which CRANDALL can be accessed are to be sent to CRANDALL in uppercase.

- MYDB and the federated database use the same collating sequence.

```
CREATE SERVER CRANDALL
    TYPE DB2/MVS
    VERSION 4.1
    WRAPPER DB2WRAP
    AUTHORIZATION MURROW
    PASSWORD DROWSSAP
    OPTIONS
```

```
( NODE 'MYNODE',
DBNAME 'MYDB',
FOLD_ID 'U',
FOLD_PW 'U',
COLLATING_SEQUENCE 'Y' )
```

*Example 2:* Define an Oracle 7.2 data source that's accessible through a wrapper called KLONDIKE. Call the data source CUSTOMERS. Specify that:

- CUSTOMERS is defined to the Oracle RDBMS as an instance called ABC.

Provide these statistics for the optimizer:

- 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 7.2
   WRAPPER KLONDIKE
   OPTIONS
       ( NODE 'ABC',
       CPU_RATIO '2.0',
       IO_RATIO '1.5' )
```

## 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 may include other attributes of the table, such as its primary key or check constraints.

### 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 at least one of the following:
- SYSADM or DBADM authority
- CREATETAB authority on the database and one of:
  - 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.

If a subtable is being defined, the authorization ID must be the same as the definer of the root table of the table hierarchy.

To define a foreign key, the privileges held by the authorization ID of the statement must include one of the following on the parent table:
- REFERENCES privilege on the table
- REFERENCES privilege on each column of the specified parent key
- CONTROL privilege on the table
- SYSADM or DBADM authority.

To define a summary table (using a fullselect) the privileges held by the authorization ID of the statement must include at least one of the following on each table or view identified in the fullselect:
- SELECT privilege on the table or view and ALTER privilege if REFRESH DEFERRED or REFRESH IMMEDIATE is specified
- CONTROL privilege on the table or view
- SYSADM or DBADM authority.

### Syntax

```
►►─CREATE──────────────TABLE──table-name───────────────────►
           └─SUMMARY─┘
```
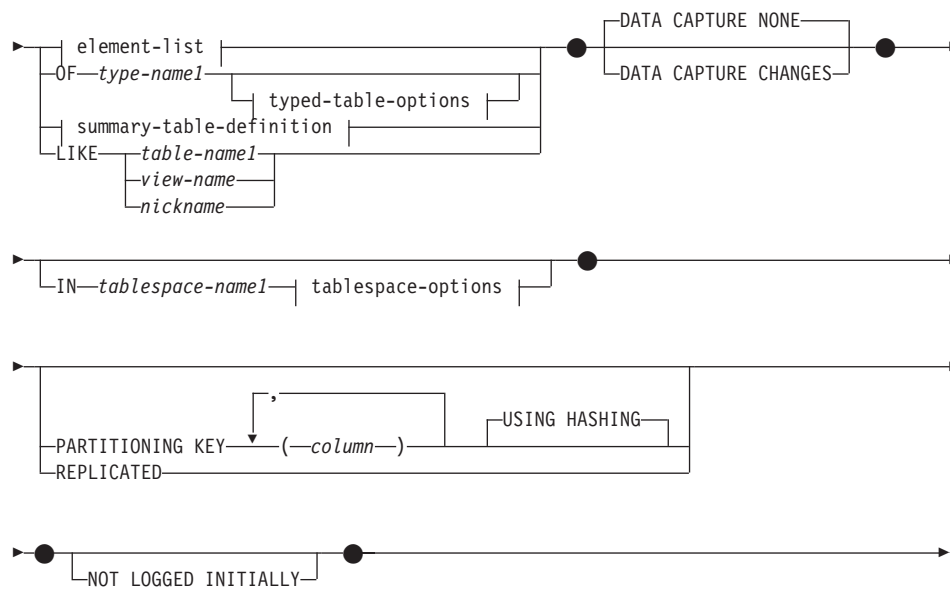
## CREATE TABLE



**element-list:**
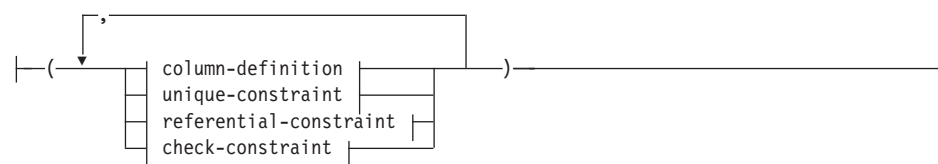


**typed-table-options:**



**under-clause:**



**typed-element-list:**
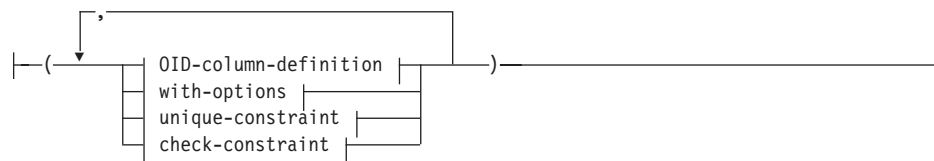
```
        ┌─────────────,───────────────────────┐
├──(──▼──┬─ OID-column-definition ─┬──────┬──)──────────────────────────────┤
         ├─ with-options ──────────┤
         ├─ unique-constraint ─────┤
         └─ check-constraint ──────┘
```

**summary-table-definition:**

```
├──AS──(──fullselect──)──┤ summary-table-options ├──────────────────────────┤
```

**summary-table-options:**

```
├──┬─DEFINITION ONLY─────────────────────────────────────────────────┬──────┤
   └─DATA INITIALLY──DEFERRED──REFRESH──┬─DEFERRED──┬──────────────────┘
                                        └─IMMEDIATE─┘
```

**tablespace-options:**

```
├──┬───────────────────────────────────┬──┬─────────────────────────┬───────┤
   │                             (1)    │  └─LONG IN──tablespace-name3─┘
   └─INDEX IN──tablespace-name2──────────┘
```

**column-definition:**

```
├──column-name──┤ data-type ├──┬────────────────────┬───────────────────────┤
                              └─┤ column-options ├──┘
```
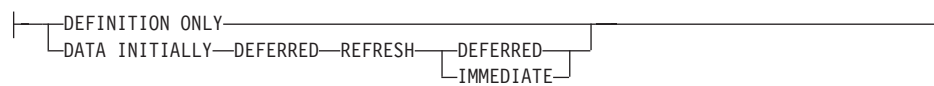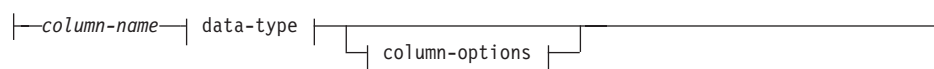
**CREATE TABLE**

**column-options:**

```
            ┌──────────────────────────────────────────────────────────┐
  ├─────────┼──NOT NULL────────────────────────────────────────────────┼───────┤
            │                                                          │
            ├──┤ default-clause ├────────────────────────────────────  │
            │                   (2)                                     │
            ├──┤ lob-options ├─────────────────────────────────────    │
            │                      (3)                                  │
            ├──┤ datalink-options ├──────────────────────────────      │
            │                          (4)                              │
            ├──SCOPE──┬──typed-table-name──┬───────────────────────     │
            │         └──typed-view-name───┘                            │
            │                                                           │
            │                                    ┌──PRIMARY KEY──┐      │
            │            (5)                      ├──UNIQUE───────┤      │
            └──CONSTRAINT──────constraint-name──┬┤ references-clause ├─┘
                                                 └──CHECK──(──check-condition──)──┘
```
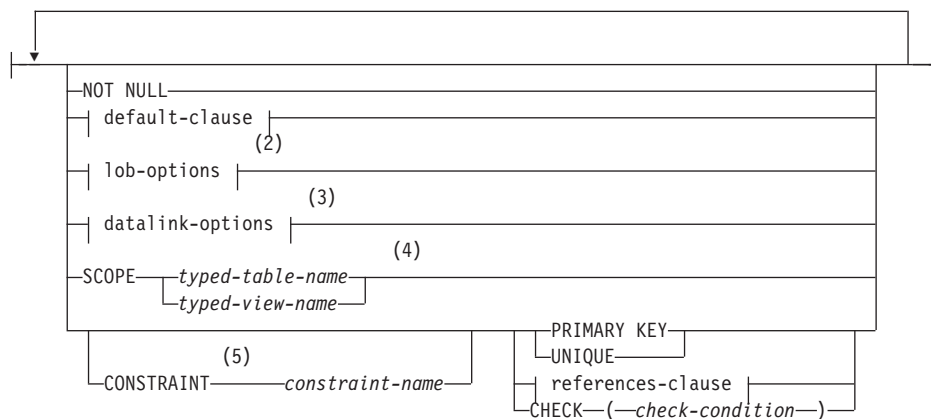
**Notes:**

**1** Specifying which table space will contain a table's index can only be done when the table is created.

**2** The lob-options clause only applies to large object types (BLOB, CLOB and DBCLOB) and distinct types based on large object types.

**3** The datalink-options clause only applies to the DATALINK type and distinct types based on the DATALINK type. The LINKTYPE URL clause is required for these types.

**4** The SCOPE clause only applies to the REF type.

**5** For compatibility with Version 1, the CONSTRAINT keyword may be omitted in a *-column-definition* defining a references-clause.

642

**data-type:**

```
├──┬─SMALLINT───────────────────────────────────────────────────────────────┬──┤
   ├─┬─INTEGER─┬──────────────────────────────────────────────────────────────┤
   │ └─INT────┘                                                                │
   ├─BIGINT──────────────────────────────────────────────────────────────────┤
   ├─FLOAT──┬──────────────┬─────────────────────────────────────────────────┤
   │        └─(─integer─)──┘                                                  │
   ├─REAL────────────────────────────────────────────────────────────────────┤
   ├─DOUBLE──┬─PRECISION─┬───────────────────────────────────────────────────┤
   │         └───────────┘                                                    │
   ├─┬─DECIMAL─┬──┬────────────────────────┬─────────────────────────────────┤
   │ ├─DEC─────┤  └─(─integer─┬──────────┬─)┘                                 │
   │ ├─NUMERIC─┤              └─,integer─┘                                    │
   │ └─NUM─────┘                                                              │
   ├─┬─┬─CHARACTER─┬──┬───────────┬────────────────┬───┬──────────────────┬──┤
   │ │ └─CHAR──────┘  └─(integer)─┘                │   │        (1)       │  │
   │ ├─┬─VARCHAR──────────────┬──(─integer─)───────┤   └─FOR BIT DATA─────┘  │
   │ │ └─┬─CHARACTER─┬─VARYING─┘                    │                        │
   │ │   └─CHAR──────┘                              │                        │
   │ └─LONG VARCHAR─────────────────────────────────┘                        │
   ├─┬─BLOB───┬──(─integer─┬─────┬─)─────────────────────────────────────────┤
   │ ├─CLOB───┤            ├─K───┤                                           │
   │ └─DBCLOB─┘            ├─M───┤                                           │
   │                       └─G───┘                                           │
   ├─GRAPHIC──┬───────────┬──────────────────────────────────────────────────┤
   │          └─(integer)─┘                                                   │
   ├─VARGRAPHIC─(integer)────────────────────────────────────────────────────┤
   ├─LONG VARGRAPHIC─────────────────────────────────────────────────────────┤
   ├─DATE────────────────────────────────────────────────────────────────────┤
   ├─TIME────────────────────────────────────────────────────────────────────┤
   ├─TIMESTAMP───────────────────────────────────────────────────────────────┤
   ├─DATALINK──┬───────────────┬─────────────────────────────────────────────┤
   │           └─(─integer─)───┘                                             │
   ├─distinct-type-name──────────────────────────────────────────────────────┤
   └─REF─(type-name2)────────────────────────────────────────────────────────┘
```
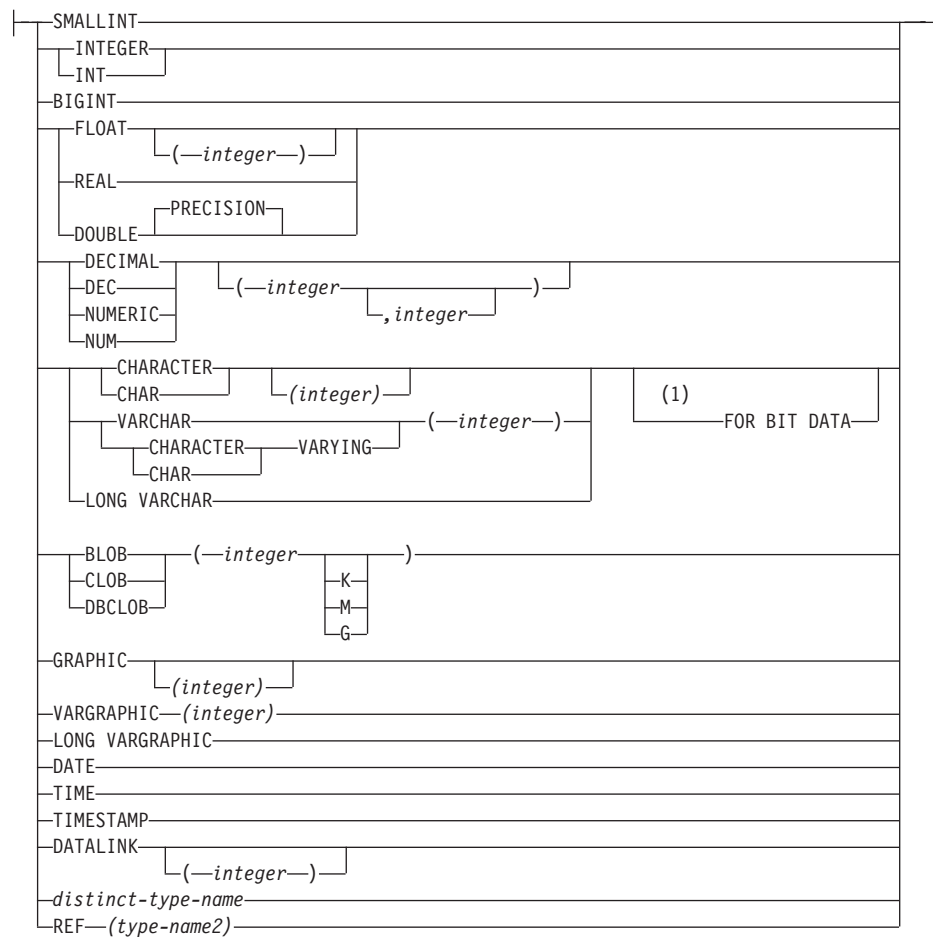
**Notes:**

**1** The FOR BIT DATA clause may be specified in random order with the other column constraints that follow.

**default-clause:**

```
├──┬─WITH─┬──DEFAULT──┬────────────────┬──────────────────────────────────────┤
   └──────┘           └─default-values─┘
```

## CREATE TABLE

**default-values:**

```
├─ constant ──────────────────────────────────────────┤
   datetime-special-register
   USER
   NULL
   cast-function ─(─ constant ─────────────────── )
                     datetime-special-register
                     USER
```

**lob-options:**

```
├─● LOGGED ──── ● NOT COMPACT ── ● ──────────────────┤
      NOT LOGGED       COMPACT
```

**datalink-options:**

```
├─ LINKTYPE URL ─ NO LINK CONTROL ───────────────────────┤
                  FILE LINK CONTROL ─ file-link-options ─
                                      MODE DB2OPTIONS
```

**file-link-options:**

```
├─● INTEGRITY ── ALL ─● READ PERMISSION ─ FS ──────────────►
                                          DB

►─● WRITE PERMISSION ─ FS ────── ● RECOVERY ─ NO ──────────►
                       BLOCKED              YES

►─● ON UNLINK ─ RESTORE ─ ● ──────────────────────────────┤
                DELETE
```

**references-clause:**

```
├─ REFERENCES ─ table-name ──────────── rule-clause ─┤
               (─ column-name ─)
```

**rule-clause:**

```
       ┌─ON DELETE NO ACTION──────┐      ┌─ON UPDATE NO ACTION──┐
●──┬──────────────────────────────┼──●──┬──────────────────────┼──●───────
   └─ON DELETE──┬─RESTRICT───┬─────┘     └─ON UPDATE RESTRICT───┘
                ├─CASCADE────┤
                └─SET NULL───┘
```

**unique-constraint:**

```
                                                   ┌──────,──────┐
├──┬──────────────────────────────────┬──┬─UNIQUE──────┬──(──▼─column-name──┴──)──┤
   └─CONSTRAINT─constraint-name────────┘  └─PRIMARY KEY─┘
```

**referential-constraint:**

```
                                                          ┌────,──────┐
├──┬──────────────────────────────────┬──FOREIGN KEY──(──▼─column-name──┴──)──►
   └─CONSTRAINT─constraint-name────────┘
                              (1)
►──┤ references-clause ├──────────────────────────────────────────────────────┤
```

**check-constraint:**

```
├──┬──────────────────────────────────┬──CHECK──(─check-condition─)──────┤
   └─CONSTRAINT─constraint-name────────┘
```

**OID-column-definition:**

```
├──REF IS─OID-column-name─USER GENERATED───────────────────────────────┤
```

**with-options:**

```
├──column-name──WITH OPTIONS──┤ column-options ├───────────────────────┤
```

**Notes:**

**1**    For compatibility with Version 1, constraint-name may be
specified following FOREIGN KEY (without the CONSTRAINT
keyword).

# CREATE TABLE

## Description

**SUMMARY**
Indicates that a summary table is being defined. The keyword is optional, but when specified, the statement must include a *summary-table-definition*.

*table-name*
Names the table. The name, including the implicit or explicit qualifier, must not identify a table, view, or alias described in the catalog. The schema name must not be SYSIBM, SYSCAT, SYSFUN, or SYSSTAT (SQLSTATE 42939).

**OF** *type-name1*
Specifies that the columns of the table are based on the attributes of the structured type 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 a 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*) as 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 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 being created (SQLSTATE 428DQ). If this clause is omitted when defining the root table, a name is generated by the system consisting of the name of the table being created followed by a unique suffix such that the identifier is unique within the identifiers of the existing tables, views, aliases, 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 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 identified by *supertable-name* must not have any existing subtable already defined using *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 partitioning 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 will be granted an equivalent privilege on the newly created subtable. The subtable definer is considered to be the grantor of this privilege.

*element-list*

Defines the elements of a table. This includes the definition of columns and constraints on the table.

*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.

*summary-table-definition*

If the table definition is based on the result of a query, then the table is a summary table based on the query.

**AS**

Introduces the query that is used for the definition of the table and to determine the data included in the table.

*fullselect*

Defines the query in which the table is based. The resulting column definitions are the same as those for a view defined with the same query.

Every select list element must have a name (use the AS clause for expressions - see page "select-clause" on page 369 for details) . The *summary-table-options* specified define attributes of the summary table. The option chosen also defines the contents of the fullselect as follows.

When DEFINITION ONLY is specified, any valid fullselect that does not reference a typed table or typed view can be specified.

When REFRESH DEFERRED or REFRESH IMMEDIATE is specified, the fullselect cannot include:

- references to a nickname, summary 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 summary table
- expressions that are a reference type or DATALINK type (or distinct type based on these types)
- functions that have external action
- functions that depend on physical characteristics (for example NODENUMBER, PARTITION)
- table or view references to system objects (explain tables also should not be specified).

When REFRESH IMMEDIATE is specified:
- the fullselect must be a subselect
- the subselect cannot include:
  - functions that are not deterministic
  - scalar fullselects
  - predicates with fullselects
  - special registers
- a GROUP BY clause must be included in the subselect
- The supported column functions are SUM, COUNT and COUNT_BIG (without DISTINCT). The select list must contain a COUNT(*) or COUNT_BIG(*) column. If the summary table select list contains SUM(X) where X is a nullable argument, then the summary table must also have COUNT(X) in its select list. These column functions cannot be part of any expressions.
- The FROM clause reference rules are the same as for REFRESH DEFERRED summary tables with the additional restriction that the FROM clause of a REFRESH IMMEDIATE summary table must not involve table functions or the VALUES clause
- 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
- all GROUP BY items must be included in the select list
- no grouping sets are allowed (including CUBE and ROLLUP) or grouping on constants
- a HAVING clause is not allowed
- if in a multiple partition nodegroup, then a partitioning key must be a subset of the group by items.

Furthermore, for REFRESH IMMEDIATE, the base table must have at least one unique index defined and the SELECT clause must include all of the columns of this unique index.

*summary-table-options*
> Define the attributes of the summary table.

> **DEFINITION ONLY**
>> The query is used only to define the table. The table is not populated using the results of query and the REFRESH TABLE statement cannot be used. When the CREATE TABLE statement is completed, the table is no longer considered a summary table.

>> The columns of the table are defined based on the definitions of the columns that result from the fullselect. If the fullselect references a single table in the FROM clause, select list items that are columns of that table are defined using the column name, data type, nullability characteristic and column default value of the referenced table.

> **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 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. Summary tables defined with this attribute do not allow INSERT, UPDATE or DELETE statements (SQLSTATE 42807).

>> **IMMEDIATE**
>>> The changes made to the underlying tables as part of a DELETE, INSERT, or UPDATE are cascaded to the summary table. In this case, the content of the table, at any point-in-time, is the same as if the specified *subselect* is processed. Summary tables defined with this attribute do not allow INSERT, UPDATE, or DELETE statements (SQLSTATE 42807).

**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. A typed table or typed view cannot be specified.

> The use of LIKE is an implicit definition of $n$ columns, where $n$ is the number of columns in the identified table, view or nickname. The implicit definition includes the column name, data type, nullability characteristic and column default value of the identified table, view or nickname.

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, or indexes. 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.

Specification of this clause is the same as specifying a *summary-table-definition* of the form:

```
CREATE TABLE table-name AS
    (SELECT * FROM table-name1) DEFINITION ONLY
```

*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.

A table may have the following:

- a 4K page size with maximum of 500 columns where the byte counts of the columns must not be greater than 4005 in a 4K page size. Refer to "Row Size" on page 675 for more details.
- an 8K page size with maximum of 1 012 columns where the byte counts of the columns must not be greater than 8101. Refer to "Row Size" on page 675 for more details.
- an 16K page size with maximum of 1 012 columns where the byte counts of the columns must not be greater than 16 293.
- an 32K page size with maximum of 1 012 columns where the byte counts of the columns must not be greater than 32 677.

*data-type*

Is one of the types in the following list. Use:

**SMALLINT**

For a small integer.

**INTEGER** or **INT**

For a large integer.

**BIGINT**

For a big integer.

**FLOAT(***integer***)**

For a single or double precision floating-point number, depending on the value of the *integer*. The value of the integer must be in the range 1 through 53. The values 1 through 24 indicate single precision and the values 25 through 53 indicate double precision.

You can also 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. |

**DECIMAL***(precision-integer, scale-integer)* or **DEC***(precision-integer, scale-integer)*

For a decimal number. The first integer is the precision of the number; that is, the total number of digits; it may range from 1 to 31. The second integer is the scale of the number; that is, the number of digits to the right of the decimal point; it may range from 0 to the precision of the number.

If precision and scale are not specified, the default values of 5,0 are used. The words **NUMERIC** and **NUM** can be used as synonyms for **DECIMAL** and **DEC**.

**CHARACTER***(integer)* or **CHAR***(integer)* or **CHARACTER** or **CHAR**

For a fixed-length character string of length *integer*, which may range from 1 to 254. If the length specification is omitted, a length of 1 character is assumed.

**VARCHAR***(integer)*, or **CHARACTER VARYING***(integer)*, or **CHAR VARYING***(integer)*

For a varying-length character string of maximum length *integer*, which may range from 1 to 32 672.

**LONG VARCHAR**

For a varying-length character string with a maximum length of 32700.

**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.

**BLOB***(integer [K | M | G])*

For a binary large object string of the specified maximum length in bytes.

The length may be in the range of 1 byte to 2 147 483 647 bytes.

If *integer* by itself is specified, that is the maximum length.

> If *integer K* (in either upper or lower case) is specified, the maximum length is 1 024 times *integer*. The maximum value for *integer* is 2 097 152.
>
> If *integer M* is specified, the maximum length is 1 048 576 times *integer*. The maximum value for *integer* is 2 048.
>
> If *integer G* is specified, the maximum length is 1 073 741 824 times *integer*. The maximum value for *integer* is 2.
>
> To create BLOB strings greater than 1 gigabyte, you must specify the NOT LOGGED option.
>
> Any number of spaces is allowed between the integer and K, M, or G. Also, no space is required. For example, all the following are valid.
>
> ```
> BLOB(50K)     BLOB(50 K)     BLOB (50   K)
> ```

**CLOB***(integer [K | M | G])* [70]
> For a character large object string of the specified maximum length in bytes.
>
> The meaning of the *integer K | M | G* is the same as for BLOB.
>
> To create CLOB strings greater than 1 gigabyte, you must specify the NOT LOGGED option.

**DBCLOB***(integer [K | M | G])*
> For a double-byte character large object string of the specified maximum length in double-byte characters.
>
> The meaning of the *integer K | M | G* is similar to that for BLOB. The differences are that the number specified is the number of double-byte characters and that the maximum size is 1 073 741 823 double-byte characters.
>
> To create DBCLOB strings greater than 1 gigabyte, you must specify the NOT LOGGED option.

**GRAPHIC***(integer)*
> For a fixed-length graphic string of length *integer* which may range from 1 to 127. If the length specification is omitted, a length of 1 is assumed.

**VARGRAPHIC***(integer)*
> For a varying-length graphic string of maximum length *integer*, which may range from 1 to 16 336.

---

70. Observe that it is not possible to specify the FOR BIT DATA clause for CLOB columns. However, a CHAR FOR BIT DATA string can be assigned to a CLOB column and a CHAR FOR BIT DATA string can be concatenated with a CLOB string.

**LONG VARGRAPHIC**
For a varying-length graphic string with a maximum length of
16 350.

**DATE**
For a date.

**TIME**
For a time.

**TIMESTAMP**
For a timestamp.

**DATALINK** or **DATALINK***(integer)*
For a link to data stored outside the database.

The column in the table consists of "anchor values" that contain
the reference information that is required to establish and
maintain the link to the external data as well as an optional
comment.

The length of a DATALINK column is 200 bytes. If *integer* is
specified, it must be 200. If the length specification is omitted, a
length of 200 bytes is assumed.

A DATALINK value is an encapsulated value with a set of built-in
scalar functions. There is a function called DLVALUE to create a
DATALINK value. The following functions can be used to extract
attributes from a DATALINK value.
- DLCOMMENT
- DLLINKTYPE
- DLURLCOMPLETE
- DLURLPATH
- DLURLPATHONLY
- DLURLSCHEME
- DLURLSERVER

A DATALINK column has the following restrictions:
- The column cannot be part of any index. 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 42830).
- A default value (WITH DEFAULT) cannot be specified for the
  column. If the column is nullable, the default for the column is
  NULL (SQLSTATE 42894).

*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.

If a column defined 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.

**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 data type VARCHAR(16) FOR BIT DATA.

*column-options*
Defines additional options 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 provided by the WITH DEFAULT clause.

*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 Table 19 on page 454.

If a column is defined as a DATALINK, then a default value cannot be specified (SQLSTATE 42613). The only possible default is NULL.

If the column is based on an attribute of a structured type, 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.

Omission of DEFAULT from a *column-definition* results in the use of the null value as the default for the column.

*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 as described in Chapter 3
- not be a floating-point constant unless the column is defined with a floating-point data type
- not have non-zero 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 characters 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 or UPDATE 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**
Specifies the value of the USER special register at the time of INSERT or UPDATE as the default for the column. If USER is specified, the data type of the column must be a character string with a length not less than the length attribute of USER.

**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 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. An example using the *cast-function* is given in Example 8 on page 469.

*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**

Specifies the USER special register. 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.

If the value specified is not valid, an error (SQLSTATE 42894) is raised.

*lob-options*

Specifies options for LOB data types.

**LOGGED**

Specifies that changes 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.

LOBs greater than 1 gigabyte cannot be logged (SQLSTATE 42993) and LOBs greater than 10 megabytes should probably not be logged.

**NOT LOGGED**

Specifies that changes made to the column are not to be logged.

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 or not the LOB value is logged. The implication of not logging is that during a roll forward 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 roll forward. During crash recovery, all committed changes and changes rolled back will reflect the expected results. See the *Administration Guide* for the implications of not logging LOB columns.

**COMPACT**

Specifies that the values in the LOB column should take up minimal disk space (free any extra disk pages in the last group used by the LOB value), rather than leave any left-over space at the end of the LOB storage area that might facilitate subsequent append operations. Note that storing data in this way may cause a performance penalty in any 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.

*datalink-options*

Specifies the options associated with a DATALINK data type.

**LINKTYPE URL**
This defines the type of link as a Uniform Resource Locator (URL).

**NO LINK CONTROL**
Specifies that there will not be any check made to determine that the file exists. Only the syntax of the URL will be checked. There is no database manager control over the file.

**FILE LINK CONTROL**
Specifies that a check should be made for the existence of the file. Additional options may be used to give the database manager further control over the file.

**file-link-options**
Additional options to define the level of database manager control of the file link.

> **INTEGRITY**
> Specifies the level of integrity of the link between a DATALINK value and the actual file.
>
> > **ALL**
> > Any file specified as a DATALINK value is under the control of the database manager and may NOT be deleted or renamed using standard file system programming interfaces.
>
> **READ PERMISSION**
> Specifies how permission to read the file specified in a DATALINK value is determined.
>
> > **FS** The read access permission is determined by the file system permissions. Such files can be accessed without retrieving the file name from the column.
>
> > **DB**
> > The read access permission is determined by the database. Access to the file will only be allowed by passing a valid file access token, returned on retrieval of the DATALINK value from the table, in the open operation.
>
> **WRITE PERMISSION**
> Specifies how permission to write to the file specified in a DATALINK value is determined.
>
> > **FS** The write access permission is determined by the file system permissions. Such files can be accessed without retrieving the file name from the column.

**BLOCKED**

Write access is blocked. The file cannot be directly updated through any interface. An alternative mechanism must be used to cause updates to the information. For example, the file is copied, the copy updated, and then the DATALINK value updated to point to the new copy of the file.

**RECOVERY**

Specifies whether or not DB2 will support point in time recovery of files referenced by values in this column.

**YES**

DB2 will support point in time recovery of files referenced by values in this column. This value can only be specified when INTEGRITY ALL and WRITE PERMISSION BLOCKED are also specified.

**NO**

Specifies that point in time recovery will not be supported.

**ON UNLINK**

Specifies the action taken on a file when a DATALINK value is changed or deleted (unlinked). Note that this is not applicable when WRITE PERMISSION FS is used.

**RESTORE**

Specifies that when a file is unlinked, the DataLink File Manager will attempt to return the file to the owner with the permissions that existed at the time the file was linked. In the case where the user is no longer registered with the file server, the result is product-specific.[71]This can only be specified when INTEGRITY ALL and WRITE PERMISSION BLOCKED are also specified.

**DELETE**

Specifies that the file will be deleted when it is unlinked. This can only be specified when READ PERMISSION DB and WRITE PERMISSION BLOCKED are also specified.

---

71. With DB2 Universal Database, the file is assigned to a special predefined "dfmunknown" user id.

**MODE DB2OPTIONS**

This mode defines a set of default file link options. The defaults defined by DB2OPTIONS are:

- INTEGRITY ALL
- READ PERMISSION FS
- WRITE PERMISSION FS
- RECOVERY NO

ON UNLINK is not applicable since WRITE PERMISSION FS is used.

**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 TABLE statement to allow the target table to be defined, usually in the case of 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 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 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 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-character identifier unique within the identifiers of the existing constraints defined on the table, is generated[72]by the system.

When used with a PRIMARY KEY or UNIQUE constraint, the *constraint-name* may 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 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) since the primary key is inherited from the supertable.

See PRIMARY KEY within the description of the *unique-constraint* below.

**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.

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 description of the *unique-constraint* below.

*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* under *referential-constraint* below.

---

72. The identifier is formed of "SQL" followed by a sequence of 15 numeric characters generated by a timestamp-based function.

**CHECK (***check-condition***)**
> This provides a shorthand method of defining a check constraint that applies to a single column. See CHECK (*check-condition*) below.

*unique-constraint*
> Defines a unique or primary key constraint. If the table has a partitioning key, then any unique or primary key must be a superset of the partitioning key. A unique or primary key constraint cannot be specified for a table that is a subtable (SQLSTATE 429B3). 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. See page 660.

**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 table and the same column must not be identified more than once. The number of identified columns must not exceed 16 and the sum of their length attributes must not exceed 1 024. The length of any individual column must not exceed 255 bytes. No LOB, LONG VARCHAR, LONG VARGRAPHIC, or DATALINK column may be used as part of a unique key (even if the length attribute of the column is small enough to fit within the 255 byte limit) (SQLSTATE 42962).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). [73]

> A unique constraint cannot be specified if the table is a subtable (SQLSTATE 429B3) since unique constraints are inherited from the supertable.

> The description of the table as recorded in the catalog includes the unique key and its unique index. A unique index will automatically be created for the columns in the sequence specified with ascending order for each column. The name of the index will be 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 will be SQL, followed by a character timestamp (*yymmddhhmmssxxx*), with SYSIBM as the schema name.

---

73. If LANGLEVEL is SQL92E or MIA then an error is returned, SQLSTATE 42891.

**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 table and the same column must not be identified more than once. The number of identified columns must not exceed 16 and the sum of their length attributes must not exceed 1 024. The length of any individual column must not exceed 255 bytes. No LOB, LONG VARCHAR, LONG VARGRAPHIC, or DATALINK column may be used as part of a primary key (even if the length attribute of the column is small enough to fit within the 255 byte limit) (SQLSTATE 42962). The set of columns in the primary key cannot be the same as the set of columns of a unique key (SQLSTATE 01543). [73]

Only one primary key can be defined on a table.

A primary key cannot be specified if the table is a subtable (SQLSTATE 429B3) since the primary key is inherited from the supertable.

The description of the table as recorded in the catalog includes the primary key and its primary index. A unique index will automatically be created for the columns in the sequence specified with ascending order for each column. The name of the index will be 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 will be SQL, followed by a character timestamp (*yymmddhhmmssxxx*), with SYSIBM as the schema name.

If the table has a partitioning key, the columns of a *unique-constraint* must be a superset of the partitioning key columns; column order is unimportant.

*referential-constraint*

Defines a referential constraint.

**CONSTRAINT** *constraint-name*

Names the referential constraint. See page 660.

**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 16 and the sum of their length attributes must not exceed 1 024 minus the number of columns that allow null values. The length of any individual column must not exceed 255 bytes. No LOB, LONG VARCHAR, LONG VARGRAPHIC or DATALINK column may 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 and parent key for the referential constraint.

**REFERENCES** *table-name*
The table specified in a REFERENCES clause must identify a base table 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 are the same as the foreign key, parent key and parent table of a previously specified referential constraint. Duplicate referential constraints are ignored and a warning is issued (SQLSTATE 01543).

In the following discussion, let T2 denote the identified parent table and let T1 denote the table being created[74](T1 and T2 may 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.

---

74. or altered, in the case where this clause is referenced from the description of the ALTER TABLE statement.

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 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.

A cycle involving two or more tables must not cause a table to be delete-connected to itself unless all of the delete rules in the cycle are CASCADE. Thus, if the new relationship would form a cycle and T2 is already

delete connected to T1, then the constraint can only be defined if it has a delete rule of CASCADE and all other delete rules of the cycle are CASCADE.

If T1 is delete-connected to T2 through multiple paths, those relationships in which T1 is a dependent and which form all or part of those paths must have the same delete rule and it must not be SET NULL. The NO ACTION and RESTRICT actions are treated identically. Thus, if T1 is a dependent of T3 in a relationship with a delete rule of *r*, the referential constraint cannot be defined when *r* is SET NULL if any of these conditions exist:

- T2 and T3 are the same table
- T2 is a descendant of T3 and the deletion of rows from T3 cascades to T2
- T3 is a descendant of T2 and the deletion of rows from T2 cascades to T3
- T2 and T3 are both descendants of the same table and the deletion of rows from that table cascades to both T2 and T3.

If *r* is other than SET NULL, the referential constraint can be defined, but the delete rule that is implicitly or explicitly specified in the FOREIGN KEY clause must be the same as *r*.

**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 under CREATE TABLE in "Notes" on page 672.

*check-constraint*
Defines a check constraint. A *check-constraint* is a *search-condition* that must evaluate to not false.

**CONSTRAINT** *constraint-name*
Names the check constraint. See page 660.

**CHECK (***check-condition***)**
Defines a check constraint. A *check-condition* is a *search-condition* except as follows:

- A column reference must be to a column of the table being created
- It cannot contain any of the following:
  - subqueries
  - dereference operations or DEREF functions
  - column functions
  - variant user-defined functions
  - user-defined functions using the EXTERNAL ACTION option
  - user-defined functions using the SCRATCHPAD option
  - host variables
  - parameter markers
  - special registers
  - an alias

If a check constraint is specified as part of a *column-definition* then a column reference can only be made to the same column. Check constraints specified as part of a table definition can have column references identifying columns 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 check-condition "IS NOT NULL" can be specified, however it is recommended that nullability be enforced directly 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 are enforced when rows in the table are inserted or updated. A check constraint defined on a table automatically applies to all subtables of that 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).

**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 42613).

If the table is a typed table, then this option is not supported (SQLSTATE 428DH or 42HDR).

**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 defined to allow data on a partition other than the catalog partition (multiple partition nodegroup or nodegroup with a partition other than the catalog partition), then this option is not supported (SQLSTATE 42997).

Further information about using replication can be found in the *Administration Guide* and the *Replication Guide and Reference*.

**IN** *tablespace-name1*
Identifies the table space in which the table will be created. The table space must exist, and be a REGULAR table space. If no other table space is specified, then all table parts will be stored in this table space. This clause cannot be specified when creating a subtable (SQLSTATE 42613), since the table space is inherited from the root table of the table hierarchy. If this clause is not specified, a table space for the table is determined as follows:

```
IF table space IBMDEFAULTGROUP exists with sufficient page size
THEN use it
ELSE IF user created table spaces exists then choose one
which is of the smallest pagesize that is sufficient for this table
THEN use it
ELSE IF table space USERSPACE1 exists with sufficient page size
THEN use it
ELSE issue an error (SQLSTATE 42727).
```

The sufficient page size of a table is determined by either the byte count of the row or the number of columns. Refer to "Row Size" on page 675 for more information.

*tablespace-options:*
Specifies the table space in which indexes and/or long column values will be stored. See "CREATE TABLESPACE" on page 681 for details on types of table spaces.

**INDEX IN** *tablespace-name2*
Identifies the table space in which any indexes on the table will be created. This option is allowed only when the primary table space specified in the IN clause is a DMS table space. The specified table space must exist, be a REGULAR DMS table space and must be in the same nodegroup as *tablespace-name1* (SQLSTATE 42838).

Note that specifying which table space will contain a table's index can only be done when the table is created.

**LONG IN** *tablespace-name3*
Identifies the table space in which the values of any long columns (LONG VARCHAR, LONG VARGRAPHIC, LOB data types, or distinct types with any of these as source types) will

be stored. This option is allowed only when the primary table space specified in the IN clause is a DMS table space. The table space must exist, be a LONG DMS table space and must be in the same nodegroup of *tablspace-name1* (SQLSTATE 42838).

**PARTITIONING KEY (***column-name***,...)**
Specifies the partitioning key used when data in the table is partitioned. Each *column-name* must identify a column of the table and the same column must not be identified more than once. No LOB, LONG VARCHAR, LONG VARGRAPHIC or DATALINK column may be used as part of a partitioning key (SQLSTATE 42962). A partitioning key cannot be specified for a table that is a subtable (SQLSTATE 42613), since the partitioning key is inherited from the root table in the table hierarchy.

If this clause is not specified, and this table resides in a multiple partition nodegroup, then the partitioning key is defined as follows:

- if the table is a typed table, the object identifier column
- if a primary key is specified, the first column of the primary key is the partitioning key
- otherwise, the first non-long column (LOB or long column type) is the partitioning key.

If none of the columns satisfy the requirement of the default partitioning key, the table is created without one. Such tables are allowed only in table spaces defined on single-partition nodegroups.

For tables in table spaces defined on single-partition nodegroups, any collection of non-long type columns can be used to define the partitioning key. If you do not specify this parameter, no partitioning key is created.

For restrictions related to the partitioning key, see "Rules" on page 671.

**USING HASHING**
Specifies the use of the hashing function as the partitioning method for data distribution. This is the only partitioning method supported.

**REPLICATED**
Specifies that the data stored in the table is physically replicated on each database partition of the nodegroup of the table space in which the table is defined. This means that a copy of all the data

in the table exists on each of these database partitions. This option can only be specified for a summary table defined with REFRESH DEFERRED (SQLSTATE 42997).

**NOT LOGGED INITIALLY**

Any changes 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. See "Notes" on page 672 for other considerations when using this option.

All catalog changes and storage related information are logged, as are all operations that are done on the table in subsequent units of work.

A foreign key constraint cannot be defined on a table that references a parent with the NOT LOGGED INITIALLY attribute. This clause cannot be specified when creating a subtable (SQLSTATE 42613).

**Note:** An error in any operation in the unit of work in which the table is created will result in the entire unit of work being rolled back.

## Rules

- The sum of the byte counts of the columns must not be greater than the row size limit that is based on the page size of the table space (SQLSTATE 54010). Refer to "Byte Counts" on page 676 and Table 30 on page 921 for more information. 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 (additional subtable columns must be considered nullable for byte count purposes, even when 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.
- An object identifier column of a typed table cannot be updated (SQLSTATE 42808).
- A partitioning key column of a table cannot be updated (SQLSTATE 42997).
- Any unique or primary key constraint defined on the table must be a superset of the partitioning key (SQLSTATE 42997).
- A nullable column of a partitioning key cannot be included as a foreign key column when the relationship is defined with ON DELETE SET NULL (SQLSTATE 42997).

## CREATE TABLE

- The following table provides the supported combinations of DATALINK options in the *file-link-options* (SQLSTATE 42613).

*Table 21. Valid DATALINK File Control Option Combinations*

| INTEGRITY | READ PERMISSION | WRITE PERMISSION | RECOVERY | ON UNLINK |
|-----------|-----------------|------------------|----------|----------------|
| ALL | FS | FS | NO | Not applicable |
| ALL | FS | BLOCKED | NO | RESTORE |
| ALL | FS | BLOCKED | YES | RESTORE |
| ALL | DB | BLOCKED | NO | RESTORE |
| ALL | DB | BLOCKED | NO | DELETE |
| ALL | DB | BLOCKED | YES | RESTORE |
| ALL | DB | BLOCKED | YES | DELETE |

The following rules only apply to partitioned databases.

- Tables composed of only long columns can only be created in table spaces defined on single-partition nodegroups.
- The partitioning key definition of a table in a table space defined on a multiple partition nodegroup cannot be altered.
- The partitioning key column of a typed table must be the OID column.

### 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. A delete or update rule of RESTRICT is enforced before all other constraints including those referential constraints with modifying rules such as CASCADE or SET NULL. A delete or update rule of NO ACTION is enforced after other referential constraints. There are very few cases where

this can make a difference during a delete or update. One example where different behavior is evident involves a DELETE of rows in 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 delete rule of RESTRICT, a restrict violation will be raised (SQLSTATE 23001) if there are any child rows for parent keys of T1 in T3.

If table T1 is a parent of table T3 with delete rule of NO ACTION, the child rows may be deleted by the delete rule of CASCADE when deleting rows from T2 before the NO ACTION delete rule is enforced for the deletes from T1. If deletes from T2 did not result in deleting all child rows for parent keys of T1 in T3, then a constraint violation will be 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 defined on multiple partition nodegroups, table collocation should be considered in choosing the partitioning keys. Following is a list of items to consider:
  - The tables must be in the same nodegroup for collocation. The table spaces may be different, but must be defined in the same nodegroup.
  - The partitioning keys of the tables must have the same number of columns, and the corresponding key columns must be partition compatible for collocation. For more information, see "Partition Compatibility" on page 110.
  - The choice of partitioning key also has an impact on performance of joins. If a table is frequently joined with another table, you should consider the joining column(s) as a partitioning key for both tables.
- The NOT LOGGED INITIALLY clause can not be used when DATALINK columns with the FILE LINK CONTROL attribute are present in the table (SQLSTATE 42613) .
- The NOT LOGGED INITIALLY option is useful for situations where a large result set needs to be created with data from an alternate 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 permitted is to drop the table.
    - Once such a table is backed up as part of a database or table space back up, recovery of the table becomes possible.
- A REFRESH DEFERRED summary table may be used to optimize the processing of queries if CURRENT REFRESH AGE is set to ANY. A REFRESH IMMEDIATE summary table is always considered for optimization. In order for this optimization be able to use a REFRESH DEFERRED or REFRESH IMMEDIATE summary table, the fullselect must conform to certain rules in addition to those already described. The fullselect must:
    - be a subselect with a GROUP BY clause or a subselect with a single table reference
    - not include DISTINCT anywhere in the select list
    - not include any special registers
    - not include functions that are not deterministic.

  If the query specified when creating a summary table does not conform to these rules, a warning is returned (SQLSTATE 01633).
- If a summary table is defined with REFRESH IMMEDIATE, it is possible for an error to occur when attempting to apply the change resulting from an insert, update or delete of an underlying table. The error will cause the failure of the insert, update or delete of the underlying table.
- *Inoperative summary tables:* An inoperative summary table is a table that is no longer available for SQL statements. A summary table becomes inoperative if:
    - A privilege upon which the summary table definition is dependent is revoked.
    - An object such as a table, alias or function, upon which the summary table definition is dependent is dropped.

  In practical terms, an inoperative summary table is one in which the summary table definition has been unintentionally dropped. For example, when an alias is dropped, any summary table defined using that alias is made inoperative. All packages dependent on the summary table are no longer valid.

Until the inoperative summary table is explicitly recreated or dropped, a statement using that inoperative summary table cannot be compiled (SQLSTATE 51024) with the exception of the CREATE ALIAS, CREATE TABLE, DROP TABLE, and COMMENT ON TABLE statements. Until the inoperative summary table has been explicitly dropped, its qualified name cannot be used to create another view, base table or alias. (SQLSTATE 42710).

An inoperative summary table may be recreated by issuing a CREATE TABLE statement using the definition text of the inoperative summary table. This summary table query text is stored in the TEXT column of the SYSCAT.VIEWS catalog. When recreating an inoperative summary table, it is necessary to explicitly grant any privileges required on that table by others, due to the fact that all authorization records on a summary table are deleted if the summary table is marked inoperative. Note that there is no need to explicitly drop the inoperative summary table in order to recreate it. Issuing a CREATE TABLE statement that defines a summary table with the same *table-name* as an inoperative summary table will cause that inoperative summary table to be replaced, and the CREATE TABLE statement will return a warning (SQLSTATE 01595).

Inoperative summary tables 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:* 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 the immediate supertable is automatically granted on the subtable with the table definer as the grantor.
- *Row Size:* The maximum number of bytes allowed in the row of a table is dependent on the page size of the the table space in which the table is created (*tablspace-name1*). The following list shows the row size limit and number of columns limit associated with each table space page size.

*Table 22. Limits for Number of Columns and Row Size in Each Table Space Page Size*

| Page Size | Row Size Limit | Column Count Limit |
|-----------|----------------|--------------------|
| 4K        | 4 005          | 500                |
| 8K        | 8 101          | 1 012              |
| 16K       | 16 293         | 1 012              |
| 32K       | 32 677         | 1 012              |

The actual number of columns for a table may be further limited by the following formula:

Total Columns * 8 + Number of LOB Columns * 12 + Number of Datalink Columns * 28 <= row size limit for page size.

- *Byte Counts:* The following list contains the byte counts of columns by data type for columns that do not allow null values. For a column that allows null values the byte count is one more than shown in the list.

  If the table is created based on a structured type, an additional 4 bytes of overhead is reserved to identify rows of subtables regardless of whether or not subtables are defined. Also, additional subtable columns must be considered nullable for byte count purposes, even when defined as not nullable.

| Data type | Byte count |
|---|---|
| **INTEGER** | 4 |
| **SMALLINT** | 2 |
| **BIGINT** | 8 |
| **REAL** | 4 |
| **DOUBLE** | 8 |
| **DECIMAL** | The integral part of $(p/2)+1$, where $p$ is the precision. |
| **CHAR($n$)** | $n$ |
| **VARCHAR($n$)** | $n+4$ |
| **LONG VARCHAR** | 24 |
| **GRAPHIC($n$)** | $n*2$ |
| **VARGRAPHIC($n$)** | $(n*2)+4$ |
| **LONG VARGRAPHIC** | 24 |
| **DATE** | 4 |
| **TIME** | 3 |
| **TIMESTAMP** | 10 |
| **DATALINK($n$)** | $n+54$ |
| **LOB types** | 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. The following table shows typical sizes: |

| Maximum LOB Length | LOB Descriptor Size |
|---|---|
| 1 024 | 72 |
| 8 192 | 96 |
| 65 536 | 120 |
| 524 000 | 144 |

|  |  |
|---|---|
| 4 190 000 | 168 |
| 134 000 000 | 200 |
| 536 000 000 | 224 |
| 1 070 000 000 | 256 |
| 1 470 000 000 | 280 |
| 2 147 483 647 | 316 |

| | |
|---|---|
| **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. |

## Examples

*Example 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
```

*Example 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
```

**CREATE TABLE**

*Example 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 descirbed for the *IN tablespace-name1* 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)
```

*Example 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 DISTINCT TYPE JOHNDOE.T_SALARY AS INTEGER WITH COMPARISONS

CREATE DISTINCT TYPE JOHNDOE.MILES AS FLOAT WITH COMPARISONS

CREATE TABLE EMPLOYEE
  (ID          INTEGER NOT NULL,
   NAME        CHAR (30),
   SALARY      T_SALARY NOT NULL WITH DEFAULT,
   LIVING_DIST MILES    DEFAULT MILES(1) )
```

*Example 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 descirbed for the *IN tablespace-name1* clause. Assume the CURRENT PATH is the default.

```
CREATE DISTINCT TYPE IMAGE AS BLOB (10M)

CREATE DISTINCT TYPE AUDIO AS BLOB (1G)

CREATE TABLE PERSON
  (SSN   INTEGER NOT NULL,
   NAME  CHAR (30),
   VOICE AUDIO,
   PHOTO IMAGE)
```

*Example 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
```

*Example 7:* Create a table that is wholly contained in the PAYROLL table space.

```
CREATE TABLE EMPLOYEE .....
   IN PAYROLL
```

*Example 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
```

*Example 9:* Create a table and log SQL changes in the default format.

```
CREATE TABLE SALARY1 .....
```

or

```
CREATE TABLE SALARY1 .....
   DATA CAPTURE NONE
```

*Example 10:* Create a table and log SQL changes in an expanded format.

```
CREATE TABLE SALARY2 .....
   DATA CAPTURE CHANGES
```

*Example 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.

*Example 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 place, and a description. When available, it will also point to places where newspaper articles about the game are stored and where still and moving pictures of the goal are stored. The newspaper articles are to be linked so they cannot be deleted or renamed but all existing display and update applications must continue to operate. The still pictures and movies are to be linked with access under complete control of DB2. The still pictures are to have recovery and are to be returned to their original owner if unlinked. The movie pictures are not to have recovery and are to be deleted if unlinked. The description column and the three DATALINK columns are 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),
    ARTICLES        DATALINK  LINKTYPE URL FILE LINK CONTROL MODE DB2OPTIONS,
    SNAPSHOT        DATALINK  LINKTYPE URL FILE LINK CONTROL
                              INTEGRITY ALL
                              READ PERMISSION DB WRITE PERMISSION BLOCKED
                              RECOVERY YES ON UNLINK RESTORE,
    MOVIE           DATALINK  LINKTYPE URL FILE LINK CONTROL
                              INTEGRITY ALL
                              READ PERMISSION DB WRITE PERMISSION BLOCKED
                              RECOVERY NO ON UNLINK DELETE )
```

*Example 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
   ) DEFINITION ONLY
```

## CREATE TABLESPACE

The CREATE TABLESPACE statement creates 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. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

### Authorization

The authorization ID of the statement must have SYSCTRL or SYSADM authority.

### Syntax
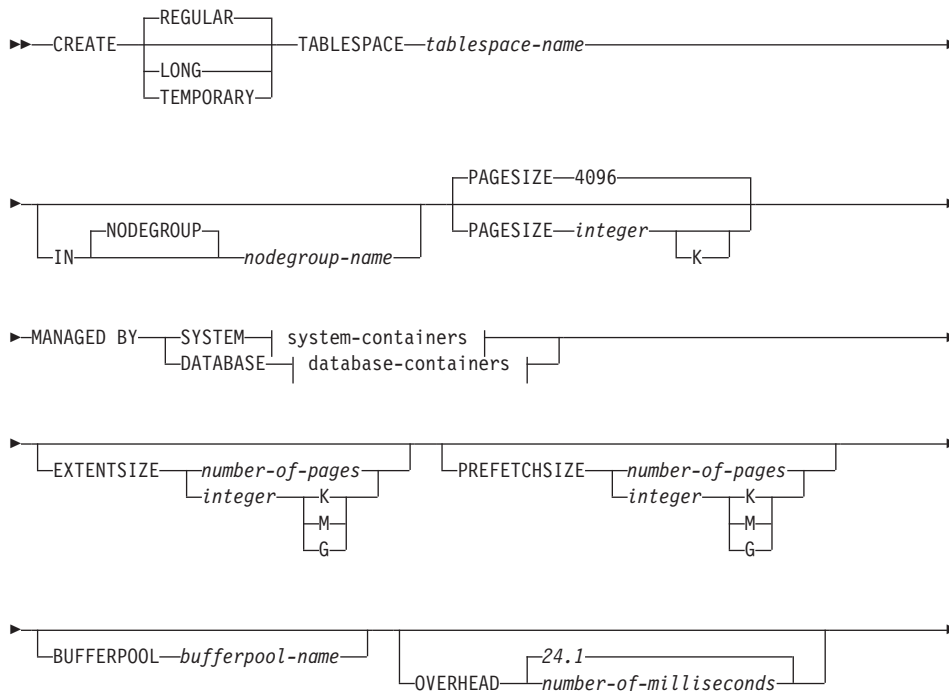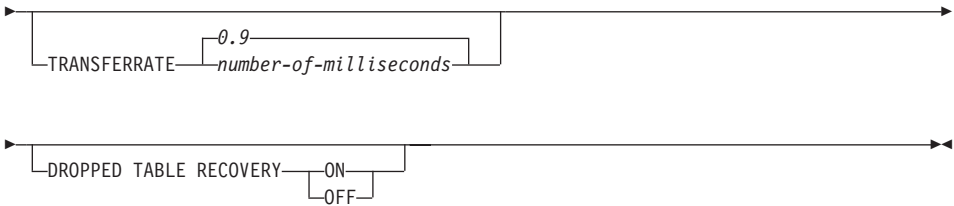
```
                    ┌─REGULAR──┐
►►─CREATE───────────┼─LONG─────┼─TABLESPACE──tablespace-name──────────────►
                    └─TEMPORARY┘


                                          ┌─PAGESIZE──4096─────────┐
►──┬──────────────────────────────────┬──┼────────────────────────┼──────►
   │   ┌─NODEGROUP─┐                   │  └─PAGESIZE──integer───────┘
   └─IN─┴───────────┴──nodegroup-name──┘                    └─K─┘


►─MANAGED BY──┬─SYSTEM────│ system-containers  ├──────────────────────────►
              └─DATABASE──┤ database-containers ├


►──┬───────────────────────────────┬──┬──────────────────────────────────┬─►
   └─EXTENTSIZE──┬─number-of-pages─┬┘  └─PREFETCHSIZE──┬─number-of-pages─┬┘
                 └─integer──┬───┬──┘                    └─integer──┬───┬──┘
                            ├─K─┤                                  ├─K─┤
                            ├─M─┤                                  ├─M─┤
                            └─G─┘                                  └─G─┘


►──┬────────────────────────────────┬──┬────────────────────────────┬────►
   └─BUFFERPOOL──bufferpool-name─────┘  │              ┌─24.1──────────────────┐ │
                                        └─OVERHEAD──┴─number-of-milliseconds─┘
```

## CREATE TABLESPACE

```
                 ┌─0.9──────────────────┐
►──┬─────────────────────────────────────┬──────────────────────────►
   └─TRANSFERRATE─┬─0.9──────────────────┬─┘
                  └─number-of-milliseconds─┘
```

```
►──┬──────────────────────────────────────┬──────────────────────►◄
   └─DROPPED TABLE RECOVERY─┬─ON──┬────────┘
                            └─OFF─┘
```

**system-containers:**

```
   ┌──────────────────────────────────────────────┐
   │            ┌─,──────────────┐                  │
├──▼─USING─(──▼─'container-string'─┬──)──┬─────────────────┬─┤
                                          └─on-nodes-clause─┘
```

**database-containers:**

```
   ┌────────────────────────────────────┐
├──▼─USING─┤ container-clause ├─┬─────────────────┬─┤
                               └─on-nodes-clause─┘
```

**container-clause:**

```
        ┌─,────────────────────────────────────────────────┐
├──(──▼─┬─FILE───┬──'container-string'─┬─number-of-pages──────┬──)─┤
        └─DEVICE─┘                     └─integer─┬─K─┬────────┘
                                                 ├─M─┤
                                                 └─G─┘
```

**on-nodes-clause:**

```
                      ┌─,──────────────────────────┐
├──ON─┬─NODE──┬──(──▼─node-number1─┬──────────────────────┬──)─┤
      └─NODES─┘                    └─TO─node-number2──────┘
```
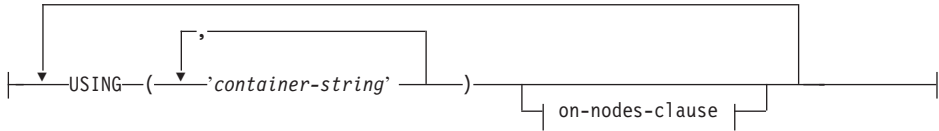
## Description

**REGULAR**
> Stores all data except for temporary tables.

682

**LONG**

Stores long or LOB table columns. The table space must be a DMS table space.

**TEMPORARY**

Stores temporary tables. (Temporary tables are work areas used by the database manager to perform operations such as sorts or joins.) Note that a database must always have at least one TEMPORARY table space, as temporary tables can only be stored in such a table space. A temporary table space is created automatically when a database is created.

(See CREATE DATABASE in the *Command Reference*.)

*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 NODEGROUP** *nodegroup-name*

Specifies the nodegroup for the table space. The nodegroup must exist. The only nodegroup that can be specified when creating a TEMPORARY table space is IBMTEMPGROUP. The NODEGROUP keyword is optional.

If the nodegroup is not specified, the default nodegroup (IBMDEFAULTGROUP) is used unless TEMPORARY is specified and then IBMTEMPGROUP is used.

**PAGESIZE** *integer* **[K]**

Defines the size of pages used for the table space. The valid values for *integer* without the suffix K are 4 096 or 8 192, 16 384, or 32 768. The valid values for *integer* with the suffix K are 4 or 8, 16, or 32. An error occurs if the page size is not one of these values (SQLSTATE 428DE) or the page size is not the same as the page size of the bufferpool associated with the table space (SQLSTATE 428CB). The default is 4 096 byte (4K) pages. Any number of spaces is allowed between *integer* and K, including no space.

**MANAGED BY SYSTEM**

Specifies that the table space is to be a system managed space (SMS) table space.

**system-containers**

Specify the containers for an SMS table space.

**USING** *('container-string',...)*

For a SMS table space, identifies one or more containers that will belong to the table space and into which the table space's 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. 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 exist, it must not contain any files or subdirectories (SQLSTATE 428B2).

The format of *container-string* is dependent on the operating system. The containers are specified in the normal manner for the operating system. For example, an OS/2 Windows 95 and Windows NT directory path begins with a drive letter and a ":", while on UNIX-based systems, a path begins with a "/".

Note that remote resources (such as LAN-redirected drives on OS/2, Windows 95 and Windows NT or NFS-mounted file systems on AIX) are not supported.

*on-nodes-clause*

Specifies the partition or partitions on which the containers are created in a partitioned database. If this clause or any other *on-nodes-clause* of this statement is not specified, then the containers are created on all partitions or nodes currently in the nodegroup. For a TEMPORARY table space when the clause is not specified, the containers will also be created on all new partitions or nodes added to the database. See page 685 for details on specifying this clause.

**MANAGED BY DATABASE**

Specifies that the table space is to be a database managed space (DMS) table space.

**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 into which the table space's 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, the *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 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, the *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* is dependent on the operating system. The containers will be specified in the normal manner for the operating system. For more detail on declaring containers, refer to the Administration Guide.

Remote resources (such as LAN-redirected drives on OS/2, Windows 95 and Windows NT or NFS-mounted file systems on AIX) are not supported.

*on-nodes-clause*
Specifies the partition or partitions on which the containers are created in a partitioned database. If this clause or any other *on-nodes-clause* of this statement is not specified, then the containers are created on all partitions currently in the nodegroup. For a TEMPORARY table space when the clause is not specified, the containers will also be created on all new partitions added to the database.

**on-nodes-clause**
Specifies the partitions on which containers are created in a partitioned database.

**ON NODES**
Keywords that indicate that specific partitions are specified. NODE is a synonym for NODES.

*node-number1*
Specify a specific partition (or node) number.

**TO** *node-number2*

 Specify a range of partition (or node) numbers. The value of *node-number2* must be greater than or equal to the value of *node-number1* (SQLSTATE 428A9). All partitions between and including the specified partition numbers are included in the partitions for which the containers are created if the node is included in the nodegroup of the table space.

The partition specified by number and every partition (or node) in the range of partition must exist in the nodegroup on which the table space is defined (SQLSTATE 42729). A partition-number may only appear explicitly or within a range in exactly one *on-nodes-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), 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 value for extent size. The database manager cycles repeatedly through the containers as data is stored.

The default value is provided by the DFT_EXTENT_SZ configuration parameter.

**PREFETCHSIZE** *number-of-pages*

 Specifies the number of PAGESIZE pages that will be read from the table space when data prefetching is being performed. The prefetch size value 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 value for prefetch size. Prefetching reads in data needed by a query prior to it being referenced by the query, so that the query need not wait for I/O to be performed.

The default value is provided by the DFT_PREFETCH_SZ configuration parameter. (This configuration parameter, like all configuration parameters, is explained in detail in the *Administration Guide*.)

**BUFFERPOOL** *bufferpool-name*

 The name of the buffer pool used for tables in this table space. The buffer pool must exist (SQLSTATE 42704). If not specified, the default buffer pool (IBMDEFAULTBP) is used. The page size of the bufferpool must match the page size specified (or defaulted) for the table space (SQLSTATE 428CB). The nodegroup of the table space must be defined for the bufferpool (SQLSTATE 42735).

OVERHEAD *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. This value is used to determine the cost of I/O during query optimization.

TRANSFERRATE *number-of-milliseconds*

Any numeric literal (integer, decimal, or floating point) that specifies the time to read one page 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. This value is used to determine the cost of I/O during query optimization.

DROPPED TABLE RECOVERY

Dropped tables in the specified table space may be recovered using the RECOVER TABLE ON option of the ROLLFORWARD command.

## Notes

- For information on how to determine the correct EXTENTSIZE, PREFETCHSIZE, OVERHEAD, and TRANSFERRATE values, refer to the *Administration Guide*.

- Choosing between a database-managed space or a system-managed space for a table space is a fundamental choice involving tradeoffs. See the *Administration Guide* for a discussion of those tradeoffs.

- When more than one TEMPORARY table space exists in the database, they will be used in round-robin fashion in order to balance their usage. See the *Administration Guide* for information on using more than one table space, rebalancing and recommended values for EXTENTSIZE, PREFETCHSIZE, OVERHEAD, and TRANSFERRATE.

- In a partitioned database if more than one partition resides on the same physical node, then the same device or specific path cannot be specified for such partitions (SQLSTATE 42730). For this environment, either specify a unique *container-string* for each partition or use a relative path name.

- You can specify a node expression for container string syntax when creating either SMS or DMS containers. You would typically specify the node expression if you are using multiple logical nodes in the partitioned database system. This ensures that container names are unique across nodes (database partition servers). When you specify the expression, either the node number is part of the container name, or, if you specify additional arguments, the result of the argument is part of the container name.

  You use the argument " $N" ([blank]$N) to indicate the node expression. The argument must occur at the end of the container string and can only be used in one of the following forms. In the table that follows, the node number is assumed to be 5:

**CREATE TABLESPACE**

*Table 23. Arguments for Creating Containers*

| Syntax | Example | Value |
|---|---|---|
| [blank]$N | " $N" | 5 |
| [blank]$N+[number] | " $N+1011" | 1016 |
| [blank]$N%[number] | " $N%3" | 2 |
| [blank]$N+[number]%[number] | " $N+12%13" | 4 |
| [blank]$N%[number]+[number] | " $N%3+20" | 22 |
| **Note:** | | |
| – % is modulus | | |
| – In all cases, the operators are evaluated from left to right. | | |

Some examples are as follows: *Example 1:*

```
CREATE TABLESPACE TS1 MANAGED BY DATABASE USING
    (device '/dev/rcont $N' 20000)
```

On a two-node system, the following containers would be used:

```
/dev/rcont0  - on NODE 0
/dev/rcont1  - on NODE 1
```

*Example 2:*

```
CREATE TABLESPACE TS2 MANAGED BY DATABASE USING
    (file '/DB2/containers/TS2/container $N+100' 10000)
```

On a four-node system, the following containers would be created:

```
/DB2/containers/TS2/container100  - on NODE 0
/DB2/containers/TS2/container101  - on NODE 1
/DB2/containers/TS2/container102  - on NODE 2
/DB2/containers/TS2/container103  - on NODE 3
```

*Example 3:*

```
CREATE TABLESPACE TS3 MANAGED BY SYSTEM USING
('/TS3/cont $N%2','/TS3/cont $N%2+2')
```

On a two-node system, the following containers would be created:

```
/TS3/cont0  - On NODE 0
/TS3/cont2  - On NODE 0
/TS3/cont1  - On NODE 1
/TS3/cont3  - On NODE 1
```

## Examples

*Example 1:* Create a regular DMS table space on a UNIX-based system using
3 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 24.1
   TRANSFERRATE 0.9
```

*Example 2:*  Create a regular SMS table space on OS/2 or Windows NT using 3 directories on three separate drives, with a 64-page extent size, and a 32-page prefetch size.

```
CREATE TABLESPACE ACCOUNTING
   MANAGED BY SYSTEM
   USING ('d:\acc_tbsp', 'e:\acc_tbsp', 'f:\acc_tbsp')
   EXTENTSIZE 64
   PREFETCHSIZE 32
```

*Example 3:*  Create a temporary DMS table space on Unix using 2 files of 50,000 pages each, and a 256-page extent size.

```
CREATE TEMPORARY TABLESPACE TEMPSPACE2
   MANAGED BY DATABASE
   USING (FILE '/tmp/tempspace2.f1' 50000,
      FILE '/tmp/tempspace2.f2' 50000)
   EXTENTSIZE 256
```

*Example 4:*  Create a DMS table space on nodegroup ODDNODEGROUP (nodes 1,3,5) on a Unix partitioned database. On all partitions (or nodes), use the device /dev/rhdisk0 for 10 000 4K pages. Also specify a partition specific device for each partition with 40 000 4K pages.

```
CREATE TABLESPACE PLANS
   MANAGED BY DATABASE
   USING (DEVICE '/dev/rhdisk0' 10000, DEVICE '/dev/rn1hd01' 40000)
   ON NODE (1)
   USING (DEVICE '/dev/rhdisk0' 10000, DEVICE '/dev/rn3hd03' 40000)
   ON NODE (3)
   USING (DEVICE '/dev/rhdisk0' 10000, DEVICE '/dev/rn5hd05' 40000)
   ON NODE (5)
```

## CREATE TRIGGER

The CREATE TRIGGER statement defines a trigger in the database.

### 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 when the trigger is created must include at least one of the following:
- SYSADM or DBADM authority.
- ALTER privilege on the table on which the trigger is defined, or ALTERIN privilege on the schema of the table on which the trigger is defined 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.

If the authorization ID of the statement does not have SYSADM or DBADM authority, the privileges that the authorization ID of the statement holds (without considering PUBLIC or group privileges) must include all of the following as long as the trigger exists:
- SELECT privilege on the table on which the trigger is defined, if any transition variables or tables are specified
- SELECT privilege on any table or view referenced in the triggered action condition
- Necessary privileges to invoke the triggered SQL statements specified.

If a trigger definer can only create the trigger because the definer has SYSADM authority, then the definer is granted explicit DBADM authority for the purpose of creating the trigger.

### Syntax

```
►►──CREATE TRIGGER──trigger-name──┬─NO CASCADE BEFORE─┬─────────────────────►
                                  └─AFTER─────────────┘
```

```
►──┬─INSERT──────────────────┬──ON──table-name──────────────────►
   ├─DELETE──────────────────┤
   └─UPDATE─┬──────────────┬─┘
            │      ┌─,──────┐ │
            └─OF──▼─column-name─┘
```

```
►──────────────────────────────────────────────────────────────►
   │            ┌─────────────(1)──(2)──────────────────────┐
   └─REFERENCING──▼─┬─OLD──────┬─AS─┬──correlation-name─┬───┘
                    │          └────┘                   │
                    ├─NEW──────┬─AS─┬──correlation-name─┤
                    │          └────┘                   │
                    ├─OLD_TABLE─┬─AS─┬──identifier───────┤
                    │           └────┘                   │
                    └─NEW_TABLE─┬─AS─┬──identifier───────┘
                                └────┘
```

```
►──┬─FOR EACH ROW──────────┬──MODE DB2SQL─┤ triggered-action ├──►◄
   │         (3)           │
   └──────FOR EACH STATEMENT─┘
```

**triggered-action:**

```
├──┬────────────────────────────────┬───────────────────────────►
   └─WHEN──(──search-condition──)───┘
```

```
►──┬─triggered-SQL-statement──────────────────────────┬─────────┤
   │                    ┌◄────────────────────┐        │
   └─BEGIN ATOMIC──────▼─triggered-SQL-statement──;─┬──END─┘
```
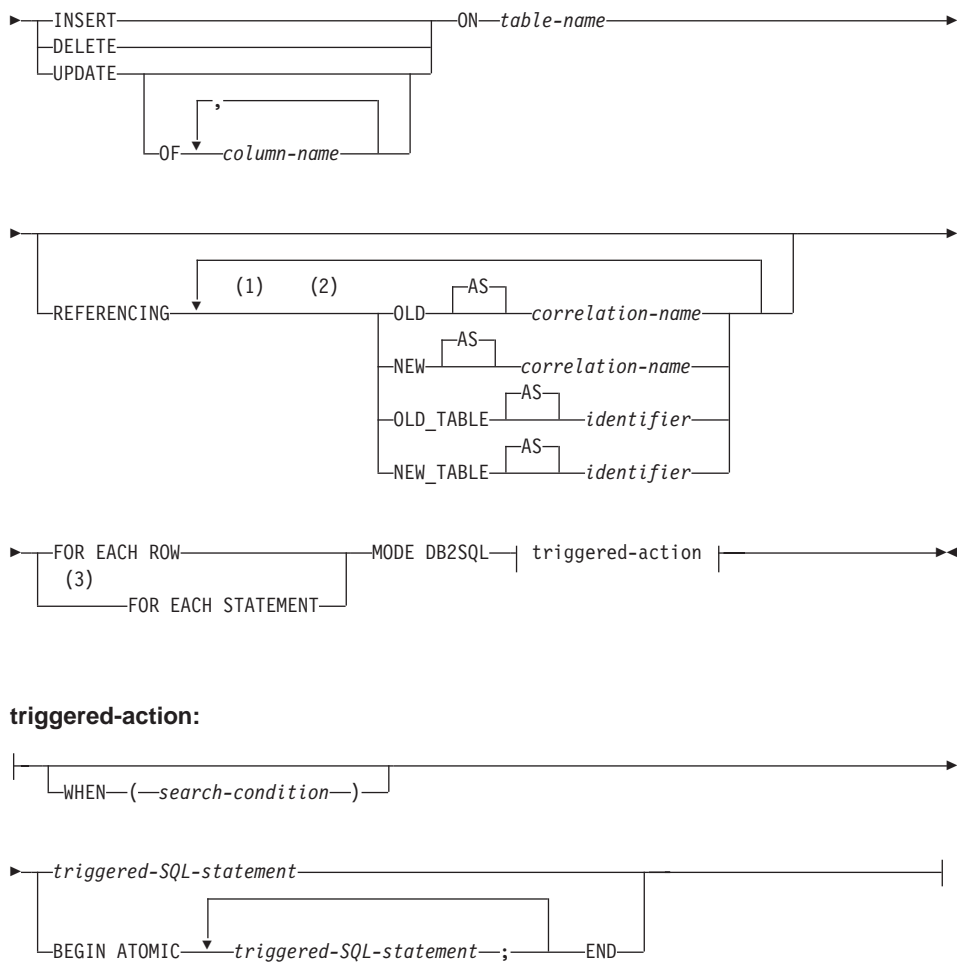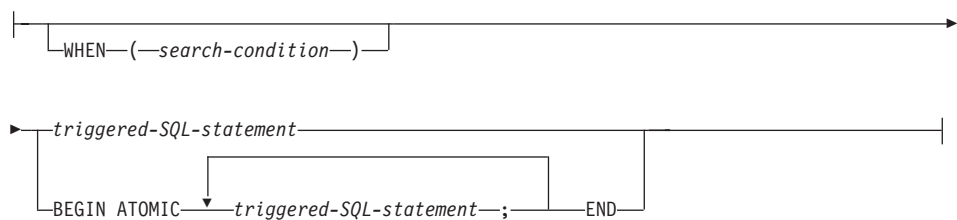
**Notes:**

1   OLD and NEW may only be specified once each.

2   OLD_TABLE and NEW_TABLE may only be specified once each and only for AFTER triggers.

3   FOR EACH STATEMENT may not be specified for BEFORE triggers.

## Description

*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).

## CREATE TRIGGER

**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.

**INSERT**
Specifies that the triggered action associated with the trigger is to be executed whenever an INSERT operation is applied to the designated base table.

**DELETE**
Specifies that the triggered action associated with the trigger is to be executed whenever a DELETE operation is applied to the designated base table.

**UPDATE**
Specifies that the triggered action associated with the trigger is to be executed whenever an UPDATE operation is applied to the designated base table subject to the columns specified or implied.

If the optional *column-name* list is not specified, every column of the table 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.

**OF** *column-name,...*
Each *column-name* specified must be a column of the base table (SQLSTATE 42703). No *column-name* shall appear more than once in the *column-name* list (SQLSTATE 42711). The trigger will only be activated by the update of a column identified in the *column-name* list.

**ON** *table-name*
Designates the subject table of the trigger definition. The name must specify a base table or an alias that resolves to a base table (SQLSTATE 42809). The name must not specify a catalog table (SQLSTATE 42832), a typed table or summary table (SQLSTATE 42997), or a nickname (SQLSTATE 42809).

**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 prior to 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 a temporary table name which identifies the set of affected rows prior to the triggering SQL operation.

**NEW_TABLE AS** *identifier*
> Specifies a temporary table name which identifies the affected rows as modified by the triggering SQL operation and by any SET statement in a BEFORE trigger that has already executed.

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).
- 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.
- OLD_TABLE and NEW_TABLE *identifier*s cannot be defined for a BEFORE trigger (SQLSTATE 42898).

## CREATE TRIGGER

- OLD and NEW *correlation-name*s cannot be defined for 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.

**FOR EACH ROW**
Specifies that the triggered action is to be applied once for each row of the subject table 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 (SQLSTATE 42613). If specified, an UPDATE or DELETE trigger is activated even when no rows are affected by the triggering UPDATE or DELETE statement.

**MODE DB2SQL**
This clause is used to specify the mode of triggers. This is the only valid mode currently supported.

**triggered-action**
Specifies the action to be performed when a trigger is activated. A triggered-action is composed of one or several *triggered-SQL-statement*s and by an optional condition for the execution of the *triggered-SQL-statement*s. If there is more than one *triggered-SQL-statement* in the triggered-action for a given trigger, they must be enclosed within the BEGIN ATOMIC and END keywords, separated by a semi-colon, [75] and are executed in the order they are specified.

  **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 *triggered-SQL-statement*s are always performed.

---

[75]. When using this form in the Command Line Processor, the statement terminating character cannot be the semi-colon. See the *Command Reference* for information on specifying an alternative terminating character.

*triggered-SQL-statement*

If the trigger is a BEFORE trigger, then a triggered SQL statement must be one of the following (SQLSTATE 42987):
- a fullselect [76]
- a SET transition-variable SQL statement.
- a SIGNAL SQLSTATE statement

If the trigger is an AFTER trigger, then a triggered SQL statement must be one of the following (SQLSTATE 42987):
- an INSERT SQL statement
- a searched UPDATE SQL statement
- a searched DELETE SQL statement
- a SIGNAL SQLSTATE statement
- a fullselect [76]

The *triggered-SQL-statement* can not reference an undefined transition variable (SQLSTATE 42703).

The *triggered-SQL-statement* in a BEFORE trigger can not reference a summary table defined with REFRESH IMMEDIATE (SQLSTATE 42997).

### 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 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.

---

76. A common-table-expression may precede a fullselect.

**CREATE TRIGGER**

- The result of a fullselect specified as a *triggered-SQL-statement* is not available inside or outside of the trigger.
- 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 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 are invalidated.
  - If a delete trigger is created, packages that have a delete usage on the target table 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 an view is dropped, any trigger with a *triggered-SQL-statement* defined using 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 recreated 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 SYSCAT.TRIGGERS. Note that there is no need to explicitly drop the inoperative trigger in order to recreate 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.

  A *triggered-SQL-statement* could be a SIGNAL SQLSTATE statement or contain 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.

## 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

```
CREATE TRIGGER NEW_HIRED
    AFTER INSERT ON EMPLOYEE
    FOR EACH ROW MODE DB2SQL
    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 MODE DB2SQL
    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 MODE DB2SQL
    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:* 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 MODE DB2SQL
    WHEN (N.SALARY > 1.1 * O.SALARY)
        SIGNAL SQLSTATE '75000' ('Salary increase>10%')
```

*Example 4:* 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 MODE DB2SQL
  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 MODE DB2SQL
  BEGIN ATOMIC
    INSERT INTO QUOTEHISTORY
      VALUES (NEWQUOTE.SYMBOL, NEWQUOTE.QUOTE, CURRENT TIMESTAMP);
  END
```

## CREATE TYPE (Structured)

The CREATE TYPE statement defines a user-defined structured type. A user-defined structured type may include zero or more attributes. A structured type may be a subtype allowing attributes to be inherited from a supertype. Successful execution of the statement also generates functions to cast between the reference type and its representation type and generates support for the comparison operators (=, <>, <, <=, >, and >=) on the reference 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. 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 as least one of the following:
- SYSADM or DBADM authority
- 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.

If UNDER is specified and the authorization ID of the statement is not the same as the definer of the root type of the type hierarchy, then SYSADM or DBADM authority is required.

### Syntax

```
►►──CREATE TYPE──type-name─────────────────────────────────────────────►
                          └─UNDER──supertype-name─┘


                                                    ┌─WITHOUT COMPARISONS─┐
►─────────────────────────────────────────────●─────┴─────────────────────┴──●──►
       ┌─────────────────────────────┐
       │         ┌─,─────────────┐   │
    └─AS──(──▼──┤ attribute-definition ├──┴──)─┘


   ┌─NOT FINAL─┐
►──┴───────────┴──●──MODE DB2SQL──●─────────────────────────────●──────────►◄
                                  └─REF USING──┤ rep-type ├──┘
```
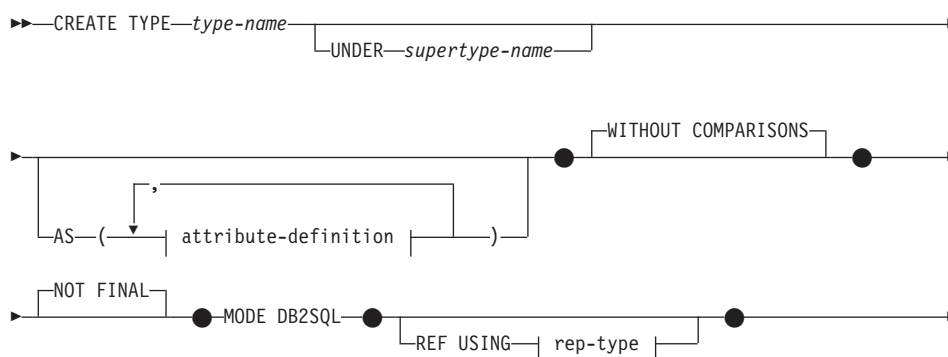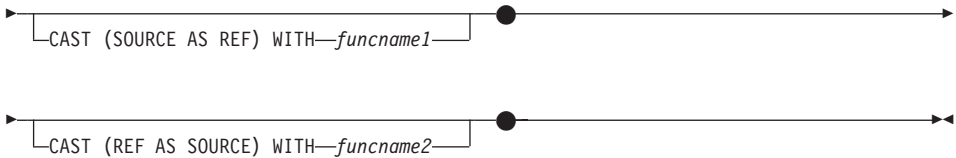
**attribute-definition:**
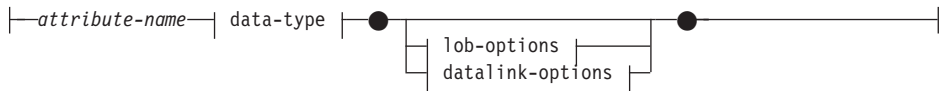


**rep-type:**
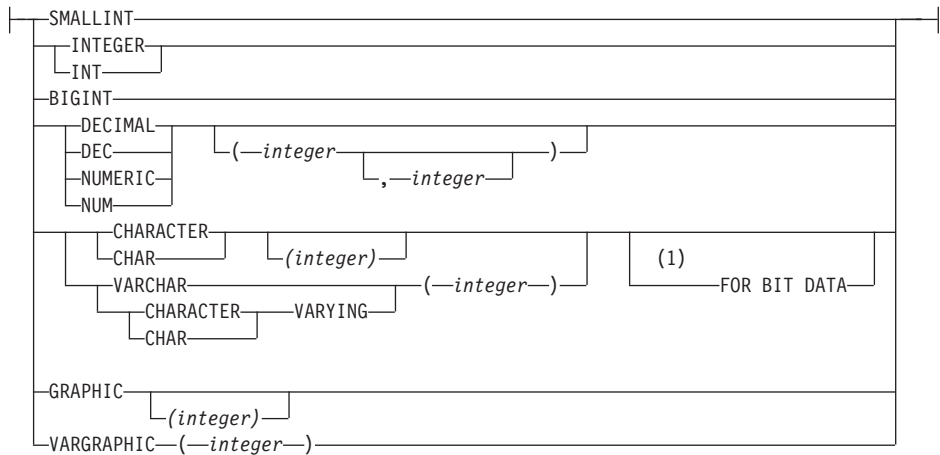


**Notes:**

**1** The FOR BIT DATA clause may be specified in random order with the other column constraints that follow.

## 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) already described in the catalog. The unqualified name must not be the same as the name of a built-in data type or BOOLEAN (SQLSTATE 42918). In dynamic SQL statements, the CURRENT SCHEMA special register is used

## CREATE TYPE (Structured)

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.

A number of names used as keywords in predicates are reserved for system use, and may not be used as a *type-name*. The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH and the comparison operators as described in "Basic Predicate" on page 167. Failure to observe this rule will lead to an error (SQLSTATE 42939).

If a two-part *type-name* is specified, the schema name cannot begin with "SYS"; otherwise, an error (SQLSTATE 42939) is raised.

**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 may not be used as an *attribute-name*. The names are SOME, ANY, ALL, NOT, AND, OR, BETWEEN, NULL, LIKE, EXISTS, IN, UNIQUE, OVERLAPS, SIMILAR, MATCH and the comparison operators as described in "Basic Predicate" on page 167. Failure to observe this rule will lead to an error (SQLSTATE 42939).

*data-type*
The data type of the attribute. It is one of the data types listed under "CREATE TABLE" on page 639 other than LONG VARCHAR, LONG VARGRAPHIC, or a distinct type based on LONG VARCHAR or LONG VARGRAPHIC (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" on page 639. 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).

*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" on page 639.

*datalink-options*

> Specifies the options associated with DATALINK types (or distinct types based on DATALINK types). For a detailed description of *datalink-options*, see "CREATE TABLE" on page 639.

> Note that if no options are specified for a DATALINK type or distinct type sourced on DATALINK, LINKTYPE URL and NO LINK CONTROL options are the defaults.

**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 used to specify the mode of the type. This is the only value for mode currently supported.

**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).

> 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).

### 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 root type of the type hierarchy is defined without any attributes.
- Creation of a structured type automatically generates a set of functions for use with the type. All the functions are generated in the same schema as the structured type. If the signature of the generated function conflicts with the signature of an existing function in this schema, the statement fails (SQLSTATE 42710, SQLCODE -600). The generated functions cannot be dropped without dropping the structured type (SQLSTATE 42917, SQLCODE -658). The following functions are generated:
  - 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 behaviour as the comparison operators for the underlying data type of REF(*type-name*).[77]

    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 funcname2 (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*.

---

[77]. 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. Since 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 format of this function is:

```
CREATE FUNCTION funcname1 ( 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)
```

• Constructor

The constructor function is generated to allow a new instance of the type to be constructed. This function cannot be used in the current version.

The format of the generated constructor function is:

```
CREATE FUNCTION type-name ( )  RETURNS type-name ...
```

The schema name of the structured type must be included in the SQL path (see "SET PATH" on page 892 or the FUNCPATH BIND option as described in the *Application Development Guide*) for successful use of these operators and cast functions in SQL statements.

## Examples

*Example 1:* Create a type for department.

```
CREATE TYPE DEPT AS
      (NAME        VARCHAR(32),
       NUMBER    CHAR(6))
           MODE DB2SQL
```

*Example 2:* Create a type hierarchy consisting of a type for employees and a subtype for managers. Every employee has a six digit ID number so use a DECIMAL(6,0) as the reference type. The cast functions should be called ID_TO_REF and REF_TO_ID.

```
CREATE TYPE EMP AS
      (NAME        VARCHAR(32),
       DEPTREF      REF(DEPT),
       SALARY    DECIMAL(10,2) )
```

## CREATE TYPE (Structured)

```
                        MODE DB2SQL
      REF USING DECIMAL (6,0)
      CAST (SOURCE AS REF) WITH ID_TO_REF
      CAST (REF AS SOURCE) WITH REF_TO_ID

        CREATE TYPE MGR UNDER EMP AS
              (HEADCOUNT INTEGER,
               BUDGET    DECIMAL(10,2) )
                      MODE DB2SQL
```

## CREATE TYPE MAPPING

The CREATE TYPE MAPPING statement creates a mapping between these data types:

- A data type of a column of 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 either (1) a specified data source or (2) a range of data sources; for example, all data sources of a particular type and version.

A data type mapping has to be created only if an existing one is not adequate.

### 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 have SYSADM or DBADM authority.

### Syntax
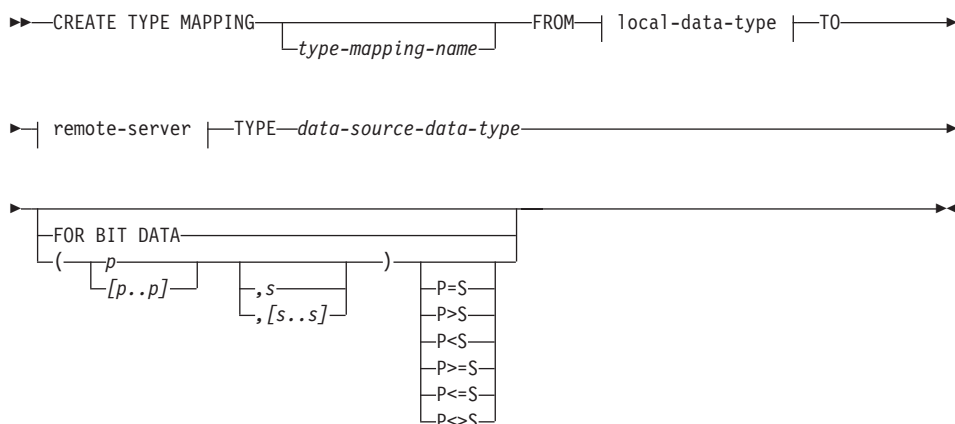
**CREATE TYPE MAPPING**

**local-data-type:**
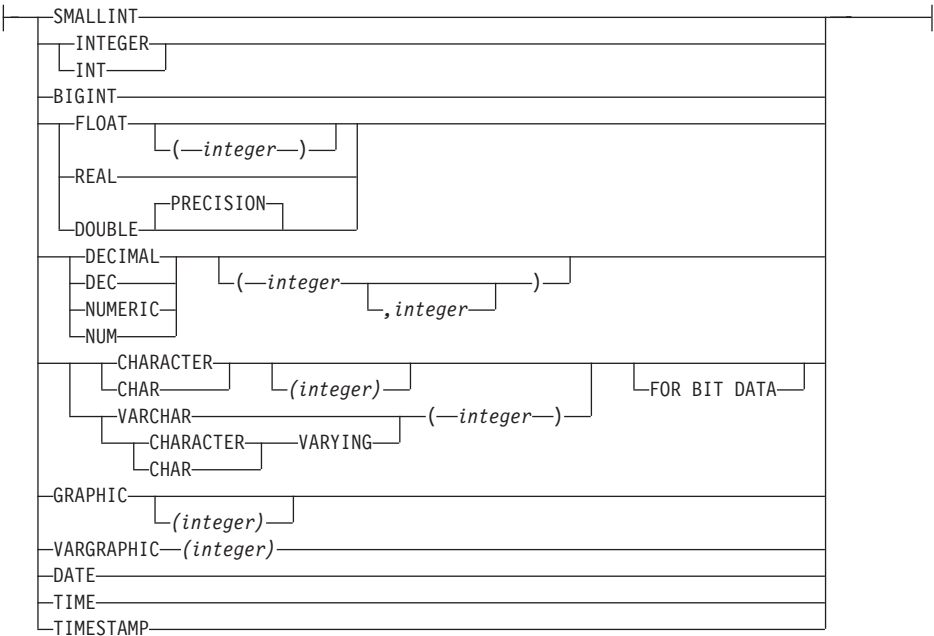


**remote-server:**
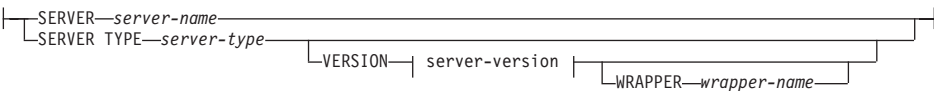


**server-version:**
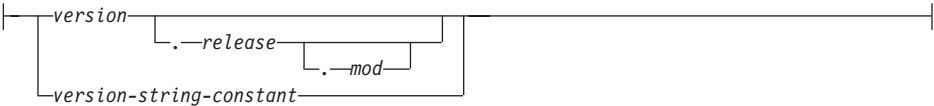


## Description

*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.

*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 on the SQL path (defined by the FUNCPATH preprocessing option for static SQL and by the CURRENT PATH register for dynamic SQL).

The *local-data-type* cannot be LONG VARCHAR, LONG VARGRAPHIC, DATALINK, a large object (LOB) type, or a user-defined type.

**SERVER** *server-name*
Names the data source to which *data-source-data-type* is defined.

**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. *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*
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*.

*data-source-data-type*
Specifies the data source data type that is being mapped to *local-data-type*. If *data-source-data-type* is qualified by a schema name at the data source, it is allowable, but not required, to specify this qualifier.

The *data-source-data-type* must be a built-in data type. If the type has a short and long form (for example, CHAR and CHARACTER), the short form should be specified.

*p* For a decimal data type, *p* specifies the maximum number of digits that a value can have. For all other data types for character data, *p* specifies the maximum number of characters that a value can have. The *p* must be

valid with respect to the data type (SQLSTATE 42611). If *p* is not specified and the data type requires it, the system will determine the best match.

*[p..p]*
>   For a decimal data type, *[p..p]* specifies the minimum and maximum number of digits that a value can have. For all other data types for character data, *[p..p]* specifies 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 (SQLSTATE 42611).

*s*
>   For a decimal data type, *s* specifies the allowable maximum number of digits to the right of the decimal point. This number must be valid with respect to the data type (SQLSTATE 42611). If a number is not specified and the data type requires one, the system will determine the best match.

*[s..s]*
>   For a decimal data type, *[s..s]* specifies 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 (SQLSTATE 42611).

**P** [operand] **S**
>   For a decimal data type, P [operand] S specifies a comparison between the maximum allowable precision and the maximum number of digits allowed to the right of the decimal point. For example, the operand = indicates that the maximum allowable precision and the maximum number digits allowed in the decimal fraction are the same. Specify P [operand] S only if the level of checking that it enforces is required.

**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 on a character data type.

### Notes

A CREATE TYPE MAPPING statement 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 a SELECT statement that references 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 a SELECT statement that references a nickname for a table or view within one of these data sources.

## Examples

*Example 1:* Create a mapping between SYSIBM.DATE and the Oracle data type DATE at all Oracle data sources.

```
CREATE TYPE MAPPING MY_ORACLE_DATE
   FROM SYSIBM.DATE
   TO SERVER TYPE ORACLE
   TYPE DATE
```

*Example 2:* Create a mapping between SYSIBM.DECIMAL(10,2) and the Oracle data type NUMBER([10..38],2) at data source ORACLE1.

```
CREATE TYPE MAPPING MY_ORACLE_DEC
   FROM SYSIBM.DECIMAL(10,2)
   TO SERVER ORACLE1
   TYPE NUMBER([10..38],2)
```

## 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. However, if the bind option DYNAMICRULES BIND
applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

### Authorization

If the authorization ID of the statement is different than the authorization
name that is being mapped to the data source, then the authorization ID must
include SYSADM or DBADM authority. Otherwise, if the authorization ID and
the authorization name match, then no privileges or authorities are required.

### Syntax

```
►►──CREATE USER MAPPING FOR──┬─authorization-name─┬──SERVER──server-name──────────►
                             └─USER───────────────┘

                        ┌──────────────────────,──────────────────────┐
                        │     ┌─ADD─┐                                  │
►──OPTIONS──(───────────▼─────┴─────┴──user-option-name──string-constant──────┴──)───►◄
```
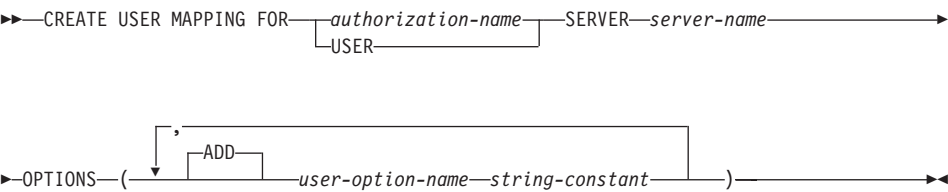
### Description

*authorization-name*
   Specifies the authorization name under which a user or application
   connects to a federated database. This name is to be mapped to an
   identifier under which the data source denoted by *server-name* can be
   accessed.

**USER**
   The value in the special register USER. When USER is specified, then the
   authorization ID of the CREATE USER MAPPING statement 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 that is accessible under the mapping
   authorization ID.

**OPTIONS**
Indicates what user options are to be enabled. Refer to "User Options" on page 1053 for descriptions of *user-option-name*s and their settings.

**ADD**
Enables one or more user options.

*user-option-name*
Names a user option that will be used to complete the user mapping that is being created.

*string-constant*
Specifies the setting for *user-option-name* as a character string constant.

## Notes

- A user mapping cannot be created in a given unit of work (UOW) if the UOW already includes a SELECT statement that references a nickname for a table or view at the data source that is to be included in the mapping.

## Examples

*Example 1:* To access a data source called S1, you need to map your authorization name and password for your local database to your user ID and password for S1. Your authorization name is RSPALTEN, and the user ID and password that you use for S1 are SYSTEM and MANAGER, respectively.

```
CREATE USER MAPPING FOR RSPALTEN
   SERVER S1
   OPTIONS
   ( REMOTE_AUTHID 'SYSTEM',
      REMOTE_PASSWORD 'MANAGER' )
```

*Example 2:* Marc already has access to a DB2 data source. He now needs access to an Oracle data source, so that he can create joins between certain DB2 and Oracle tables. He acquires a username and password for the Oracle data source; the username is the same as his authorization ID for the federated database, but his Oracle and federated database passwords are different. To be able to access Oracle from the federated database, he must map the two passwords together.

```
CREATE USER MAPPING FOR MARCR
   SERVER ORACLE1
   OPTIONS
   ( REMOTE_PASSWORD 'NZXCZY' )
```

## CREATE VIEW

The CREATE VIEW statement creates 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. 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 at least one of the following:
- SYSADM or DBADM authority or
- For each table, view or nickname identified in any fullselect:
  - CONTROL privilege on that table or view, or
  - SELECT privilege on that table or view

  and at least one of the following:
  - 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.

  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.
  - have SELECT WITH GRANT on the underlying table of the subview or the superview must not have SELECT privilege granted to any user other than the view definer.

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 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 may be mapped to a different remote authorization ID.

If a view definer can only create the view because the definer has SYSADM authority, then the definer is granted explicit DBADM authority for the purpose of creating the view.

## Syntax

```
►►──CREATE──────────────────VIEW──view-name──────────────────────────────────►
              └─FEDERATED─┘
```
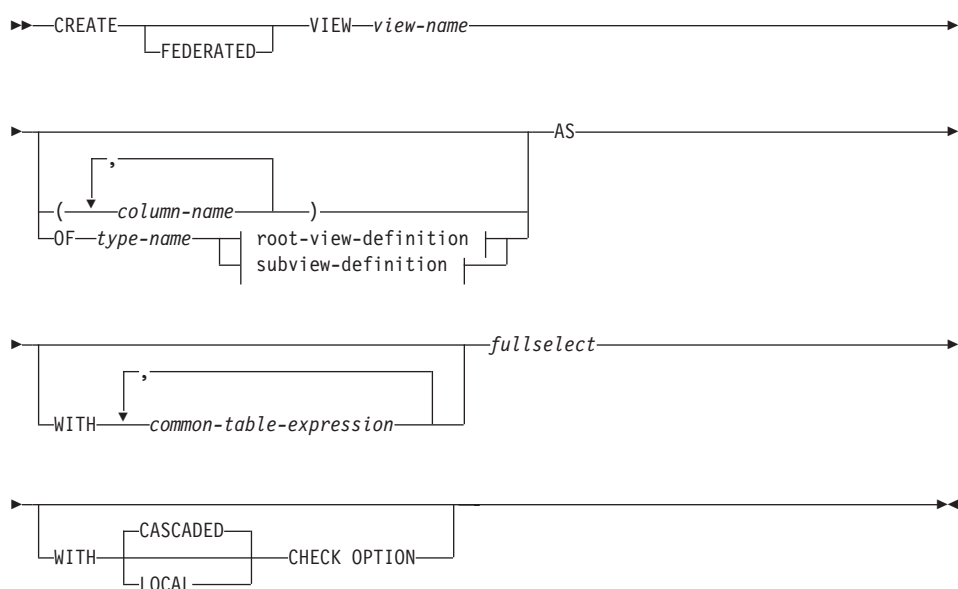
```
►────────────────────────────────────────────AS──────────────────────────────►
       ┌──────────,──────────┐
       │                     │
    ┌──(───▼──column-name──┬───)──────────────┐
    │                                          │
    └──OF──type-name──┬─ root-view-definition ─┤
                      └─ subview-definition ───┘
```

```
►───────────────────────────────────fullselect──────────────────────────────►
                ┌────────────,────────────┐
                │                          │
    └──WITH──────▼──common-table-expression──┘
```

```
►──────────────────────────────────────────────────────────────────────────►◄
       ┌─CASCADED─┐
    └──WITH──┤          ├──CHECK OPTION──┘
       └─LOCAL────┘
```

**root-view-definition:**

```
├──MODE DB2SQL──(──┤ oid-column ├───────────────────────)──────────────┤
                                  └──,──┤ with-options ├──┘
```

**subview-definition:**

```
├──MODE DB2SQL──┤ under-clause ├──────────────────────────────────────────┤
                                 └──(──┤ with-options ├──)──┘  └──EXTEND──┘
```

**oid-column:**

```
├──REF IS──oid-column-name──USER GENERATED────────────────────────────┤
                                                └──UNCHECKED──┘
```
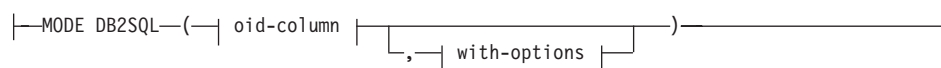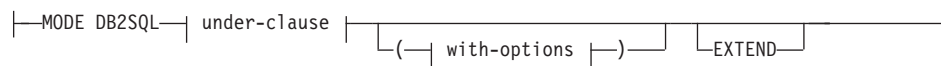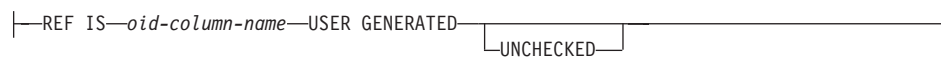
**with-options:**

## CREATE VIEW

```
        ┌──────────────────────────────────────────────────┐
        │  ,                                               │
        ▼                      ┌──────────────────┐         │
──┬──column-name──WITH OPTIONS──┬──SCOPE──┬──typed-table-name──┬──┬──┬──
                                │         └──typed-view-name───┘  │  │
                                └──READ ONLY──────────────────────┘
```
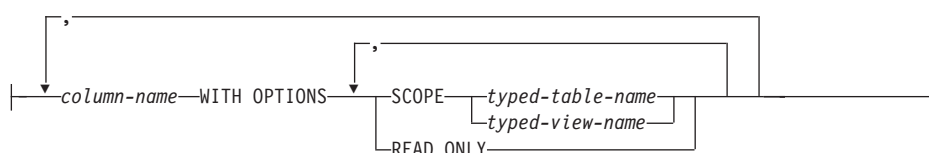
**under-clause:**

```
├──UNDER──superview-name──INHERIT SELECT PRIVILEGES───────────────────┤
```

> **Note:** See "Chapter 5. Queries" on page 367 for the syntax of
> *common-table-expression* and *fullselect*.

## Description

**FEDERATED**

Indicates that the view is created with a reference to nickname. A warning message is generated if a view referencing a nickname is created without specifying FEDERATED optional clause.

*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" on page 723). 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 (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 722 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. See "common-table-expression" on page 413.

*fullselect*
> Defines the view. At any time, the view consists of the rows that would result if the SELECT statement were executed. The fullselect must not reference host variables or parameter markers.

> **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 NODENUMBER or PARTITION 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 (i.e., a view may not 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, then 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 the view is read-only (SQLSTATE 42813). If WITH CHECK OPTION is specified for an updatable view that does not allow inserts, then the constraint applies to updates only.

WITH CHECK OPTION must not be specified if the view references the NODENUMBER or PARTITION function, a non-deterministic function, or a function with external action (SQLSTATE 42997).

WITH CHECK OPTION must not be specified if the view is a typed view (SQLSTATE 42997).

WITH CHECK OPTION must not be specified if a nickname is the update target of the view.

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:

## CREATE VIEW

|  | **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)
```

### 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)
- *Deletable views:*  A view is *deletable* if all of the following 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)
  - 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 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 all of the following are true:
  - the view is deletable
  - the column resolves to a column of a base table (not using a dereference operation) and 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 ALL columns of the view are updatable and the fullselect of the view does not include UNION ALL.
- *Read-only views:*  A view is *read-only* if it is NOT deletable.

  The READONLY column in the SYSCAT.VIEWS catalog view indicates if a view is read-only.
- Common table expressions and nested table expressions follow the same set of rules for determining whether they are deletable, updatable, insertable or read-only.
- *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.

**CREATE VIEW**

      – 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 recreated 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 recreated 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 recreate 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 SYSADM or DBADM authority.

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.

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 the privilege.

Privileges are not considered when defining a view on 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)
    ```

- *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 both at create time and when the view is subsequently referenced in a query. If a view is created that references a nickname and the FEDERATED keyword is not included, a warning is issued to indicate that the authorization requirements for this view are different because of the reference to a nickname.

  A nickname has no associated DML privileges and therefore when the view is created, no privilege checking is done to determine whether the view definer has access to the nickname or to the underlying data source table or view. 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 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.

# CREATE VIEW

## 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:
User ZORPIE (who does not have either DBADM or SYSADM authority) has
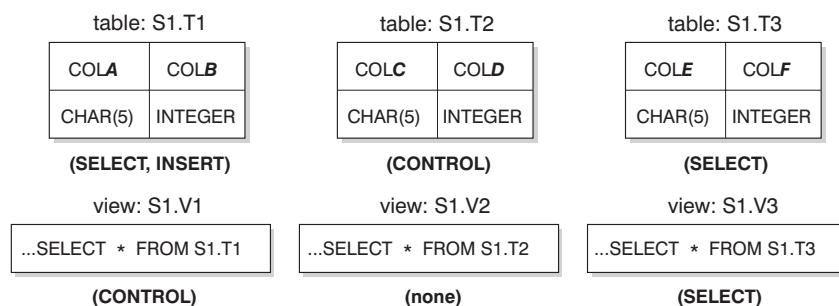


*Figure 13. Tables and Views for Example 6*

been granted the privileges shown in brackets below 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.[78] It 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
    ```

---

78. CONTROL on S1.V1 must have been granted to ZORPIE by someone with DBADM or SYSADM authority.

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 WRAPPER

The CREATE WRAPPER statement registers a wrapper—a mechanism by which a federated server can interact with a certain category of data sources—to a federated database.

### 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 authorization ID of the statement must have SYSADM or DBADM authority.

### Syntax

```
►►──CREATE WRAPPER──wrapper-name──────────────────────────────────►◄
                              └─LIBRARY──'library-name'─┘
```

### 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 to *'library-name'*.

        The predefined names are:

        DRDA            For all DB2 family data sources

        NET8            For all Oracle data sources that are supported by Oracle's Net8 client software

        SQLNET          For all Oracle data sources that are supported by Oracle's SQL*Net client software
    - A user-supplied name. If such a name is provided, it is necessary to also specify *'library-name'*.

**LIBRARY** *'library-name'*
    Names the file that contains the wrapper module. Refer to the *Administration Guide, Performance* for information on how to determine what files to specify for each wrapper.

## CREATE WRAPPER

### Notes

Refer to *Installation and Configuration Supplement* for more information on how to select and define wrappers.

### Examples

*Example 1:* Register a wrapper that the federated server can use to interact with an Oracle data source that is supported by Oracle's SQL*Net client software. Use the predefined name.

```
CREATE WRAPPER SQLNET
```

*Example 2:* Register a wrapper that the federated server on an AIX system can use to interact with DB2 for VM and VSE data sources. Specify a name to indicate that these data sources are used for testing.

```
CREATE WRAPPER TEST
   LIBRARY 'libsqlds.a'
```

The extension in the library name (a) indicates that wrapper TEST is for data sources that reside in an AIX system.

## 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.

### Authorization

The term "SELECT statement of the cursor" is used in order to specify the authorization rules. The SELECT statement of the cursor is one of the following:
* The prepared select-statement identified by the *statement-name*
* The specified *select-statement*.

For each table or view identified (directly or using an alias) in the SELECT statement of the cursor, the privileges held by the authorization ID of the statement must include at least one of the following:
* SYSADM or DBADM authority.
* For each table or view identified in the *select-statement:*
  – SELECT privilege on the table or view, or
  – CONTROL privilege of the table or view.

**If *statement-name* is specified:**
* The authorization ID of the statement is the run-time authorization ID.
* The authorization check is performed when the select-statement is prepared.
* The cursor cannot be opened unless the select-statement is successfully prepared.

**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

```
►►──DECLARE──cursor-name──CURSOR──────────────────────────────────────────►
                                  └─WITH HOLD─┘  └─WITH RETURN─┘
```

## DECLARE CURSOR

```
►──FOR──┬─select-statement─┬──────────────────────────────────────────►◄
         └─statement-name───┘
```

### 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 (see "OPEN" on page 819).

**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.
> - 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 may 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 may result in errors during subsequent execution.

**WITH RETURN**
Option is available for applications that require compatibility with DB2 Universal Database for OS/390. This option should only be used when declaring a cursor in the code for a stored procedure. When specified, it is ignored by DB2 Universal Database.

*select-statement*
Identifies the SELECT statement of the cursor. The *select-statement* must not include parameter markers, but may include references to host variables. The declarations of the host variables must precede the DECLARE CURSOR statement in the source program. See "select-statement" on page 412 for an explanation of *select-statement*.

*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" on page 824.

## 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.
- 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 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 all of the following are true:

## DECLARE CURSOR

- 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 select list of the outer fullselect does not include DISTINCT
- the select-statement does not include an ORDER BY clause
- the select-statement does not include a FOR READ ONLY clause
- one or more of the following is true:
  - the FOR UPDATE clause is specified
  - the cursor is statically defined
  - 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 all of the following are 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 [80] (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 all of the following are true:
- the select-statement is dynamically prepared
- the select-statement does not include either the FOR READ ONLY clause [79] or the FOR UPDATE clause [80]
- 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 deletable.

- Cursors in stored 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. See the "Notes" on page 489.

---

79. The FOR READ ONLY clause is defined in the "read-only-clause" on page 420.

80. The FOR UPDATE clause is defined in the "update-clause" on page 419.

734

- Cursors referencing a nickname as a potential target are assumed to be read only.

## Example

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';
```

## DELETE

The DELETE statement deletes rows from a table or view. Deleting a row from a view deletes the row from the table on which the view is based.

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:
- DELETE privilege on the table or view for which rows are to be deleted
- CONTROL privilege on the table or view for which rows are to be deleted
- SYSADM or DBADM 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 for each table or view referenced by a subquery:
- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM authority.

When the package is precompiled with SQL92 rules[81] and the searched form of a DELETE 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:
- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM authority.

When 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.

---

81. The package used to process the statement is precompiled using option LANGLEVEL with value SQL92E or MIA.

Group privileges are not checked for static DELETE statements.

## Syntax

**Searched DELETE:**

```
►►──DELETE FROM──┬─table-name─────────────────────┬──┬────────────────────────┬──►
                 ├─view-name──────────────────────┤  │ ┌─AS─┐                  │
                 └─ONLY──(──┬─table-name─┬──)──────┘  └─┴────┴──correlation-name─┘
                           └─view-name──┘

►──┬─────────────────────────────────┬──►◄
   └─WHERE──search-condition──────────┘
```

**Positioned DELETE:**

```
►►──DELETE FROM──┬─table-name─────────────────────┬──WHERE CURRENT OF──cursor-name──►◄
                 ├─view-name──────────────────────┤
                 └─ONLY──(──┬─table-name─┬──)──────┘
                           └─view-name──┘
```
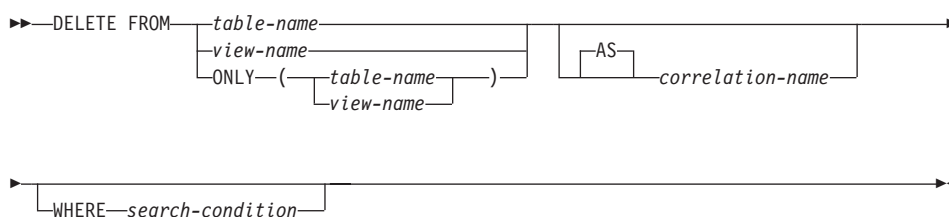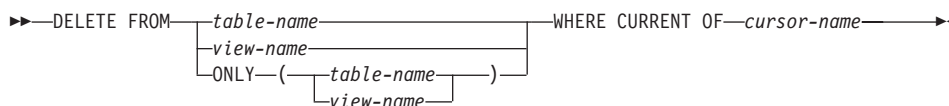
## Description

**FROM** *table-name* or *view-name*

Identifies the table or view from which rows are to be deleted. The name must identify a table or view that exists in the catalog, but it must not identify a catalog table, a catalog view, a summary table or a read-only view. (For an explanation of read-only views, see "CREATE VIEW" on page 714.)

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.

Only the columns of the specified table may 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.

**correlation-name**

May be used within the search-condition to designate the table or view. (For an explanation of correlation-name, see "Chapter 3. Language Elements" on page 61.)

**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*

Is any search condition as described in "Search Conditions" on page 185. 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 or view 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 or view 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" on page 731.)

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.

## Rules

- 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 above 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.

## 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 deleted from the object table after the statement executes. It does not include rows that were deleted as a result of a CASCADE delete rule. 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 NULL 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. (For a description of the SQLCA, see "Appendix B. SQL Communications (SQLCA)" on page 927.)

- 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 any deleted row that includes currently linked files through DATALINK columns, the files are unlinked, and will be either restored or deleted, depending on the datalink column definition.

  An error may occur when attempting to delete a DATALINK value if the file server of value is no longer registered with the database server (SQLSTATE 55022).

  An error may also occur when deleting a row that has a link to a server that is unavailable at the time of deletion (SQLSTATE 57050).

## 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
```

## DESCRIBE

The DESCRIBE statement obtains information about a prepared statement. For an explanation of prepared statements, see "PREPARE" on page 824.

### Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

### Authorization

None required.

### Syntax

```
►►──DESCRIBE──statement-name──INTO──descriptor-name──────────────────►◄
```

### Description

*statement-name*
Identifies the statement about which information is required. When the DESCRIBE statement is executed, the name must identify a prepared statement.

**INTO** *descriptor-name*
Identifies an SQL descriptor area (SQLDA), which is described in "Appendix C. SQL Descriptor Area (SQLDA)" on page 933. Before the DESCRIBE statement is executed, the following variables in the SQLDA must be set:

**SQLN** Indicates the number of variables represented by SQLVAR. (SQLN provides the dimension of the SQLVAR array.) SQLN must be set to a value greater than or equal to zero before the DESCRIBE statement is executed.

When the DESCRIBE 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, called SQLDOUBLED, is set to '2' if the SQLDA contains two SQLVAR entries for every select-list item (or, *column* of the result table). This technique is used in order to accommodate LOB, distinct or reference type result columns. Otherwise, SQLDOUBLED is set to the space character.

## DESCRIBE

> The doubled flag is set to space if there is not enough room in the SQLDA to contain the entire DESCRIBE reply.
>
> The eighth byte is set to the space character.

**SQLDABC**    Length of the SQLDA.

**SQLD**    If the prepared statement is a SELECT, the number of columns in its result table; otherwise, 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 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 so that the first occurrence of SQLVAR contains a description of the first column of the result table, the second occurrence of SQLVAR contains a description of the second column of the result table, and so on. The description of a column consists of the values assigned to SQLTYPE, SQLLEN, SQLNAME, SQLLONGLEN, and SQLDATATYPE_NAME.

*Basic SQLVAR*

**SQLTYPE**    A code showing the data type of the column and whether or not it can contain null values.

**SQLLEN**    A length value depending on the data type of the result columns. SQLLEN is 0 for LOB data types.

**SQLNAME**    If the derived column is not a simple column reference, then sqlname contains an ASCII numeric literal value, which represents the derived column's original position within the select list; otherwise, sqlname contains the name of the column.

*Secondary SQLVAR*

These variables are only used if the number of SQLVAR entries are doubled to accommodate LOB or distinct type columns.

**SQLLONGLEN**
> The length attribute of a BLOB, CLOB, or DBCLOB column.

**SQLDATATYPE_NAME**
> For a distinct type column, the database

manager sets this to the fully qualified distinct
type name. Otherwise, schema name is
SYSIBM and the low order portion of the
name is the name in the TYPENAME column
of the SYSCAT.DATATYPES catalog view.

**Notes**

- Before the DESCRIBE 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. 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 statement is executed but prior to 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.

  See the *Application Development Guide* for more information on using
  locators and file reference variables with the SQLDA.
- Code page conversions between extended Unix code (EUC) code pages and
  DBCS code pages can result in the expansion and contraction of character
  lengths. See the *Application Development Guide* for information on handling
  such situations.
- **Allocating the SQLDA:** Among the possible ways to allocate the SQLDA
  are the three described below.

  *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 or distinct 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 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.[82]

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 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.

## Example

In a C program, execute a DESCRIBE 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             */
```

---

82. 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.

```
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).[83]

### 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

```
            (1)
►►──DISCONNECT──┬─server-name──┬────────────────────────────────►◄
               ├─host-variable─┤
               ├─CURRENT───────┤
               │        ┌─SQL─┐│
               └─ALL────┴─────┴┘
```

**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-justified 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.

---

83. If a single connection is the target of the DISCONNECT statement, then the connection is destroyed only if the database has participated in any existing unit of work, not 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.

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.

## DISCONNECT

- 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. See "Options that Govern Distributed Unit of Work Semantics" on page 37 for information about the specification of the DISCONNECT option.

### 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. (See "Inoperative Trigger" on page 696 and "Inoperative views" on page 723 for details.) 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. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

### Authorization

The privileges that must be held by the authorization ID of the DROP statement when dropping objects that allow two-part names must include one of the following:
- SYSADM or DBADM authority
- DROPIN privilege on the schema for the object
- definer of the object as recorded in the DEFINER column of the catalog view for the object
- CONTROL privilege on the object (applicable only to indexes, index specifications, nicknames, packages, tables, and views).

The authorization ID of the DROP statement when dropping a table or view hierarchy must hold one of the above privileges for each of the tables or views in the hierarchy.

The authorization ID of the DROP statement when dropping a schema must have SYSADM or DBADM authority or be the schema owner as recorded in the OWNER column of SYSCAT.SCHEMATA.

The authorization ID of the DROP statement when dropping a buffer pool, nodegroup, or table space must have SYSADM or SYSCTRL authority.

The authorization ID of the DROP statement when dropping an event monitor, server definition, data type mapping, function mapping or a wrapper must have SYSADM or DBADM authority.

The authorization ID of the DROP statement when dropping a user mapping must have SYSADM or DBADM authority, if this authorization ID is different

## DROP

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
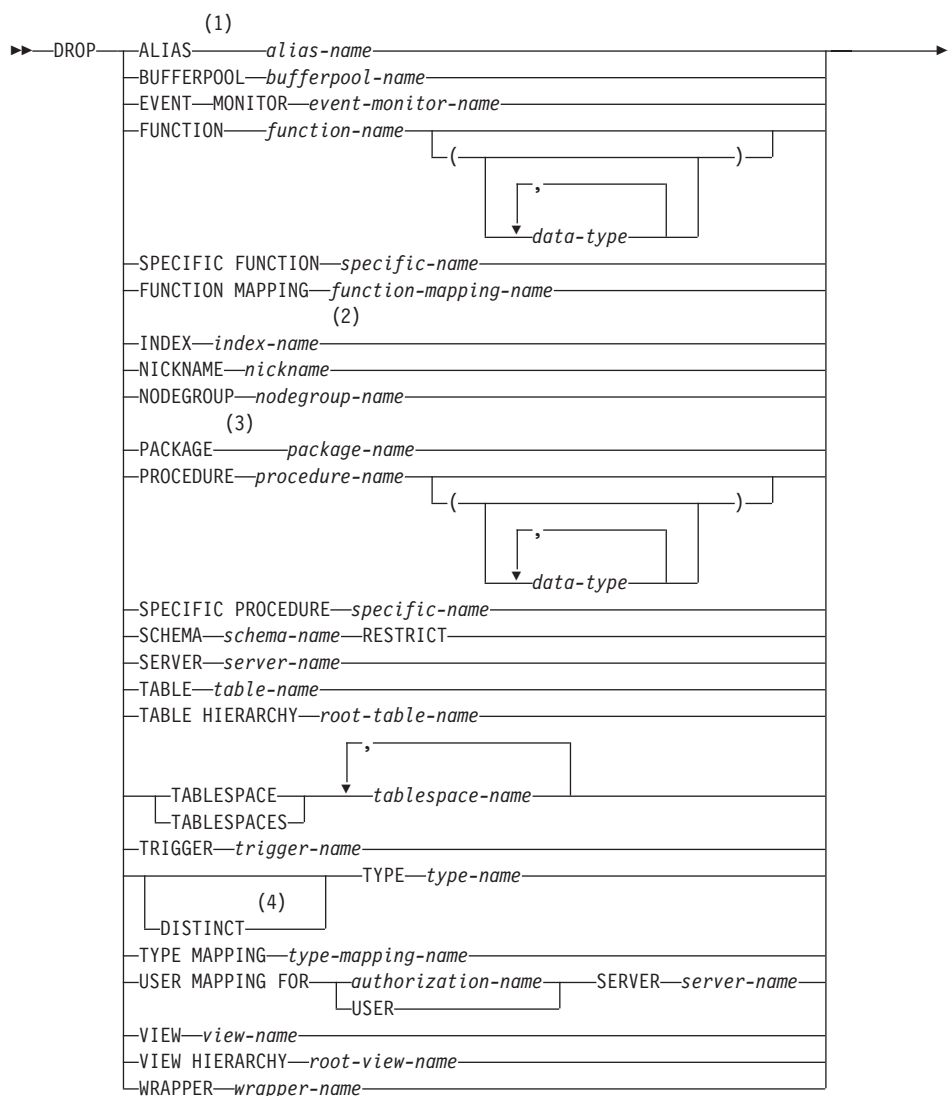


**Notes:**

**1**    SYNONYM can be used as a synonym for ALIAS.

**2**    *-Index-name* can be the name of either an index or an index specification.

**3** PROGRAM can be used as a synonym for PACKAGE.

**4** DATA can also be used when dropping any user-defined type.

## Description

**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.

All tables, views and triggers[84]that reference the alias are made inoperative.

**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). The storage for the buffer pool will not be released until the database is stopped.

**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 ON, an error (SQLSTATE 55034) is raised. Otherwise, the event monitor is deleted.

If there are event files in the target path of the event monitor when the event monitor is dropped, the event files are not deleted. However, if a new event monitor is created which specifies the same target path, then the event files are deleted.

**FUNCTION**
Identifies an instance of a user-defined function (either a complete function or a function template) that is to be dropped. The function instance specified must be a user-defined function described in the catalog. Functions implicitly generated by the CREATE DISTINCT TYPE statement cannot be dropped.

There are several different ways available to identify the function instance:

**FUNCTION** *function-name*
Identifies the particular function, and is valid only if there is exactly one function instance with the *function-name*. The function thus identified may have any number of parameters defined for it. In

---

84. This includes both the table referenced in the ON clause of the CREATE TRIGGER statement and all tables referenced within the triggered SQL statements.

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 specific instance of the function in the named or implied schema, an error (SQLSTATE 42854) is raised.

**FUNCTION** *function-name* **(***data-type***,...)**
Provides the function signature, which uniquely identifies the function to be dropped. The function selection algorithm is **not** used.

*function-name*
Gives the function name of the function 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.

*(data-type,...)*
Must match the data types that were specified on the CREATE FUNCTION statement in the corresponding position. The number of data types, and the logical concatenation of the data types is used to identify the specific function instance which is to be dropped.

If the *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 may be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601) since 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 PROCEDURE statement.

A type of FLOAT(n) does not need to match the defined value for n since 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 named or implied schema, an error (SQLSTATE 42883) is raised.

**SPECIFIC FUNCTION** *specific-name*
> Identifies the particular user-defined function that is to be dropped, 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/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.

It is not possible to drop a function that is in either the SYSIBM schema or the SYSFUN schema (SQLSTATE 42832).

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 page 760 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 reprepared by use of the PREP command. See the *Command Reference* for information on these commands.

**FUNCTION MAPPING** *function-mapping-name*
> Identifies the function mapping 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. However, they can be disabled. For an example, see Example 3 in "CREATE FUNCTION MAPPING" on page 603.

> Packages having a dependency on a dropped function mapping are invalidated.

**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 required by the system for a primary key or unique constraint or for a replicated summary table (SQLSTATE 42917). The specified index or index specification is deleted.

**DROP**

> Packages having a dependency on a dropped index or index specification are invalidated.

> **NICKNAME** *nickname*
> Identifies the nickname 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 index specifications that are dependent on the nickname are dropped. Any views dependent on the nickname are marked inoperative. Any packages depending on the dropped index specifications or inoperative views are invalidated. The data source table that the nickname references is not affected.

> **NODEGROUP** *nodegroup-name*
> Identifies the nodegroup that is to be dropped. *nodegroup-name* must identify a nodegroup that is described in the catalog (SQLSTATE 42704). This is a one-part name.
>
> Dropping a nodegroup drops all table spaces defined in the nodegroup. All existing database objects with dependencies on the tables in the table spaces (such as packages, referential constraints, etc.) are dropped or invalidated (as appropriate), and dependent views and triggers are made inoperative.
>
> System defined nodegroups cannot be dropped (SQLSTATE 42832).
>
> If a DROP NODEGROUP is issued against a nodegroup that is currently undergoing a data redistribution, the DROP NODEGROUP operation fails an error is returned (SQLSTATE 55038). However, a partially redistributed nodegroup can be dropped. A nodegroup can become partially redistributed if a REDISTRIBUTE NODEGROUP command does not execute to completion. This can happen if it gets interrupted by either an error or a force application all command[85] .

> **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. All privileges on the package are also deleted.

> **PROCEDURE**
> Identifies an instance of a stored procedure that is to be dropped. The procedure instance specified must be a stored procedure described in the catalog.

---

85. For a partially redistributed nodegroup, the REBALANCE_PMAP_ID in the SYSCAT.NODEGROUPS catalog is not 1.

There are several different ways available to identify the procedure instance:

**PROCEDURE** *procedure-name*
> Identifies the particular procedure, and is valid only if there is exactly one procedure instance with the *procedure-name* in the schema. The procedure thus identified may have any number of parameters defined for it. If no procedure by this name exists in the named or implied schema, an error (SQLSTATE 42704) is raised. 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 there is more than one specific instance of the procedure in the named or implied schema, an error (SQLSTATE 42854) is raised.

**PROCEDURE** *procedure-name* **(***data-type***,...)**
> Provides the procedure signature, which uniquely identifies the procedure to be dropped. The procedure selection algorithm is **not** used.

> *procedure-name*
> > Gives the procedure name of the procedure 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.

> *(data-type,...)*
> > Must match the data types that were specified on the CREATE PROCEDURE statement in the corresponding position. The number of data types, and the logical concatenation of the data types is used to identify the specific procedure instance which is to be dropped.

> > If the *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 may be coded to indicate that these attributes are to be ignored when looking for a data type match.

> > FLOAT() cannot be used (SQLSTATE 42601) since 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.

> A type of FLOAT(n) does not need to match the defined value for n since 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 named or implied schema, an error (SQLSTATE 42883) is raised.

**SPECIFIC PROCEDURE** *specific-name*
Identifies the particular stored procedure that is to be dropped, 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/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.

**SCHEMA** *schema-name* **RESTRICT**
Identifies the schema that is to be dropped. The *schema-name* must identify a schema that is described in the catalog (SQLSTATE 42704). 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).

**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.

**TABLE** *table-name*
Identifies the base table or summary table that is to be dropped. The *table-name* must identify a table that is described in the catalog (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 specified table is deleted from the database.

All indexes, primary keys, foreign keys, and check constraints referencing the table are dropped. All views and triggers[86] that reference the table are made inoperative. All packages depending on any object dropped or marked inoperative will be invalidated. This includes packages dependent on any supertables above the subtable in the hierarchy. Any reference columns for which the dropped table is defined as the scope of the reference become unscoped.

All files that are linked through any DATALINK columns are unlinked. The unlink operation is performed asynchronously so the files may not be immediately available for other operations.

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.

**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,summary 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.

All files that are linked through any DATALINK columns are unlinked. The unlink operation is performed asynchronously so the files may not be immediately available for other operations.

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.

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

---

86. This includes both the table referenced in the ON clause of the CREATE TRIGGER statement and all tables referenced within the triggered SQL statements.

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). System table spaces cannot be dropped (SQLSTATE 42832). A temporary table space can not be dropped (SQLSTATE 55026) if it is the only temporary table space that exists in the database.

Dropping a table space drops all objects defined in the table space. All existing database objects with dependencies on the table space, such as packages, referential constraints, etc. are dropped or invalidated (as appropriate), and dependent views and triggers are made inoperative.

Containers created by the user are not deleted. Any directories in the path of the container name that were created by the database manager on CREATE TABLESPACE will be deleted. All containers that are below the database directory are deleted. For SMS table spaces, the deletions occur after all connections are disconnected or DEACTIVATE DATABASE command is issued.

**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. See the "Notes" section in "CREATE TRIGGER" on page 690 concerning the creation of triggers (which follows the same rules).

**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. If DISTINCT is specified, then the *type-name* must identify a distinct type described in the catalog. The type is not dropped (SQLSTATE 42893) if any of the following are true.

- The type is a distinct type 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 a data type of a typed table.
- 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 that cannot be dropped.
- The type is used in a check constraint, trigger or view definition.

If the user-defined type can be dropped, then for every function, F, 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 FUNCTION F
```

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).

**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.

**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.

**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 summary 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.

**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.

**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.

## Rules

*Dependencies:* Table 24 on page 761 shows the dependencies[87]that objects have on each other. 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/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.

---

87. Not all dependencies are explicitly recorded in the catalog. For example, there is no record of which constraints a package has a dependency on.

Some DROP statement parameters and objects are not shown in Table 24 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, bufferpool, partitioning 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 24. Dependencies

| Statement ↓ \ Object Type → | CONSTRAINT | FUNCTION | FUNC MAPPING | INDEX | NICKNAME | NODEGROUP | PACKAGE | SERVER | TABLE | TABLESPACE | TRIGGER | TYPE | TYPE MAPPING | USER MAPPING | VIEW |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ALTER NICKNAME ALTER COLUMN (OPTIONS(DROP)) | - | - | - | - | - | - | A | - | - | - | - | - | - | - | - |
| ALTER SERVER OPTIONS (DROP) | - | - | - | - | - | - | A | - | - | - | - | - | - | - | - |
| ALTER TABLE DROP CONSTRAINT | C | - | - | - | - | - | A[1] | - | - | - | - | - | - | - | - |
| ALTER TABLE DROP PARTITIONING KEY | - | - | - | - | - | R[20] | A[1] | - | - | - | - | - | - | - | - |
| DROP ALIAS | - | - | - | - | - | - | A[3] | - | R[3] | - | X[3] | - | - | - | X[3] |
| DROP BUFFERPOOL | - | - | - | - | - | - | - | - | - | R | - | - | - | - | - |
| DROP FUNCTION | R | R[7] | R | - | - | - | X | - | R | - | R | - | - | - | R |
| DROP FUNCTION MAPPING | - | - | - | - | - | - | A | - | - | - | - | - | - | - | - |
| DROP INDEX | R | - | - | - | - | - | A | - | - | - | - | - | - | - | R[17] |
| DROP NICKNAME | - | - | - | C | - | - | A | - | - | - | - | - | - | - | X[16] |

## DROP

*Table 24. Dependencies  (continued)*

| Statement ↓ / Object Type → | CONSTRAINT | FUNCTION | FUNC MAPPING | INDEX | NICKNAME | NODEGROUP | PACKAGE | SERVER | TABLE | TABLESPACE | TRIGGER | TYPE | TYPE MAPPING | USER MAPPING | VIEW |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DROP NODEGROUP | - | - | - | - | - | - | - | - | - | C | - | - | - | - | - |
| DROP SERVER | - | - | C[19] | - | C | - | A | - | - | - | - | - | C[19] | C | - |
| DROP TABLE | C | - | - | C | - | - | A[9] | - | RC[11] | - | X[16] | - | - | - | X[16] |
| DROP TABLE HIERARCHY | C | - | - | C | - | - | A[9] | - | RC[11] | - | X[16] | - | - | - | X[16] |
| DROP TABLESPACE | - | - | - | C[6] | - | - | - | - | CR[6] | - | - | - | - | - | - |
| DROP TRIGGER | - | - | - | - | - | - | A[1] | - | - | - | - | - | - | - | - |
| DROP TYPE | R[13] | C[5] | - | - | - | - | A[12] | - | R[18] | - | R[13] | R[4] | - | - | R[14] |
| DROP VIEW | - | - | - | - | - | - | A[2] | - | - | - | X[16] | - | - | - | X[15] |
| DROP VIEW HIERACHY | - | - | - | - | - | - | A[2] | - | - | - | X[16] | - | - | - | X[16] |
| DROP WRAPPER | - | - | C | - | - | - | - | C | - | - | - | - | C | - | - |
| REVOKE a privilege[10] | - | - | - | - | - | - | A[1] | - | CX[8] | - | X | - | - | - | X[8] |

**1**    This dependency is implicit in depending on a table with these constraints, triggers, or a partitioning 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, summary 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, then 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.

762

**4**  A user-defined type can depend on another user-defined type if the depending user-defined type:

- names a user-defined type as the data type of an attribute
- names a user-defined structured type within a reference type as the data type of an attribute
- has a user-defined type as a supertype.

**5**  Dropping a data type cascades to drop the functions that use that data type. Dropping of these functions will not be prevented by the fact that they depend on each other.

**6**  Dropping a table space or a list of table spaces causes all the tables that are completely contained within the given tablespace or list to be dropped. However, if a table spans table spaces (indexes or long columns in different table spaces) and those table spaces are not in the list being dropped then the table space(s) 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.

**8**  Only loss of SELECT privilege will cause a summary 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 summary table is dependent on the underlying table or tables specified in the fullselect of the table definition.

A subtable is dependent on its supertable up to the root table. A supertable cannot be dropped until all its subtables are dropped.

**12** A package can depend on structured types as a result of using the TYPE predicate. The package has a dependency on the subtypes of each structured type specified in the right side of the TYPE predicate. Dropping or creating a structured type that alters the subtypes on which the package is dependent causes invalidation.

**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 because:
- it is used as the type of a column
- it is used as the type of the table
- it is used as an attribute of a type of the table
- it is 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.

**19** Dropping a server cascades to drop the function mappings and type mappings created for that named server.

**20** If the partitioning key is defined on a table in a multiple partition nodegroup, the partitioning key is required.

### 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.

- For any dropped table that includes currently linked files through DATALINK columns, the files are unlinked, and will be either restored or deleted, depending on the datalink column definition.

- If a table containing a DATALINK column is dropped while any DB2 Data Links Managers configured to the database are unavailable, either through DROP TABLE or DROP TABLESPACE, then the operation will fail (SQLSTATE 57050).

## Examples

*Example 1:* Drop table TDEPT.

```
DROP TABLE TDEPT
```

*Example 2:* Drop the view VDEPT.

```
DROP VIEW VDEPT
```

*Example 3:* The authorization ID HEDGES attempts to drop an alias.

```
DROP ALIAS A1
```

The alias HEDGES.A1 is removed from the catalogs.

*Example 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).

*Example 5:*

Drop the BUSINESS_OPS nodegroup. To drop the nodegroup, the two table spaces (ACCOUNTING and PLANS) in the nodegroup must first be dropped.

```
DROP TABLESPACE ACCOUNTING
DROP TABLESPACE PLANS
DROP NODEGROUP BUSINESS_OPS
```

*Example 6:* Pellow wants to drop the CENTRE function, which he created in his PELLOW schema, using the signature to identify the function instance to be dropped.

```
DROP FUNCTION CENTRE (INT,FLOAT)
```

*Example 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
```

*Example 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
```

*Example 9:* Drop the trigger SALARY_BONUS, which caused employees under a specified condition to receive a bonus to their salary.

```
DROP TRIGGER SALARY_BONUS
```

*Example 10:* Drop the distinct data type named shoesize, if it is not currently in use.

```
DROP DISTINCT TYPE SHOESIZE
```

*Example 11:* Drop the SMITHPAY event monitor.

```
DROP EVENT MONITOR SMITHPAY
```

*Example 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
```

*Example 13:* Macdonald wants to drop the DESTROY procedure, which he created in the EIGLER schema, using the specific name to identify the procedure instance to be dropped.

```
DROP SPECIFIC PROCEDURE  EIGLER.DESTROY
```

*Example 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
```

*Example 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
```

*Example 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
```

## 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

```
►►──END DECLARE SECTION──────────────────────────────────────────►◄
```

### 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 (see "BEGIN DECLARE SECTION" on page 482).

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.[88] Furthermore, the declaration of each variable must appear before the first reference to the variable.

Variables declared outside a declare section must not have the same name as variables declared within a declare section.

---

[88]. See "Rules" on page 482 for information on how host variables can be declared in REXX in the case of LOB locators and file reference variables.

**Example**

See "BEGIN DECLARE SECTION" on page 482 for examples that use the END DECLARE SECTION statement.

## EXECUTE

The EXECUTE statement executes a prepared SQL statement.

### Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

### Authorization

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. For statements where authorization checking is performed at statement preparation time (DML), no authorization is required to use this statement.

### Syntax

```
►►──EXECUTE──statement-name─────────────────────────────────────────►◄
                          │                    ,                 │
                          │              ┌──────◄──────┐         │
                          └─USING────────┴─host-variable─┴──────┘
                          └─USING DESCRIPTOR──descriptor-name─┘
```
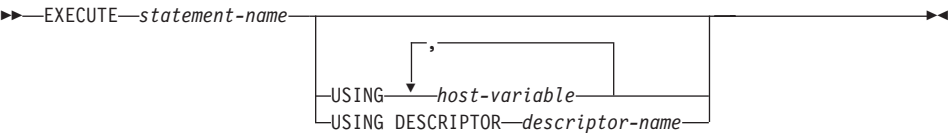
### Description

*statement-name*
> Identifies the prepared statement to be executed. The *statement-name* must identify a statement that was previously prepared and the prepared statement must not be a SELECT statement.

**USING**
> Introduces a list of host variables for which values are substituted for the parameter markers (question marks) in the prepared statement. (For an explanation of parameter markers, see "PREPARE" on page 824.) If the prepared statement includes parameter markers, USING must be used.

> *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 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.

**DESCRIPTOR** *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 input data needs to 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 more information, see "Appendix C. SQL Descriptor Area (SQLDA)" on page 933.

## Notes

- Before the prepared statement is executed, each parameter marker is effectively replaced by the value of 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. See "Rules" on page 825 for the rules affecting parameter markers.

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 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.

**EXECUTE**

 – 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. 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.

- *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 a PREPARE or EXECUTE IMMEDIATE statement. Once created, sections for dynamic SQL statements may be recreated by an implicit prepare of the statement performed 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 a 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 in the event that 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.

With the caching of dynamic SQL statements, once a statement has been created, it can be reused over multiple units of work without the need to prepare the statement again. The system will recompile the statement as required if environment changes occur.

The following events are examples of environment or data object changes which can cause cached dynamic statements to be implicitly prepared on the next PREPARE, EXECUTE, EXECUTE IMMEDIATE, or OPEN request:
– ALTER NICKNAME
– ALTER SERVER
– ALTER TABLE
– ALTER TABLESPACE
– CREATE FUNCTION
– CREATE FUNCTION MAPPING
– CREATE INDEX
– CREATE TABLE
– CREATE TEMPORARY TABLESPACE
– CREATE TRIGGER
– 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 FUNCTION PATH
– SET QUERY OPTIMIZATION
– 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 will not incur the cost of compiling the statement if the section is still valid. The cost and cardinality estimates for the current cached section will be returned. These values may differ from the values returned from any previous PREPARE for the same SQL statement.

  There will be no need to issue a PREPARE statement subsequent to a COMMIT or ROLLBACK statement.

– *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 will be invalidated.

– *COMMIT:* Dynamic SQL statements will not be invalidated but any locks acquired will be freed. Cursors not defined as WITH HOLD cursors will be closed and their locks freed. Open WITH HOLD cursors will hold onto their package and section locks to protect the active section during, and after, commit processing.

If an error occurs during an implicit prepare, an error will be returned for the request causing the implicit prepare (SQLSTATE 56098).

## Examples

*Example 1:* In this C example, an INSERT statement with parameter markers is prepared and executed. h1 - h4 are host variables that 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
```

## EXECUTE IMMEDIATE

The EXECUTE IMMEDIATE statement:

- Prepares an executable form of an SQL statement from a character string form of the statement.
- 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 be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

### Authorization

The authorization rules are those defined for the SQL statement specified by EXECUTE IMMEDIATE.

### Syntax

```
►►──EXECUTE IMMEDIATE──host-variable───────────────────────────►◄
```

### Description

*host-variable*
A host variable must be specified and it must identify a host variable that is described in the program in accordance with the rules for declaring character-string variables. It must be a character-string variable less than the maximum statement size of 65 535. Note that a CLOB(65535) can contain a maximum size statement but a VARCHAR can not.

The value of the identified host variable is called the statement string.

The statement string must be one of the following SQL statements:
- ALTER
- COMMENT ON
- COMMIT
- CREATE
- DELETE
- DROP
- GRANT
- INSERT
- LOCK TABLE

## EXECUTE IMMEDIATE

- REFRESH TABLE
- RENAME TABLE
- REVOKE
- ROLLBACK
- SET CURRENT DEGREE
- SET CURRENT EXPLAIN MODE
- SET CURRENT EXPLAIN SNAPSHOT
- SET CURRENT QUERY OPTIMIZATION
- SET CURRENT REFRESH AGE
- SET EVENT MONITOR STATE
- SET INTEGRITY
- SET PASSTHRU
- SET PATH
- SET SCHEMA
- SET SERVER OPTION
- 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 the CREATE TRIGGER statement which can contain a semi-colon (;) to separate triggered SQL statements.

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. See "Dynamic SQL Statement Caching" on page 772 for information.

### 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");
```

# EXECUTE IMMEDIATE

```
   .
   .
   .
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. (See "Appendix K. Explain Tables and Definitions" on page 1131 for information on the Explain tables and table definitions.)

An *explainable statement* is a DELETE, INSERT, SELECT, SELECT INTO, UPDATE, VALUES, or VALUES INTO SQL statement.

### 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 rules are those defined for the SQL statement specified in the EXPLAIN statement. For example, if a DELETE statement was used as the *explainable-sql-statement* (see statement syntax that follows), then the authorization rules for a DELETE statement would be applied when the DELETE statement is explained.

The authorization rules for static EXPLAIN statements are those rules that apply for static versions of the statement passed as the *explainable-sql-statement*. Dynamically prepared EXPLAIN statements use the authorization rules for the dynamic preparation of the statement provided for the *explainable-sql-statement* parameter.

The current authorization ID must have insert privilege on the Explain tables.

### Syntax

```
>>-EXPLAIN--+-PLAN SELECTION-+--+------------------+---------------->
            +-ALL------------+  +-FOR--+--SNAPSHOT--+
            |      (1)       |  '-WITH-'
            '-PLAN-----------'

>--+----------------------------+--+--------------------------------+-->
   '-SET--QUERYNO =--integer--'     '-SET--QUERYTAG =--string-constant-'
```
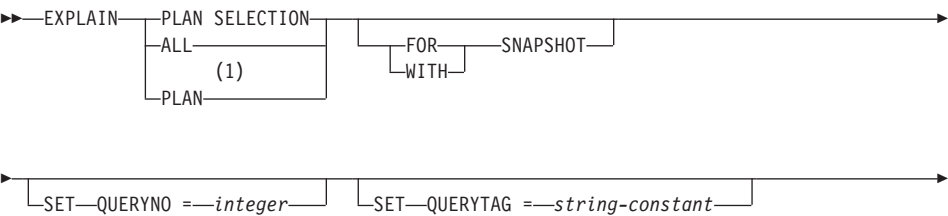
▶──FOR──*explainable-sql-statement*──────────────────────────────────────▶◀

**Notes:**

**1**    The PLAN option is supported only for syntax toleration of existing DB2
for MVS 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 SQL
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.

The Explain Snapshot information is intended for use with Visual Explain.

**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.

The Explain Snapshot information is intended for use with Visual Explain.

**default (neither FOR SNAPSHOT nor WITH SNAPSHOT specified)**
Puts Explain information into the Explain tables. No snapshot is taken for
use with Visual Explain.

**SET QUERYNO =** *integer*
Associates *integer*, via the QUERYNO column in the
EXPLAIN_STATEMENT table, with *explainable-sql-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.

## EXPLAIN

**SET QUERYTAG =** *string-constant*

Associates *string-constant*, via the QUERYTAG column in the EXPLAIN_STATEMENT table, with *explainable-sql-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 DELETE, INSERT, SELECT, SELECT INTO, 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.

### Notes

The following table shows the interaction of the snapshot keywords and the Explain information.

| Keyword Specified | Capture Explain Information? | Take Snapshot for Visual Explain? |
|---|---|---|
| none | Yes | No |
| FOR SNAPSHOT | No | Yes |
| WITH SNAPSHOT | Yes | Yes |

If neither the FOR SNAPSHOT nor WITH SNAPSHOT clause is specified, then no Explain snapshot is taken.

The Explain tables must be created by the user prior to the invocation of EXPLAIN. (See "Appendix K. Explain Tables and Definitions" on page 1131 for information on the Explain tables and table definitions.) The information generated by this statement is stored in these explain tables in the schema designated at the time the statement is compiled.

If any errors occur during the compilation of the *explainable-sql-statement* supplied, then no information is stored in the Explain tables.

The access plan generated for the *explainable-sql-statement* is not saved and thus, cannot be invoked at a later time. The Explain information for the *explainable-sql-statement* is inserted when the EXPLAIN statement itself is compiled.

For a static EXPLAIN SQL statement, the information is inserted into the Explain tables at bind time and during an explicit rebind (see REBIND in the *Command Reference*). 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/PREP options. Static EXPLAIN statements can be used to cause the Explain tables to be populated for one specific static SQL 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/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.

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 (SQLSTATE 00000)). 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 SQL 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.

## Examples

*Example 1:* Explain a simple SELECT statement and tag with QUERYNO = 13.

```
EXPLAIN PLAN SET QUERYNO = 13 FOR SELECT C1 FROM T1;
```

This statement is successful.

*Example 2:*

Explain a simple SELECT statement and tag with QUERYTAG = 'TEST13'.
```
EXPLAIN PLAN SELECTION SET QUERYTAG = 'TEST13'
  FOR SELECT C1 FROM T1;
```

This statement is successful.

*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;
```

This statement is successful.

*Example 4:*  Attempt to get Explain information when Explain tables do not exist.
```
EXPLAIN ALL FOR SELECT C1 FROM T1;
```

This statement would fail as the Explain tables have not been defined (SQLSTATE 42704).

## FETCH

The FETCH statement positions a cursor on the next row of its result table and assigns the values of that row to host 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.

### Authorization

See "DECLARE CURSOR" on page 731 for an explanation of the authorization required to use a cursor.

### Syntax

```
>>-FETCH---------------cursor-name---INTO----+--host-variable-----------+-><
           +-FROM-+                   +-USING DESCRIPTOR-descriptor-name-+
```

### Description

*cursor-name*
> Identifies the cursor to be used in the fetch operation. The *cursor-name* must identify a declared cursor as explained in "DECLARE CURSOR" on page 731. The DECLARE CURSOR statement must precede the FETCH statement in the source program. When the FETCH statement is executed, the cursor must be in the open state.
>
> 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 host variables.
>
> If the cursor is currently positioned before a row, it will be repositioned on that row, and values will be assigned to host variables as specified by INTO or USING.
>
> 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 host variables as specified by INTO or USING.

# FETCH

**INTO** *host-variable***, ...**
> Identifies one or more host variables that must be described in accordance with the rules for declaring host variables. The first value in the result row is assigned to the first host variable in the list, the second value to the second host variable, and so on. For LOB values in the select-list, the target can be a regular host variable (if it is large enough), a locator variable, or a file-reference variable.

**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 result columns need to be accommodated, there must be two SQLVAR entries for every select-list item (or column of the result table). See "Effect of DESCRIBE on the SQLDA" on page 939, which discusses SQLDOUBLED and LOB columns.
>
> SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN. For more information, see "Appendix C. SQL Descriptor Area (SQLDA)" on page 933.

The *n*th variable identified by the INTO clause or 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 the rules described in Chapter 3.. 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

- An open cursor has three possible positions:
  - Before a row
  - On a row
  - After the last row.
- 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. A cursor can only be on a row as a result of a FETCH statement.
- 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 for an error to occur that makes the state of the cursor unpredictable.
- Statement caching affects the behavior of an EXECUTE IMMEDIATE statement. See the "Notes" on page 771 for information.
- 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.

## 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
```

## FREE LOCATOR

The FREE LOCATOR statement removes the association between a locator
variable and its value.

### 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

```
              ┌─ , ◄──────────────┐
►►──FREE──LOCATOR──┴──variable-name──┴──────────────────────────────────►◄
```

### Description

**LOCATOR** *variable-name*, **...**

Identifies one or more 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 FETCH
statement or a SELECT INTO statement) and must not subsequently have
been freed (by a FREE LOCATOR statement); otherwise, an error is raised
(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.
```

## 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. Hence, at any time a partial event record is available 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 must include either SYSADM or DBADM authority (SQLSTATE 42502).

### Syntax

```
►►──FLUSH──EVENT──MONITOR──event-monitor-name─────────────────────►◄
                                              └─BUFFER─┘
```

### Description

*event-monitor-name*
: Name of the event monitor. This is a one-part name. It is an SQL 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.

## 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 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

To grant DBADM authority, SYSADM authority is required. To grant other authorities, either DBADM or SYSADM authority is required.

### Syntax

```
>>-GRANT--+-BINDADD-------------+--ON DATABASE------------------->
          |-CONNECT-------------|
          |-CREATETAB-----------|
          |-CREATE_NOT_FENCED---|
          |-IMPLICIT_SCHEMA-----|
          '-DBADM---------------'

>--TO--+----------+--authorization-name---------------------><
       |-USER-----|
       |-GROUP----|
       '-PUBLIC---'
```
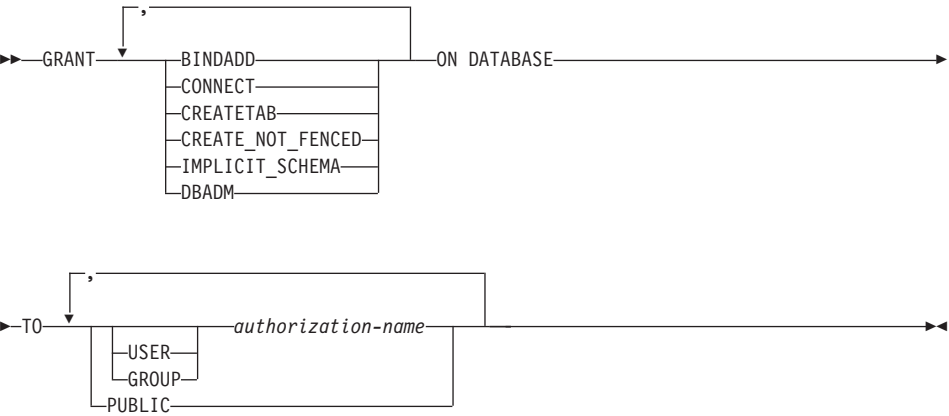
### Description

**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 subsequently 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 subsequently revoked.

There is no explicit authority required for view creation. A view can be created at any time if the authorization ID of the statement used to create the view has either CONTROL or SELECT privilege on each base table of the view.

**CREATE_NOT_FENCED**
Grants the authority to register functions that execute in the database manager's process. Care must be taken that functions so registered will not have adverse side effects (see the FENCED or NOT FENCED clause on page 562 for more information.)

Once a function has been registered as not fenced, it continues to run in this manner even if CREATE_NOT_FENCED is subsequently revoked.

**IMPLICIT_SCHEMA**
Grants the authority to implicitly create a schema.

**DBADM**
Grants the database administrator authority. A database administrator has all privileges against all objects in the database and may grant these privileges to others.

BINDADD, CONNECT, CREATETAB, CREATE_NOT_FENCED and IMPLICIT_SCHEMA are automatically granted to an *authorization-name* that is granted DBADM authority.

**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.

*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. (It is not possible to grant authorities to an *authorization-name* that is the same as the authorization ID of the GRANT statement.)

**PUBLIC**
Grants the authorities to all users. DBADM cannot be granted to PUBLIC.

## GRANT (Database Authorities)

### Rules

- If neither USER nor GROUP is specified, then
    If the authorization-name is defined in the operating system only as
    GROUP, then GROUP is assumed.
    If the authorization-name is defined in the operating system only as
    USER or if it is undefined, USER is assumed.
    If the authorization-name is defined in the operating system as both, or
    DCE authentication is used, an error (SQLSTATE 56092) is raised.

### 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. There is both a group and a user called D024 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).

## 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. 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 at least one of the following:
- DBADM authority
- SYSADM authority.

### Syntax

```
►►──GRANT──CONTROL──ON INDEX──index-name──────────────────────────────►

        ┌──────,──────────┐
►─TO──▼──┬───────────┬──authorization-name──┬──────────────────────►◄
         ├──USER──┤                         │
         └──GROUP─┘                         │
         └──PUBLIC─────────────────────────┘
```
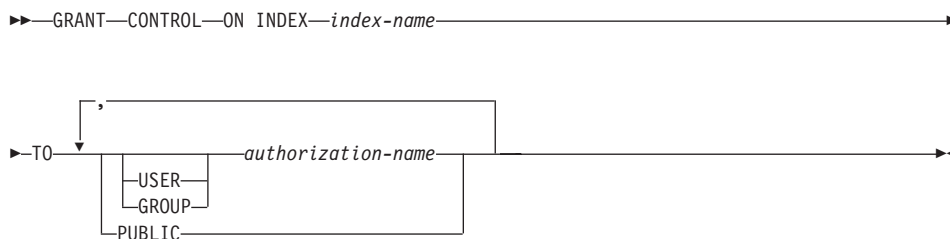
### Description

**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.

## GRANT (Index Privileges)

*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. (It is not possible to grant authorities to an *authorization-name* that is the same as the authorization ID of the GRANT statement.)

**PUBLIC**
> Grants the privileges to all users.

### Rules

- If neither USER nor GROUP is specified, then
  If the authorization-name is defined in the operating system only as GROUP, then GROUP is assumed.
  If the authorization-name is defined in the operating system only as USER or if it is undefined, USER is assumed.
  If the authorization-name is defined in the operating system as both, or DCE authentication is used, an error (SQLSTATE 56092) is raised.

### Example

```
GRANT CONTROL ON INDEX DEPTIDX TO USER USER4
```

## 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. 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 at least one of the following:
- CONTROL privilege on the referenced package
- SYSADM or DBADM authority.

To grant the CONTROL privilege, SYSADM or DBADM authority is required.

### Syntax

```
                      ,
         ┌──────────────────────┐
▶▶──GRANT──▼──┬──BIND──────┬──────┴──ON──PACKAGE──(2)──package-name──────▶
             ├──CONTROL────┤
             │        (1)  │
             └──EXECUTE────┘


         ┌──────────,──────────────┐
▶──TO──▼──┬────────────┬──authorization-name──┴──────────────────────▶◀
         │  ┌──USER──┐ │
         │  └──GROUP─┘ │
         └──PUBLIC─────┘
```

**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 is really a rebind privilege, because the package must have already been bound (by someone with BINDADD authority) to have existed at all.

## GRANT (Package Privileges)

In addition to the BIND privilege, the user must hold the necessary privileges on each table referenced by static DML statements contained in the 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.

**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.

**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.

*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. (It is not possible to grant authorities to an *authorization-name* that is the same as the authorization ID of the GRANT statement.)

**PUBLIC**
Grants the privileges to all users.

### Rules

- If neither USER nor GROUP is specified, then
  If the *authorization-name* is defined in the operating system only as GROUP, then GROUP is assumed.
  If the *authorization-name* is defined in the operating system only as USER or if it is undefined, USER is assumed.
  If the *authorization-name* is defined in the operating system as both, or DCE authentication is used, an error (SQLSTATE 56092) is raised.

## 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 (Schema Privileges)

This form of the GRANT statement grants privileges 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. 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 at least one of the following:
- the WITH GRANT OPTION for each identified privilege
- SYSADM or DBADM authority

### Syntax
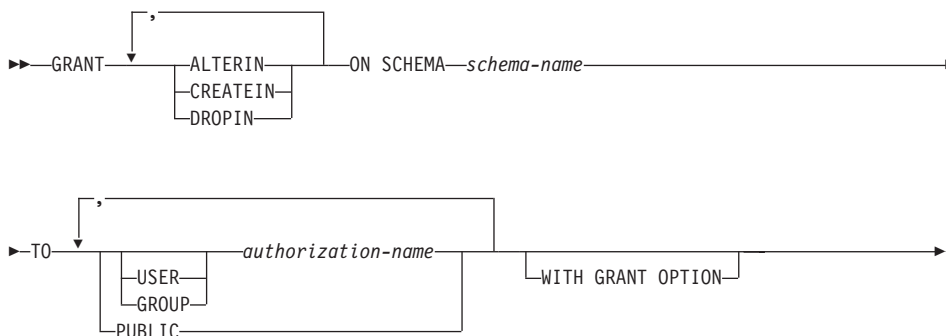


### Description

**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.

**DROPIN**
Grants the privilege to drop all objects in the schema. The owner of an explicitly created schema automatically receives DROPIN privilege.

**ON SCHEMA** *schema-name*
Identifies the schema on which the privileges are 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.
>
> *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. (It is not possible to grant authorities to an *authorization-name* that is the same as the authorization ID of the GRANT statement.)
>
> **PUBLIC**
> Grants the privileges to all users.

**WITH GRANT OPTION**
Allows the specified *authorization-name*s to GRANT the privileges to others.

If the WITH GRANT OPTION is omitted, the specified *authorization-name*s can only grant the privileges to others if they:
- have DBADM authority or
- received the ability to grant privileges from some other source.

## Rules

- If neither USER nor GROUP is specified, then
  If the authorization-name is defined in the operating system only as GROUP, then GROUP is assumed.
  If the authorization-name is defined in the operating system only as USER or if it is undefined, USER is assumed.
  If the authorization-name is defined in the operating system as both, or DCE authentication is used, an error (SQLSTATE 56092) is raised.

- Privileges cannot be granted on schema names SYSIBM, SYSCAT, SYSFUN and SYSSTAT by any user.

- In general, the GRANT statement will process the granting of privileges that the authorization ID of the statement is allowed to grant, returning a

## GRANT (Schema Privileges)

warning (SQLSTATE 01007) if one or more privileges was not granted. If no privileges were granted, an error is returned (SQLSTATE 42501).[89]

### Examples

*Example 1:* Grant USER2 to the ability to create objects in schema CORPDATA.

```
GRANT CREATEIN ON SCHEMA CORPDATA TO USER2
```

*Example 2:* Grant user BIGGUY the ability to create and drop objects in schema CORPDATA.

```
GRANT CREATEIN, DROPIN ON SCHEMA CORPDATA TO BIGGUY
```

---

89. 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.

## 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. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

### Authorization

The authorization ID of the statement must have either SYSADM or DBADM authority.

### Syntax

```
►►──GRANT PASSTHRU ON SERVER──server-name──TO─────────────────────►

    ┌─,────────────────────────────┐
►───▼──┬─────────┬──authorization-name─┴──────────────────────────►◄
       ├─USER──┤
       ├─GROUP─┤
       └─PUBLIC────────────┘
```

### 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.

> *authorization-name,...*
> > Lists the authorization IDs of one or more users or groups.

## GRANT (Server Privileges)

> The list of authorization IDs cannot include the authorization ID of
> the user issuing the statement. (It is not possible to grant authorities
> to an *authorization-name* that is the same as the authorization ID of the
> GRANT statement.)

> **PUBLIC**
> Grants to all users the privilege to pass through to *server-name.*

### 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 (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. 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 at least one of the following:
- 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.
- SYSADM or DBADM authority.

To grant the CONTROL privilege, SYSADM or DBADM authority is required.

To grant privileges on catalog tables and views, either SYSADM or DBADM authority is required.

### Syntax

## GRANT (Table, View or Nickname Privileges)

```
                   ┌─TABLE─┐                        ┌────────,────────┐
►──ON──┼───────┼──┬─table-name──┬──TO──▼──┬─USER──┬──authorization-name──┬──►
                  │             (1)       │        ├─GROUP─┤                │
                  ├─view-name───┤         │        └PUBLIC─────────────────┘
                  │             (2)       │
                  └─nickname────┘
►──┬──────────────────────┬──►◄
   └─WITH GRANT OPTION─────┘
```
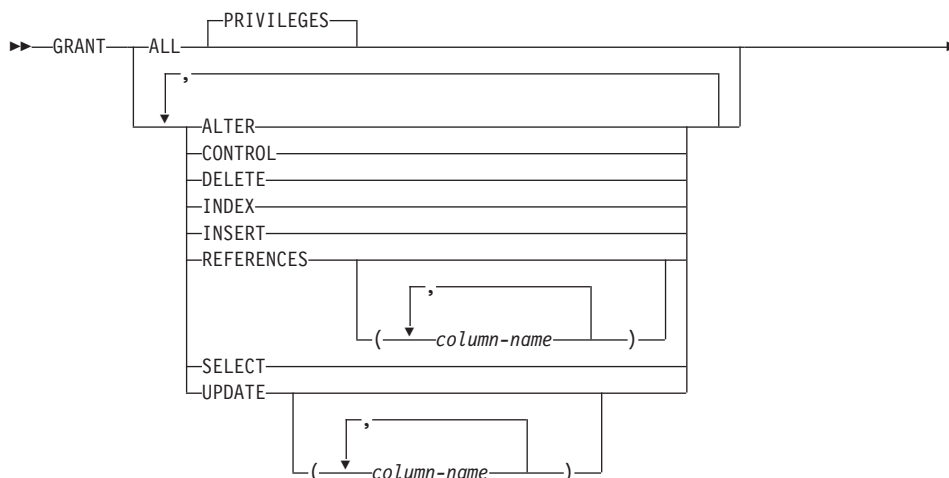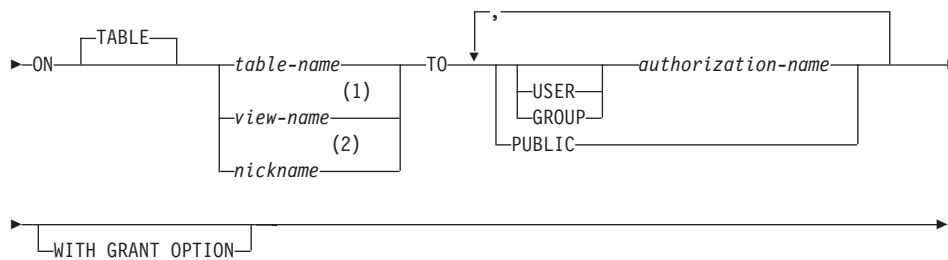
**Notes:**

**1**  ALTER, INDEX, and REFERENCES privileges are not applicable to views.

**2**  DELETE, INSERT, SELECT, and UPDATE privileges are not applicable to nicknames.

## 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 DBADM or SYSADM 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. For more information on the authorization required to create or drop a primary key or a unique constraint, see "ALTER TABLE" on page 449.
- 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.

# GRANT (Table, View or Nickname Privileges)

- Add or change a comment on a base table, a view, 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 above 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 someone with SYSADM or DBADM authority.

- The ability to execute the RUNSTATS utility on the table and indexes. See the *Command Reference* for information on RUNSTATS.
- The ability to issue SET INTEGRITY statement on the base table or summary table.

The definer of a base table, summary 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. 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.

## GRANT (Table, View or Nickname Privileges)

If the authorization ID of the statement has one of:
- DBADM or SYSADM 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. For more information on the authorization required to create or drop a foreign key, see "ALTER TABLE" on page 449.

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. See the *Command Reference* for information on EXPORT.

**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:
- DBADM or SYSADM 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 summary table (SQLSTATE 51024).

**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.
>
> *authorization-name,...*
> Lists the authorization IDs of one or more users or groups.[90]
>
> A privilege granted to a group is not used for authorization checking on static DML statements in a package. Nor is it used when checking authorization on a base table while processing a CREATE VIEW statement.
>
> In DB2 Universal Database, 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 all users.[91]

---

90. Restrictions in previous versions on grants to authorization ID of the user issuing the statement have been removed.

91. Restrictions in previous versions on the use of privileges granted to PUBLIC for static SQL statements and CREATE VIEW statements have been removed.

**GRANT (Table, View or Nickname Privileges)**

> **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

- If neither USER nor GROUP is specified, then
  If the *authorization-name* is defined in the operating system only as GROUP, then GROUP is assumed.
  If the *authorization-name* is defined in the operating system only as USER or if it is undefined, USER is assumed.
  If the *authorization-name* is defined in the operating system as both, or DCE authentication is used, an error (SQLSTATE 56092) is raised.

- 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).[92] If CONTROL privilege is specified, privileges will only be granted if the authorization ID of the statement has SYSADM or DBADM 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.

- DELETE, INSERT, SELECT, and UPDATE privileges are not defined for nicknames since operations on nicknames depend on the privileges of the authorization ID used at the data source when the statement referencing the nickname is processed.

## Examples

*Example 1:* Grant all privileges on the table WESTERN_CR to PUBLIC.

---

92. 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.

```
GRANT ALL ON WESTERN_CR
  TO PUBLIC
```

*Example 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
```

*Example 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
```

*Example 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
```

*Example 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
```

*Example 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.

*Example 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

# GRANT (Table, View or Nickname Privileges)

any privilege on CALENDAR including INSERT and SELECT as required. FRANK cannot grant CONTROL on CALENDAR to other users unless he has SYSADM or DBADM authority.

*Example 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
```

## 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

```
►►──INCLUDE──┬─SQLCA─┬─────────────────────────────────────────────►◄
             ├─SQLDA─┤
             └─name──┘
```
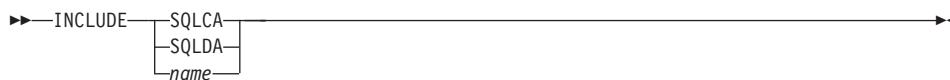
### Description

**SQLCA**
Indicates the description of an SQL communication area (SQLCA) is to be included. For a description of the SQLCA, see "Appendix B. SQL Communications (SQLCA)" on page 927.

**SQLDA**
Indicates the description of an SQL descriptor area (SQLDA) is to be included. For a description of the SQLDA, see "Appendix C. SQL Descriptor Area (SQLDA)" on page 933.

*name*
Identifies an external file containing text that is to be included in the source program being precompiled. It may be an SQL identifier without a filename extension or a literal in single quotes (' '). An SQL identifier assumes the filename extension of the source file being precompiled. If a filename extension is not provided by a literal in quotes then none is assumed.

For host language specific information, see the *Application Development Guide*.

### 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.

## INCLUDE

- The external source file must be written in the host language specified by the *name*. If it is greater than 18 characters or contains characters not allowed in an SQL identifier then it must be in single quotes. 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 or view. Inserting a row into a view also inserts the row into the table 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

To execute this statement, the privileges held by the authorization ID of the statement must include at least one of the following:
- INSERT privilege on the table or view where rows are to be inserted
- CONTROL privilege on the table or view where rows are to be inserted
- SYSADM or DBADM authority.

In addition, for each table or view 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:
- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM authority.

GROUP privileges are not checked for static INSERT statements.

### Syntax

```
►►──INSERT INTO──┬─table-name─┬──────────────────────────────────►
                 └─view-name──┘
                              ┌─────,─────────┐
                 └─(──▼─column-name──┴──)─┘
```
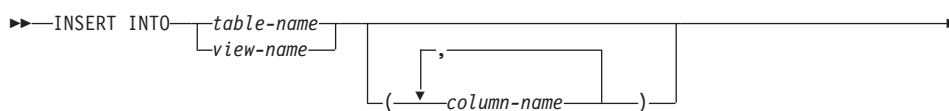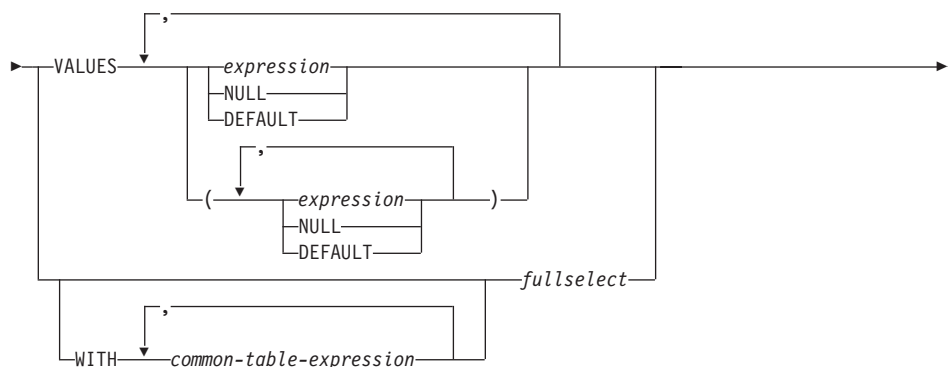
# INSERT



> **Note:** See "Chapter 5. Queries" on page 367 for the syntax of
> *common-table-expression* and *fullselect*.

## Description

**INTO** *table-name* or *view-name*
> Identifies the object of the insert operation. The name must identify a
> table or view that exists at the application server, but it must not identify
> a catalog table, a summary table, a view of a catalog table, or a read-only
> view.
>
> 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
> * A column derived from a nickname.
>
> 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.

**(***column-name***,...)**
> Specifies the columns for which insert values are provided. Each name
> must be an unqualified name that identifies a column of the table or view.
> The same column must not be identified more than once. A view column
> that cannot accept insert values must not be identified.
>
> Omission of the column list is an implicit specification of a list in which
> every column of the table 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.

The implicit column list is established at prepare time. Hence an INSERT statement embedded in an application program does not use any columns that might have been added to the table or view after prepare time.

**VALUES**

Introduces one or more rows of values to be inserted.

Each host variable named must be described in the program in accordance with the rules for declaring host variables.

The number of values for each row must equal the number of names in the column list. The first value is inserted in the first column in the list, the second value in the second column, and so on.

**expression**

An *expression* can be as defined in "Expressions" on page 146.

**NULL**

Specifies the null value and should only be specified for nullable columns.

**DEFAULT**

Specifies that the default value is to be used. The value that is inserted depends on how the column was defined, as follows:

- If the WITH DEFAULT clause is used, the default inserted is as defined for the column (see default-clause in "ALTER TABLE" on page 449).
- If the WITH DEFAULT clause or the NOT NULL clause is not used, the value inserted is NULL.
- If the NOT NULL clause is used and the WITH DEFAULT clause is not used or DEFAULT NULL is used, the DEFAULT keyword cannot be specified for that column (SQLSTATE 23502).

**WITH** *common-table-expression*

Defines a common table expression for use with the fullselect that follows. See "common-table-expression" on page 413 for an explanation of the *common-table-expression*.

*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.

## Rules

- *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.

- *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 the assignment rules described in Chapter 3..

- *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" on page 714.

- *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.

- *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 insert values.

- *Datalinks:* Insert statements that include DATALINK values will result in an attempt to link the file if a URL value is included (not empty string or blanks) and the column is defined with FILE LINK CONTROL. Errors in the DATALINK value or in linking the file will cause the insert to fail (SQLSTATE 428D1 or 57050).

## Notes

- After execution of an INSERT statement that is embedded within a program, the value of the third variable of the SQLERRD(3) portion of the SQLCA indicates the number of rows that were inserted. 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. DB2 assumes 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. See buffered insert in the *Application Development Guide* for further details.

## 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')
```

**INSERT**

*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);
```

## LOCK TABLE

The LOCK TABLE statement either prevents concurrent application processes from changing a table or prevents concurrent application processes from using 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.

### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:
* SELECT privilege on the table
* CONTROL privilege on the table
* SYSADM or DBADM authority.

### Syntax

```
►►──LOCK TABLE──table-name──IN──┬─SHARE─────┬──MODE────────────────────────────►◄
                                └─EXCLUSIVE─┘
```

### Description

*table-name*
> Identifies the table. The *table-name* must identify a table that exists at the application server, but it must not identify a catalog table. It cannot be a nickname (SQLSTATE 42809). If the *table-name* is a typed table, it must be the root table of the table hierarchy (SQLSTATE 428DR).

**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 the execution of the LOCK TABLE statement if a suitable lock already exists. The lock that prevents concurrent operations is held at least until the termination of the unit of work.

## LOCK TABLE

- In a partitioned database, a table lock is first acquired at the first partition in the nodegroup (the partition with the lowest number) and then at other partitions. If the LOCK TABLE statement is interrupted, the table may be locked on some partitions but not on others. If this occurs, either issue another LOCK TABLE statement to complete the locking on all partitions, or issue a COMMIT or ROLLBACK statement to release the current locks.
- This statement affects all partitions in the nodegroup.
- This statement cannot be used to lock tables that are referenced by nicknames.

### Example

Obtain a lock on the table EMP. Do not allow other programs either to read or update the table.

```
LOCK TABLE EMP IN EXCLUSIVE MODE
```

## 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.

### Authorization

See "DECLARE CURSOR" on page 731 for the authorization required to use a cursor.

### Syntax

```
►►──OPEN──cursor-name─────────────────────────────────────────────────►◄
                      │         ┌──────,──────┐              │
                      └─USING───▼──host-variable──┘
                      └─USING DESCRIPTOR──descriptor-name──┘
```
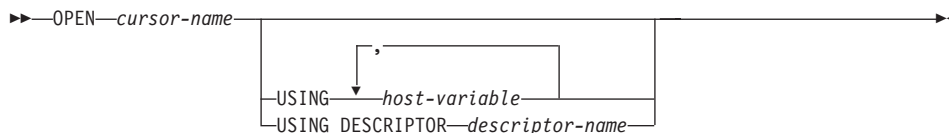
### Description

*cursor-name*
Names a cursor that is defined in a DECLARE CURSOR statement that was stated earlier in the program. When the OPEN statement is executed, the cursor 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 that SELECT statement, using the current values of any host variables specified in it or in 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".

**USING**

Introduces a list of host variables whose values are substituted for the parameter markers (question marks) of a prepared statement. (For an explanation of parameter markers, see "PREPARE" on page 824.) If the DECLARE CURSOR statement names a prepared statement that includes parameter markers, USING must be used. If the prepared statement does not include parameter markers, USING is ignored.

*host-variable*

Identifies a variable described in the program in accordance with the rules for declaring 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.

**DESCRIPTOR** *descriptor-name*

Identifies an SQLDA that must contain a valid description of host variables.

Before the OPEN 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 result columns need to be accommodated, there must be two SQLVAR entries for every select-list item (or column of the result table). See "Effect of DESCRIBE on the SQLDA" on page 939, which discusses SQLDOUBLED and LOB columns.

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN. For more information, see "Appendix C. SQL Descriptor Area (SQLDA)" on page 933.

## 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. See "Rules" on page 825 for the rules affecting parameter markers.
- 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 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. 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.

## 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.

- 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 temporary tables:* In some cases, the result table of a cursor is derived during the execution of FETCH statements. In other cases, the temporary table method is used instead. With this method the entire result table is transferred to a temporary table during the execution of the OPEN statement. When a temporary table is used, the results of a program can differ in these two 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.

  Conversely, if a temporary table 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. The *Application Development Guide* describes how locking can be used to control the effect of INSERT, UPDATE, and DELETE operations executed by concurrent units of work. Your result table can also be affected by operations executed by your own 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. See the "Notes" on page 771 for information.

## 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 the example below.)

```
EXEC SQL  BEGIN DECLARE SECTION;
   static short    hv_int;
   char            hv_vchar64[64];
   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;
```

## 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 be embedded only 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. For statements where authorization checking is performed at statement execution (DDL, GRANT, and REVOKE statements), no authorization is required to use the statement; however, the authorization is checked when the prepared statement is executed.

### Syntax

```
►►──PREPARE──statement-name──┬──────────────────────────┬──FROM──host-variable──────────►◄
                             └─INTO──descriptor-name─────┘
```

### 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.

**INTO**
    If INTO is used, and the PREPARE statement is successfully executed, information about the prepared statement is placed in the SQLDA specified by the descriptor-name.

    *descriptor-name*
        Is the name of an SQLDA.[93]

**FROM**
    Introduces the statement string. The statement string is the value of the specified host variable.

---

93. The DESCRIBE statement may be used as an alternative to this clause. See "DESCRIBE" on page 741.

*host-variable*

> Must identify a host variable that is described in the program in accordance with the rules for declaring character string variables. It must be a character-string variable (either fixed-length or varying-length).

## Rules

- *Rules for statement strings:* The statement string must be one of the following SQL statements:
  - ALTER
  - COMMENT ON
  - COMMIT
  - CREATE
  - DELETE
  - DROP
  - GRANT
  - INSERT
  - LOCK TABLE
  - REFRESH TABLE
  - RENAME TABLE
  - REVOKE
  - ROLLBACK
  - select-statement
  - SET CURRENT DEGREE
  - SET CURRENT EXPLAIN MODE
  - SET CURRENT EXPLAIN SNAPSHOT
  - SET CURRENT QUERY OPTIMIZATION
  - SET CURRENT REFRESH AGE
  - SET EVENT MONITOR STATE
  - SET INTEGRITY
  - SET PASSTHRU
  - SET PATH
  - SET SCHEMA
  - SET SERVER OPTION
  - UPDATE

  The statement string must not:
  - Be a SELECT statement with an INTO clause
  - Be a VALUES statement with an INTO clause
  - Begin with EXEC SQL and end with a statement terminator
  - Include references to host variables
  - Include comments.

- *Parameter Markers:* Although a statement string cannot include references to host variables, it may include *parameter markers*; those can be replaced by the values of host variables when the prepared statement is executed. A parameter marker is a question mark (?) that is declared where a host

variable could be stated if the statement string were a static SQL statement. For an explanation of how parameter markers are replaced by values, see "OPEN" on page 819 and "EXECUTE" on page 770.

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 above 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 parameters markers can be used in dynamic SQL statements in selected locations where host variables are supported. These locations and the resulting data type are found in Table 25. The locations are grouped in this table into expressions, predicates and functions to assist in determining applicability of an untyped parameter marker. When an untyped parameter marker is used in a function (including arithmetic operators, CONCAT and datetime operators) with an unqualified function name, the qualifier is set to 'SYSIBM' for the purposes of function resolution.

*Table 25. Untyped Parameter Marker Usage*

| Untyped Parameter Marker Location | Data Type |
|---|---|
| **Expressions (including select list, CASE and VALUES)** | |
| Alone in a select list | Error |

*Table 25. Untyped Parameter Marker Usage  (continued)*

| Untyped Parameter Marker Location | Data Type |
|---|---|
| Both operands of a single arithmetic operator, after considering operator precedence and order of operation rules.<br><br>Includes cases such as:<br>`? + ? + 10` | Error |
| One operand of a single operator in an arithmetic expression (not a datetime expression)<br><br>Includes cases such as:<br>`? + ? * 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 instance 'timecol + ?' or '? - datecol'). | Error |
| Both operands of a CONCAT operator | Error |
| One operand of a CONCAT operator where 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, then other is VARCHAR(254 - n). In all other cases the data type is VARCHAR(254). |
| One operand of a CONCAT operator where 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, then other is VARCHAR(127 - n). In all other cases the data type is VARCHAR(127). |
| One operand of a CONCAT operator where the other operand is a large object string. | Same as that of the other operand. |
| As a value on the right hand side of a SET clause of an UPDATE statement. | The data type of the column. If the column is defined as a user-defined distinct type, then it is the source data type of the user-defined distinct type. |
| The expression following CASE in a simple CASE expression | Error |

*Table 25. Untyped Parameter Marker Usage  (continued)*

| Untyped Parameter Marker Location | Data Type |
| --- | --- |
| At least one of the result-expressions in a CASE expression (both Simple and Searched) with the rest of the result-expressions either untyped parameter marker or NULL. | Error |
| Any or all expressions following WHEN in a simple CASE expression. | Result of applying the "Rules for Result Data Types" on page 105 to the expression following CASE and the expressions following WHEN that are not untyped parameter markers. |
| A result-expression in a CASE expression (both Simple and Searched) where at least one result-expression is not NULL and not an untyped parameter marker. | Result of applying the Rules for Result Data Types to all result-expressions that are other than NULL or untyped parameter markers. |
| Alone as a column-expression in a single-row VALUES clause that is not within an INSERT statement. | Error |
| 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 parameter markers. | Error |
| 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 parameter marker or NULL. | Result of applying the "Rules for Result Data Types" on page 105 on all operands that are other than untyped parameter markers. |
| 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, then it is the source data type of the user-defined distinct type. |
| 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, then it is the source data type of the user-defined distinct type. |
| As a value on the right side of a SET special register statement | The data type of the special register. |
| **Predicates** | |
| Both operands of a comparison operator | Error |

*Table 25. Untyped Parameter Marker Usage (continued)*

| Untyped Parameter Marker Location | Data Type |
|---|---|
| One operand of a comparison operator where the other operand other than an untyped parameter marker. | The data type of the other operand |
| All operands of a BETWEEN predicate | Error |
| Either<br><br>    1st and 2nd, or<br>    1st and 3rd<br><br>operands of a BETWEEN predicate | Same as that of the only non-parameter marker. |
| Remaining BETWEEN situations (i.e. one untyped parameter marker only) | Result of applying the "Rules for Result Data Types" on page 105 on all operands that are other than untyped parameter markers. |
| All operands of an IN predicate | Error |
| Both the 1st and 2nd operands of an IN predicate. | Result of applying the Rules for Result Data Types on all operands of the IN list (operands to the right of IN keyword) that are other than untyped parameter markers. |
| The 1st operand of an IN predicate where the right hand side is a fullselect. | Data type of the selected column |
| Any or all operands of the IN list of the IN predicate | Data type of the 1st operand (left hand side) while the length is result of applying the Rules for Result Data Types on all operands of the IN list (operands to the right of IN keyword) that are other than untyped parameter markers. |
| The 1st operand and zero or more operands in the IN list excluding the 1st operand of the IN list | Result of applying the Rules for Result Data Types on all operands of the IN list (operands to the right of IN keyword) that are other than untyped parameter markers. |
| 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 parameter marker. | Either VARCHAR(32672) or VARGRAPHIC(16336) depending on the data type of the first operand that is not an untyped parameter marker. |

*Table 25. Untyped Parameter Marker Usage  (continued)*

| Untyped Parameter Marker Location | Data Type |
|---|---|
| The pattern expression of the LIKE predicate when either the match expression or the escape expression is other than an untyped parameter marker. | Either VARCHAR(32672) or VARGRAPHIC(16336) depending on the data type of the first operand that is not an untyped parameter marker. If the data type of the match expression is BLOB, the data type of the pattern expression is assumed to be BLOB(32672). |
| The escape expression of the LIKE predicate when either the match expression or the pattern expression is other than an untyped parameter marker. | Either VARCHAR(2) or VARGRAPHIC(1) depending on the data type of the first operand that is not an untyped parameter marker. If the data type of the match expression or pattern expression is BLOB, the data type of the escape expression is assumed to be BLOB(1). |
| Operand of the NULL predicate | error |
| **Functions** | |
| All operands of COALESCE (also called VALUE) or NULLIF | Error |
| Any operand of COALESCE where at least one operand is other than an untyped parameter marker. | Result of applying the "Rules for Result Data Types" on page 105 on all operands that are other than untyped parameter markers. |
| An operand of NULLIF where the other operand is other than an untyped parameter marker. | The data type of the other operand |
| POSSTR (both operands) | Both operands are VARCHAR(32672). |
| POSSTR (one operand where the other operand is a character data type). | VARCHAR(32672). |
| POSSTR (one operand where the other operand is a graphic data type). | VARGRAPHIC(16336). |
| POSSTR (the search-string operand when the other operand is a BLOB). | BLOB(32672). |
| SUBSTR (1st operand) | VARCHAR(32672) |
| SUBSTR (2nd and 3rd operands) | INTEGER |
| The 1st operand of the TRANSLATE scalar function. | Error |
| The 2nd and 3rd operands of the TRANSLATE scalar function. | VARCHAR(32672) if the first operand is a character type. VARGRAPHIC(16336) if the first operand is a graphic type. |

*Table 25. Untyped Parameter Marker Usage  (continued)*

| Untyped Parameter Marker Location | Data Type |
| --- | --- |
| The 4th operand of the TRANSLATE scalar function. | VARCHAR(1) if the first operand is a character type. VARGRAPHIC(1) if the first operand is a graphic type. |
| The 2nd operand of the TIMESTAMP scalar function. | TIME |
| Unary minus | DOUBLE PRECISION |
| Unary plus | DOUBLE PRECISION |
| All other operands of all other scalar functions including user-defined functions. | Error |
| Operand of a column function | Error |

## Notes

- 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 ...** |
  | --- | --- |
  | **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.

- Statement caching affects repeated preparations. See the "Notes" on page 771 for information.

- See the *Application Development Guide* for examples of dynamic SQL statements in the supported host languages.

## 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.

## PREPARE

```
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
```

| | | |
|---|---|---|
| SQLDAID | 192 | |
| SQLDABC | 4 | |
| SQLN | 4 | |
| SQLD | | |
| SQLTYPE | 452 | |
| SQLLEN | 3 | |
| SQLDATA | | → G01 |
| SQLIND | | |
| SQLNAME | | |
| SQLTYPE | 449 | |
| SQLLEN | 29 | |
| SQLDATA | | → COMPLAINTS |
| SQLIND | | → 0 |
| SQLNAME | | |
| SQLTYPE | 453 | |
| SQLLEN | 6 | |
| SQLDATA | | |
| SQLIND | | → -1 |
| SQLNAME | | |
| SQLTYPE | 453 | |
| SQLLEN | 3 | |
| SQLDATA | | → A00 |
| SQLIND | | → 0 |
| SQLNAME | | |

To insert department number G01 named COMPLAINTS, which has no manager and reports to department A00, the structure INSERT_DA should have the above values before issuing the EXECUTE statement.

## REFRESH TABLE

The REFRESH TABLE statement refreshes the data in a summary 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:
- SYSADM or DBADM authority
- CONTROL privilege on the table.

### Syntax

```
►►──REFRESH──TABLE──table-name───────────────────────────────────────►◄
```

### Description

*table-name*
> Names a table.
>
> 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 summary tables defined with:
> - REFRESH IMMEDIATE
> - REFRESH DEFERRED

## RELEASE

The RELEASE 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

```
>>--RELEASE---server-name----(1)---------------------------------><
               |-host-variable-|
               |-CURRENT-------|
               |      -SQL-    |
               |-ALL--|   |    |
```

**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-justified 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.

**RELEASE**

> **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**
>> 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. The optional keyword SQL is included to be compatible with DB2/MVS SQL syntax.

### Notes

- If the RELEASE statement is successful, the identified connection is placed in the release-pending state and will therefore be destroyed during the next commit operation.

  If the RELEASE statement is unsuccessful, the connection state of the application process and the states of its connections are unchanged.
- If the current connection is in the release-pending state when a commit operation is performed, the destruction of that connection places the application process into the unconnected state. In this case, the next executed SQL statement should be CONNECT or SET CONNECTION.
- Type 1 CONNECT semantics do not preclude the use of RELEASE. The connection will be disconnected when the unit of work is committed.
- RELEASE does not close cursors, does not release any resources, and does not prevent further use of the connection.
- A rollback operation does not reset the state of a connection from release-pending to held.
- 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 option DISCONNECT(AUTOMATIC) or the option DISCONNECT(CONDITIONAL) is in effect. See "Options that Govern Distributed Unit of Work Semantics" on page 37 for information about the specification of the DISCONNECT option.

### 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;
```

## RENAME TABLE

The RENAME TABLE statement renames an existing 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. 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 either SYSADM or DBADM authority or CONTROL privilege.

### Syntax

```
>>--RENAME--┬──────┬--source-table-name--TO--target-identifier-------------><
            └TABLE─┘
```

### Description

*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 can be an alias identifying the table. It must not be the name of a catalog table (SQLSTATE 42832), a summary table, a typed table (SQLSTATE 42997), a nickname, or an object of other than table or alias (SQLSTATE 42809).

*target-identifier*
> Specifies the new name for the table without a schema name. The schema name of the *source-table-name* is used to qualify the new name for the table. The qualified name must *not* identify a table, view, or alias that already exists in the database (SQLSTATE 42710).

### Rules

The source table must not:
- be referenced in any existing view definitions or summary table definitions
- be referenced in any triggered SQL statements in existing triggers or be the subject table of an existing trigger
- have any check constraints
- be a parent or dependent table in any referential integrity constraints
- be the scope of any existing reference column.

An error (SQLSTATE 42986) is returned if the source table violates one or more of these conditions.

## Notes

- Catalog entries are updated to reflect the new table name.
- *All* authorizations associated with the source table name are *transferred* to the new table 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).
- Any packages that are dependent on the source table are invalidated.
- If an alias is used for the *source-table-name*, it must resolve to a table name. The table is renamed within the schema of this table. The alias is not changed by the RENAME statement and continues to refer to the old table name.
- A table with primary key or unique constraints may be renamed if none of the primary key or unique constraints are referenced by any foreign key.

## Example

Change the name of the EMP table to EMPLOYEE.

```
RENAME TABLE EMP TO EMPLOYEE
```

## 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 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 at least one of the following:
- DBADM authority
- SYSADM authority.

To revoke DBADM authority, SYSADM authority is required.

### Syntax

```
>>-REVOKE--+-+-BINDADD-----------+-+--ON DATABASE---------------->
           |                     |
           | ,                   |
           | +-CONNECT-----------+ |
           | +-CREATETAB---------+ |
           | +-CREATE_NOT_FENCED-+ |
           | +-IMPLICIT_SCHEMA---+ |
           | +-DBADM-------------+ |

>--FROM--+-+-------+--authorization-name-+-------------------->><
         |   |-USER--|                   |
         |   |-GROUP-|                   |
         |   +-PUBLIC----------------------+
```
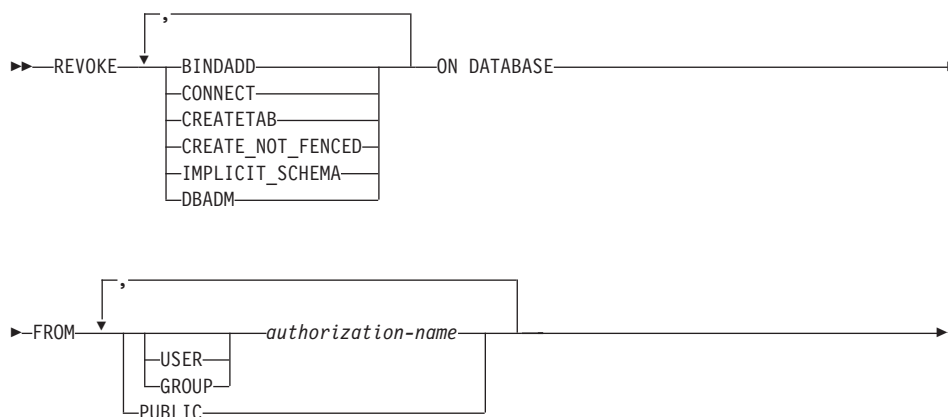
### Description

**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 his BINDADD authority is subsequently 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 subsequently granted the CONNECT authority again, 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 his CREATETAB authority is subsequently revoked.

The CREATETAB authority cannot be revoked from an *authorization-name* holding DBADM authority without also revoking the DBADM authority (SQLSTATE 42504).

**CREATE_NOT_FENCED**

Revokes the authority to register functions that execute in the database manager's process. However, once a function has been registered as not fenced, it continues to run in this manner even if CREATE_NOT_FENCED is subsequently revoked from the authorization ID that registered the function.

The CREATE_NOT_FENCED authority cannot be revoked from an *authorization-name* holding DBADM authority without also revoking the DBADM authority (SQLSTATE 42504).

**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.

**DBADM**

Revokes the DBADM authority.

DBADM authority cannot be revoked from PUBLIC (because it cannot be granted to PUBLIC).

**Revoking DBADM authority does not automatically revoke any privileges that were held by the authorization-name on objects in the database, nor does it revoke BINDADD, CONNECT, CREATETAB, IMPLICIT_SCHEMA, or CREATE_NOT_FENCED authority.**

**FROM**

Indicates from whom the authorities are revoked.

## REVOKE (Database Authorities)

    **USER**
        Specifies that the *authorization-name* identifies a user.

    **GROUP**
        Specifies that the *authorization-name* identifies a group name

    *authorization-name,...*
        Lists one or more authorization IDs.

        The authorization ID of the REVOKE statement itself cannot be used. (It is not possible to revoke the authorities from an *authorization-name* that is the same as the authorization ID of the REVOKE statement.)

    **PUBLIC**
        Revokes the authorities from PUBLIC.

### Rules

- If neither USER nor GROUP is specified, then:
  If all rows for the grantee in the SYSCAT.DBAUTH catalog view have a GRANTEETYPE of U, then USER will be assumed.
  If all rows have a GRANTEETYPE of G, then GROUP will be assumed.
  If some rows have U and some rows have G, then an error (SQLSTATE 56092) is raised.
  If DCE authentication is used, then an error is raised (SQLSTATE 56092).

### Notes

- Revoking a specific privilege does not necessarily revoke the ability to perform the action. A user may proceed with their task if other privileges are held by PUBLIC or a group, or if they have a higher level authority such as DBADM.

### 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. There are two rows 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 will occur (SQLSTATE 56092).

## 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. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

### Authorization

The authorization ID of the statement must hold either SYSADM or DBADM authority (SQLSTATE 42501).

### Syntax

```
►►──REVOKE──CONTROL──ON INDEX──index-name──────────────────────────────►

              ┌─────────────,─────────────┐
              │              ┌─USER──┐     │
►──FROM───────┴──────────────┼─GROUP─┼──authorization-name──┬──────────►◄
                             │       │                      │
                             └─PUBLIC────────────────────────┘
```
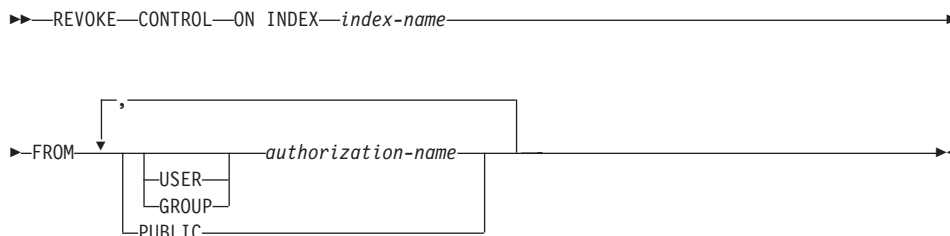
### Description

**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

    *authorization-name,...*
        Lists one or more authorization IDs.

## REVOKE (Index Privileges)

> The authorization ID of the REVOKE statement itself cannot be used.
> (It is not possible to revoke the privileges from an *authorization-name*
> that is the same as the authorization ID of the REVOKE statement.)

> **PUBLIC**
> Revokes the privileges from PUBLIC.

### Rules

- If neither USER nor GROUP is specified, then:
  If all rows for the grantee in the SYSCAT.INDEXAUTH catalog view
  have a GRANTEETYPE of U, then USER will be assumed.
  If all rows have a GRANTEETYPE of G, then GROUP will be assumed.
  If some rows have U and some rows have G, then an error (SQLSTATE
  56092) is raised.
  If DCE authentication is used, then an error is raised (SQLSTATE 56092).

### Notes

- Revoking a specific privilege does not necessarily revoke the ability to
  perform the action. A user may proceed with their task if other privileges
  are held by PUBLIC or a group, or if they have 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 USER4
```

*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 (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. 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 at least one of the following:
- CONTROL privilege on the referenced package
- SYSADM or DBADM authority.

To revoke the CONTROL privilege, SYSADM or DBADM authority are required.

### Syntax
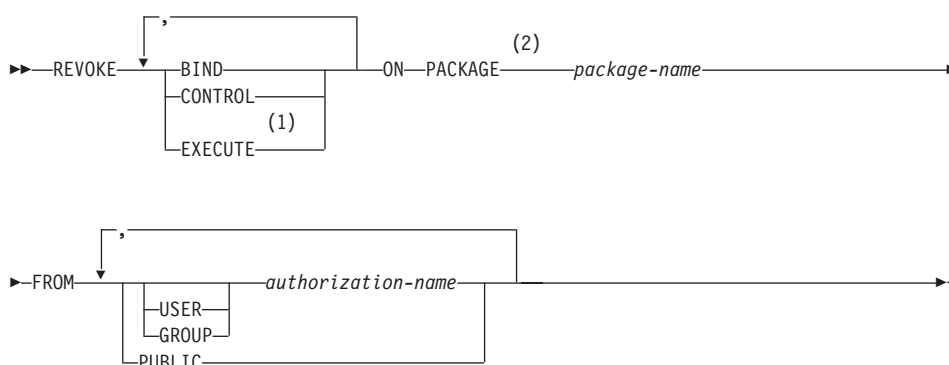


Notes:

**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 the referenced package.

**REVOKE (Package Privileges)**

     The BIND privileges 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 package on which privileges are 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.

    *authorization-name,...*
    Lists one or more authorization IDs.

    The authorization ID of the REVOKE statement itself cannot be used. (It is not possible to revoke the privileges from an *authorization-name* that is the same as the authorization ID of the REVOKE statement.)

    **PUBLIC**
    Revokes the privileges from PUBLIC.

## Rules

- If neither USER nor GROUP is specified, then:
  If all rows for the grantee in the SYSCAT.PACKAGEAUTH catalog view have a GRANTEETYPE of U, then USER will be assumed.
  If all rows have a GRANTEETYPE of G, then GROUP will be assumed.
  If some rows have U and some rows have G, then an error (SQLSTATE 56092) is raised.
  If DCE authentication is used, then an error is raised (SQLSTATE 56092).

REVOKE (Package Privileges)

## Notes

- Revoking a specific privilege does not necessarily revoke the ability to perform the action. A user may proceed with their task if other privileges are held by PUBLIC or a group, or if they have privileges such as ALTERIN 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 (Schema Privileges)

This form of the REVOKE statement revokes the privileges 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. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

### Authorization

The authorization ID of the statement must hold either SYSADM or DBADM authority (SQLSTATE 42501).

### Syntax
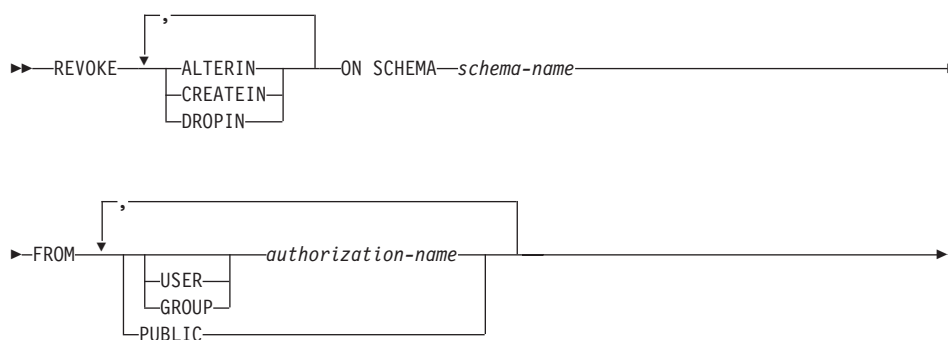


### Description

**ALTERIN**
    Revokes the privilege to alter or comment on objects in the schema.

**CREATEIN**
    Revokes the privilege to create objects in the schema.

**DROPIN**
    Revokes the privilege to drop objects in the schema.

**ON SCHEMA** *schema-name*
    Specifies the name of the schema on which privileges are 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

*authorization-name,...*
> Lists one or more authorization IDs.
>
> The authorization ID of the REVOKE statement itself cannot be used. (It is not possible to revoke the privileges from an *authorization-name* that is the same as the authorization ID of the REVOKE statement.)

**PUBLIC**
> Revokes the privileges from PUBLIC.

## Rules

- If neither USER nor GROUP is specified, then:
  > If all rows for the grantee in the SYSCAT.SCHEMAAUTH catalog view have a GRANTEETYPE of U, then USER will be assumed.
  > If all rows have a GRANTEETYPE of G, then GROUP will be assumed.
  > If some rows have U and some rows have G, then an error (SQLSTATE 56092) is raised.
  > If DCE authentication is used, then an error is raised (SQLSTATE 56092).

## Notes

- Revoking a specific privilege does not necessarily revoke the ability to perform the action. A user may proceed with their task if other privileges are held by PUBLIC or a group, or if they have 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 (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. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

### Authorization

The authorization ID of the statement must have SYSADM or DBADM authority.

### Syntax

```
►►──REVOKE PASSTHRU ON SERVER──server-name──FROM──────────────────────────────►

   ┌─────────────────────,──────────────────┐
►──┬─▼──┬──────┬──authorization-name──┬──────┴──────────────────────────────►◄
   │    ├─USER─┤                       │
   │    └─GROUP┘                       │
   └─PUBLIC────────────────────────────┘
```
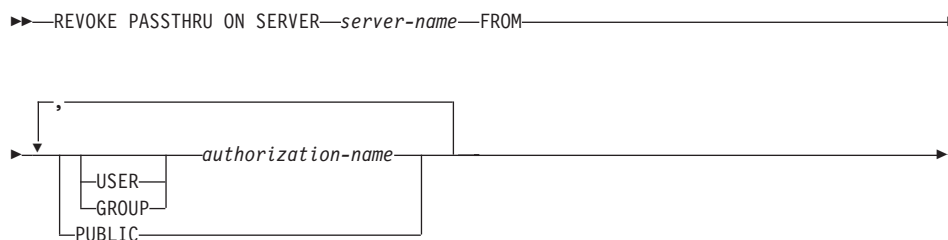
### Description

**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.

> *authorization-name,...*
>> Lists the authorization IDs of one or more users or groups.

The authorization ID of the REVOKE statement itself cannot be used. (It is not possible to revoke the privileges from an *authorization-name* that is the same as the authorization ID of the REVOKE statement.)

**PUBLIC**
Revokes from all users the privilege to pass through to *server-name.*

## 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 (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. 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 at least one of the following:
- SYSADM or DBADM authority
- CONTROL privilege on the referenced table, view, or nickname.

To revoke the CONTROL privilege, either SYSADM or DBADM authority is required.

To revoke the privileges on catalog tables and views, either SYSADM or DBADM authority is required.

### Syntax
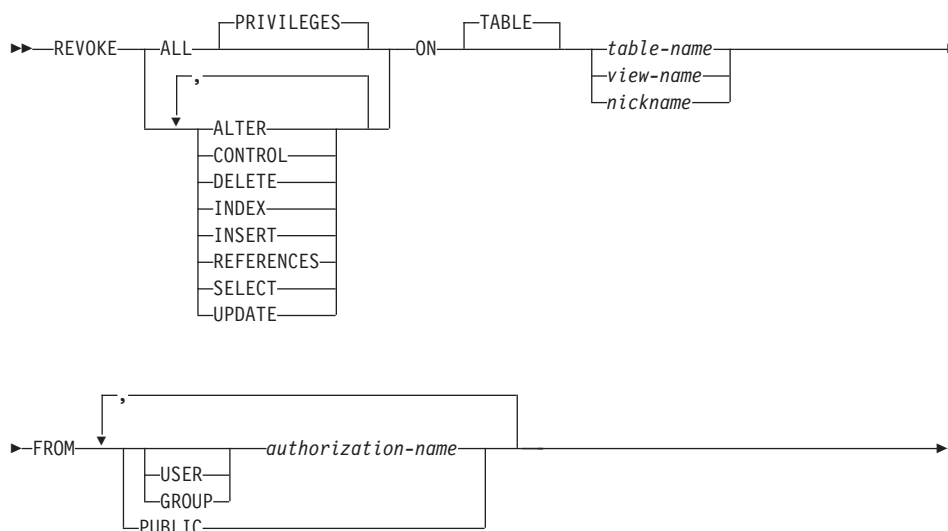
# REVOKE (Table, View or Nickname Privileges)

## Description

**ALL** or **ALL PRIVILEGES**

Revokes all privileges held by an authorization-name for the specified tables, views, or nicknames.

If ALL is not used, one or more of the keywords listed below 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 or updatable view.

**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 or updatable view, 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.

## REVOKE (Table, View or Nickname Privileges)

> Revoking SELECT privilege may cause some views to be marked inoperative. For information on inoperative views, see "Notes" on page 722.

**UPDATE**
Revokes the privilege to update rows in the table or updatable view. 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.

**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.

> *authorization-name,...*
> Lists one or more authorization IDs.
>
> The ID of the REVOKE statement itself cannot be used. (It is not possible to revoke the privileges from an *authorization-name* that is the same as the authorization ID of the REVOKE statement.)

> **PUBLIC**
> Revokes the privileges from PUBLIC.

### Rules

- If neither USER nor GROUP is specified, then:
  If all rows for the grantee in the SYSCAT.TABAUTH and SYSCAT.COLAUTH catalog views have a GRANTEETYPE of U, then USER will be assumed.
  If all rows have a GRANTEETYPE of G, then GROUP will be assumed.
  If some rows have U and some rows have G, then an error (SQLSTATE 56092) is raised.
  If DCE authentication is used, then an error is raised (SQLSTATE 56092).

### Notes

- If a privilege is revoked from the *authorization-name* used to create a view (this is called the view's DEFINER in SYSCAT.VIEWS), that privilege is also revoked from any dependent views.
- If the DEFINER 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)), then the view will be made inoperative (see the "Notes" section in "CREATE VIEW" on page 714 for information on inoperative views).

However, if a DBADM or SYSADM explicitly revokes all privileges on the view from the DEFINER, then the record of the DEFINER will not appear in SYSCAT.TABAUTH but nothing will happen to the view - it remains operative.

- Privileges on inoperative views cannot be revoked.
- All packages dependent upon an object for which a privilege is revoked are marked invalid. 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 for table T1 is revoked from USER1, then the package will be marked invalid. If SELECT authority is re-granted, or if the user holds DBADM authority, the package is successfully rebound when executed.

- 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. If one of these SELECT privileges is revoked, such packages are invalidated and such triggers or views are made inoperative.

- 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 may proceed with their task if other privileges are held by PUBLIC or a group, or if they have privileges such as ALTERIN on the schema of a table or a view.

- If the DEFINER of the summary table loses a SELECT privilege on a table on which the summary table definition depends, (or a table upon which the summary table definition depends is dropped), then the summary table will be made inoperative (see the "Notes" on page 672 for information on inoperative summary tables).

  However, if a DBADM or SYSADM explicitly revokes all privileges on the summary table from the DEFINER, then the record in SYSTABAUTH for the DEFINER will be deleted, but nothing will happen to the summary table - it remains operative.

- Revoking nickname privileges has no affect on data source object (table or view) privileges.

## REVOKE (Table, View or Nickname Privileges)

**Note:** "Rules" on page 760 lists the dependencies that objects such as tables and views can have on one another.

### 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
```

## REVOKE (Table, View or Nickname Privileges)

*Example 6:* Revoke user SHAWN's privilege to create an index specification on nickname ORAREM1.

```
REVOKE INDEX
  ON ORAREM1 FROM USER SHAWN
```

## ROLLBACK

The ROLLBACK statement is used to terminate a unit of work and back out 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

```
>>─ROLLBACK──┬──────┬────────────────────────────────────────><
             └─WORK─┘
```

### 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 issuing the ROLLBACK statement:
- SET CONNECTION,
- SET CURRENT DEGREE,
- SET CURRENT EXPLAIN MODE,
- SET CURRENT EXPLAIN SNAPSHOT,
- SET CURRENT PACKAGESET,
- SET CURRENT QUERY OPTIMIZATION,
- SET CURRENT REFRESH AGE,
- SET EVENT MONITOR STATE,
- SET PASSTHRU,
- SET PATH,
- SET SCHEMA,
- SET SERVER OPTION.

### Notes

- All locks held by the unit of work are released. 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.

- The termination of a unit of work is an implicit rollback if the program terminates abnormally due to a program check.
- Statement caching is affected by the rollback operation. See the "Notes" on page 771 for information.

## Example

Delete the alterations made since the last commit point or rollback.

```
ROLLBACK WORK
```

## SELECT

The SELECT statement is a form of query. It can be embedded in an application program or issued interactively. For detailed information, see "select-statement" on page 412 and "subselect" on page 368.

## 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, for each table or view referenced in the SELECT INTO statement, at least one of the following:
- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM authority.

GROUP privileges are not checked for static SELECT INTO statements.

### Syntax

```
►►─select-clause─INTO─┬─host-variable─┬─from-clause────────────────►
                      └──────,────────┘

►─┬────────────┬─┬────────────────┬─┬───────────────┬──────────►◄
  └where-clause┘ └group-by-clause─┘ └having-clause──┘
```
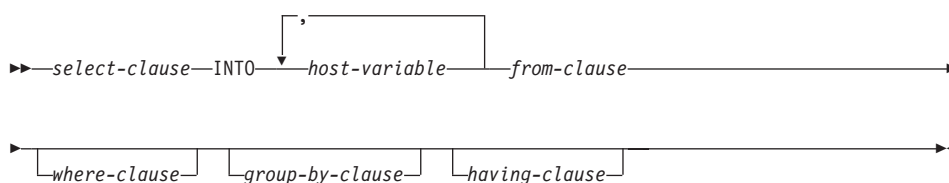
### Description

See "Chapter 5. Queries" on page 367 for a description of the *select-clause*, *from-clause*, *where-clause*, *group-by-clause*, and *having-clause*.

**INTO**
   Introduces a list of host variables.

   *host-variable*
      Identifies a variable that is described in the program under the rules for declaring host variables.

## SELECT INTO

The first value in the result row is assigned to the first variable in the list, the second value to the second variable, and so on. If the number of host variables is less than the number of column values, the value 'W' is assigned to the SQLWARN3 field of the SQLCA. (See "Appendix B. SQL Communications (SQLCA)" on page 927.)

Each assignment to a variable is made according to the rules described in "Assignments and Comparisons" on page 91. Assignments are made in sequence through the list.

If an error occurs, no value is assigned to any host variable.

### Examples

*Example 1:*  This C example puts the maximum salary in EMP 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 EMP, into host variables.

```
EXEC SQL SELECT * INTO :h1, :h2, :h3, :h4
  FROM EMP
  WHERE EMPNO = '528671';
```

## SET CONNECTION

The SET CONNECTION statement changes the state of a connection from dormant to current, making the specified location the current server. It 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

```
►►──SET CONNECTION──┬─server-name───┬──────────────────────────────►◄
                    └─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-justified 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*.

## SET CONNECTION

- 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 Type 1 CONNECT; details 514.

*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" on page 37) 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.

### Examples

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 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

No authorization is required to execute this statement.

### Syntax

```
►►──SET──CURRENT──DEGREE──┬──────────┬──┬─string-constant─┬──►◄
                          └────=────┘  └─host-variable───┘
```

### 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 intra-partition parallelism.

If the value of CURRENT DEGREE is a number when an SQL statement is dynamically prepared, the execution of that statement can involve intra-partition 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 intra-partition 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 intra-partition parallelism for static SQL statements can be controlled using the DEGREE option of the PREP or BIND command. Refer to the *Command Reference* for details on these commands.

The actual runtime degree of intra-partition parallelism will be the lower of:
- Maximum query degree (max_querydegree) configuration parameter
- Application runtime degree
- SQL statement compilation degree

The intra_parallel database manager configuration must be on to use intra-partition parallelism. If it is set to off, the value of this register will be ignored and the statement will not use intra-partition parallelism for the purpose of optimization (SQLSTATE 01623).

Some SQL statements cannot use intra-partition parallelism. See the *Administration Guide* for a description of degree of intra-partition parallelism and a list of restrictions.

### Example

*Example 1:* The following statement sets the CURRENT DEGREE to inhibit intra-partition parallelism.

```
SET CURRENT DEGREE = '1'
```

*Example 2:* The following statement sets the CURRENT DEGREE to allow intra-partition 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

No special authorization is required to execute this statement.

### Syntax

```
►►──SET CURRENT EXPLAIN MODE──┬───┬──┬─NO─────────────────┬──────►◄
                              └─=─┘  ├─YES────────────────┤
                                     ├─EXPLAIN────────────┤
                                     ├─RECOMMEND INDEXES───┤
                                     ├─EVALUATE INDEXES────┤
                                     └─host-variable──────┘
```
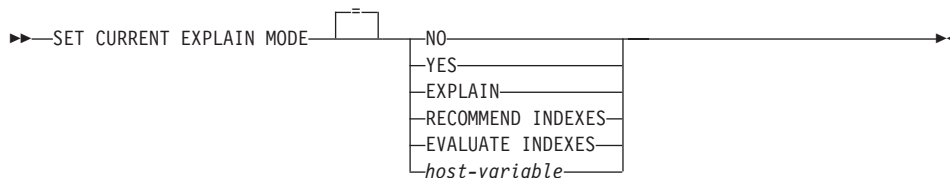
### Description

**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.

**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 indexes. The indexes to be

evaluated are read from the ADVISE_INDEX table, and must be marked with EVALUATE = Y. The optimizer generates virtual indexes based on the values from the catalogs. All queries that are executed in this explain mode will be compiled and optimized using estimated statistics based on the virtual indexes. The statements are not executed.

*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

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 runtime. If the value of the CURRENT EXPLAIN MODE register is not NO, then the value of the EXPLAIN bind option is ignored. For more information on the interaction between the EXPLAIN option and the CURRENT EXPLAIN MODE special register, see Table 127 on page 1166.

RECOMMEND INDEXES and EVALUATE INDEXES are special modes which can only be set with the SET CURRENT EXPLAIN MODE command. These modes cannot be set using PREP or BIND options, and they do not work with the SET CURRENT SNAPSHOT command.

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.

For further information, see the *Administration Guide*.

## Example

*Example 1:* 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

```
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. 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

No authorization is required to execute this statement.

### Syntax

```
>>─SET─CURRENT─EXPLAIN─SNAPSHOT──┬──────┬──┬─NO─────────────┬──><
                                 └──=───┘  ├─YES────────────┤
                                           ├─EXPLAIN────────┤
                                           └─host-variable──┘
```
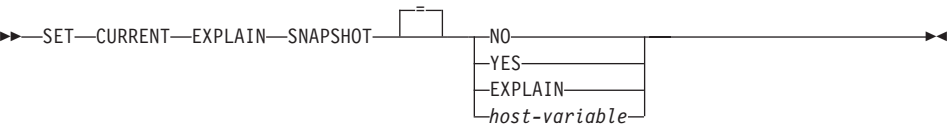
### Description

**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 (see "Appendix K. Explain Tables and Definitions" on page 1131).

   The EXPLAIN SNAPSHOT facility is intended for use with Visual Explain.

**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.

*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).

## SET CURRENT EXPLAIN SNAPSHOT

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

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 runtime. If the value of the CURRENT EXPLAIN SNAPSHOT register is not NO, then the EXPLSNAP option is ignored. For more information on the interaction between the EXPLSNAP option and the CURRENT EXPLAIN SNAPSHOT special register, see Table 128 on page 1167.

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.

For further information, see the *Administration Guide*.

## Example

*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 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

```
►►──SET──CURRENT PACKAGESET──┬──────┬──┬─string-constant─┬──────────────────►◄
                             └──=───┘  └─host-variable───┘
```

### Description

*string-constant*
    A character string constant with a maximum length of 8. If more than the
    maximum, it will be truncated at run time.

*host-variable*
    A variable of type CHAR or VARCHAR with a maximum length of 8. It
    cannot be set to null. If more than the maximum, it will be truncated at
    run time.

### 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. See the *Command Reference* for details.
- Unlike DB2 for MVS/ESA, the SET CURRENT PACKAGESET statement is
  implemented without support for a special register called CURRENT
  PACKAGESET.

## Example

Assume an application called TRYIT is precompiled by userid 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:

```
DB2 CONNECT TO SAMPLE USER PRODUSA
DB2 BIND TRYIT.BND DATETIME USA
DB2 CONNECT TO SAMPLE USER PRODEUR
DB2 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.

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. It 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

No authorization is required to execute this statement.

### Syntax
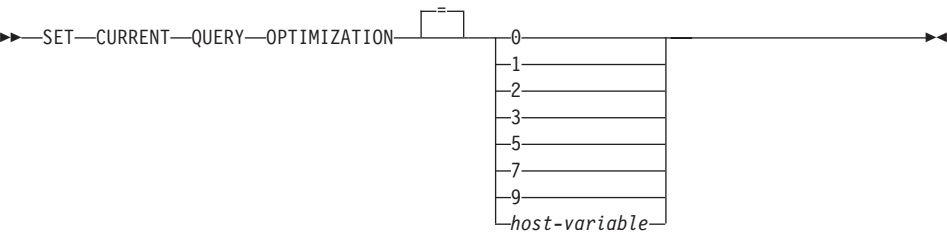


### Description

*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 (for details refer to the *Administration Guide* ).

| | |
|---|---|
| **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 above, especially for very complex queries. |
| **3** | Specifies that a moderate amount of optimization is performed to generate an access plan. |

874

| | |
|---|---|
| **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 summary 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 (SQLSTATE 01608). 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. For additional detail on the behavior associated with each query optimization class see *Administration Guide*.

- 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.

## SET CURRENT QUERY OPTIMIZATION

- 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 database configuration parameter, dft_queryopt.

- 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 desired class of optimization for statically bound statements. If QUERYOPT is not specified then, the default value specified by the database configuration parameter, dft_queryopt, is used. Refer to the BIND command in the *Command Reference* for details.

- 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. 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

No authorization is required to execute this statement.

### Syntax

```
>>─SET─CURRENT REFRESH AGE─┬──────┬─┬─numeric-constant─┬──────────><
                           └──=──┘ ├─ANY──────────────┤
                                   └─host-variable────┘
```
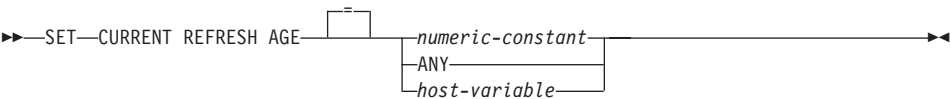
### Description

*numeric-constant*
> A DECIMAL(20,6) value representing a timestamp duration. The value must be 0 or 99 999 999 999 999 (the microseconds portion of the value is ignored and can therefore be any value).

> | **0** | Indicates that only summary tables defined with REFRESH IMMEDIATE may be used to optimize the processing of a query. |
> |---|---|
> | **99999999999999** | Indicates that any summary tables defined with REFRESH DEFERRED or REFRESH IMMEDIATE may be used to optimize the processing of a query. This value represents 9 999 years, 99 months, 99 days, 99 hours, 99 minutes, and 99 seconds. |

**ANY**
> This is a shorthand for 99999999999999.

*host-variable*
> A variable of type DECIMAL(20,6) or other type that is assignable to DECIMAL(20,6). 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 value of the host-variable must be 0 or 99 999 999 999 999.000000.

## SET CURRENT REFRESH AGE

### Notes

- The initial value of the CURRENT REFRESH AGE special register is zero.
- Setting the CURRENT REFRESH AGE special register to a value other than zero should be done with caution. By allowing a summary table that may not represent the values of the underlying base table to be used to optimize the processing of the query, the result of the query may NOT accurately represent the data in the underlying table. This may be reasonable when you know the underlying data has not changed or you are willing to accept the degree of error in the results based on your knowledge of the data.
- The CURRENT REFRESH AGE value of 99 999 999 999 999 cannot be used in timestamp arithmetic operations since the result would be outside the valid range of dates (SQLSTATE 22008).

### Examples

*Example 1:*  The following statement sets the CURRENT REFRESH AGE special register.

```
SET CURRENT REFRESH AGE ANY
```

*Example 2:*

The following example retrieves the current value of the CURRENT REFRESH AGE special register into the host variable called CURMAXAGE.

```
EXEC SQL VALUES (CURRENT REFRESH AGE) INTO  :CURMAXAGE;
```

The value would be 99999999999999.000000, set by the previous example.

## 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. However, if the bind option DYNAMICRULES BIND
applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

### Authorization

The authorization ID of the statement most hold either SYSADM or DBADM
authority (SQLSTATE 42815).

### Syntax

```
>>-SET-EVENT-MONITOR-event-monitor-name-STATE--+-=-+--+-0------------+-><
                                                       +-1------------+
                                                       '-host-variable-'
```

### 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 may 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). |

# SET EVENT MONITOR STATE

### Rules

- Although an unlimited number of event monitors may be defined, there is a limit of 32 event monitors that can be simultaneously active (SQLSTATE 54030).
- 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 an event monitor performs a reset of any counters associated with it.

### Example

The following example activates an event monitor called SMITHPAY.

```
SET EVENT MONITOR SMITHPAY STATE = 1
```

## SET INTEGRITY

The SET INTEGRITY statement is used to do one of the following:

- Turn off integrity checking for one or more tables. This includes check constraint and referential constraint checking, datalink integrity checking. If the table is a summary table with REFRESH IMMEDIATE, the immediate refreshing of the data is turned off. Note that this places the table(s) into a *check pending state* where only limited access by a restricted set of statements and commands is allowed. Primary key and unique constraints continue to be checked.
- Both turn the integrity checking back on for one or more tables and to carry out all the deferred checking. If the table is a summary table, the data is refreshed as necessary and, when defined with the REFRESH IMMEDIATE attribute, immediate refreshing of the data is turned on.
- Turn on integrity checking for one or more tables without first carrying out any deferred integrity checking. If the table is a summary table defined with the REFRESH IMMEDIATE attribute, immediate refreshing of the data is turned on.
- Place the table into Check pending state if the table is already in DataLink Reconcile Pending (DRP) or DataLink Reconcile Not Possible (DRNP) state. If a table is not in either of those states, then unconditionally set the table in DRP state and Check-pending state.

When the statement is used to check integrity for a table after it has been loaded, the system will by default incrementally process the table by checking only the append portion for constraint violations. However, there are some situations in which the system will decide that full processing (by checking the entire table for constraints violations) is necessary to ensure data integrity. There is also a situation in which user needs to explicitly request incremental processing by specifying the INCREMENTAL option. See Notes for details.

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. However, if the bind option DYNAMICRULES BIND applies, the statement cannot be dynamically prepared (SQLSTATE 42509).

### Authorization

The privileges required to execute SET INTEGRITY depend on the use of the statement, as outlined below:

1. Turn off integrity checking.

## SET INTEGRITY

The privileges of the authorization ID of the statement must include at least one of the following:
- CONTROL privilege on the tables and all their dependents and descendants in referential integrity constraints.
- SYSADM or DBADM authority.

2. Both turn on integrity checking and carry out checking.

   The privileges of the authorization ID of the statement must include at least one of the following:
   - SYSADM or DBADM authority.
   - CONTROL privilege on the tables that are being checked **and** if exceptions are being posted to one or more tables, INSERT privilege on the exception tables.

3. Turn on integrity checking without first carrying out checking.

   The authorization ID of the statement must have at least one of the following:
   - SYSADM or DBADM authority
   - CONTROL privilege on the tables that are being checked.

### Syntax
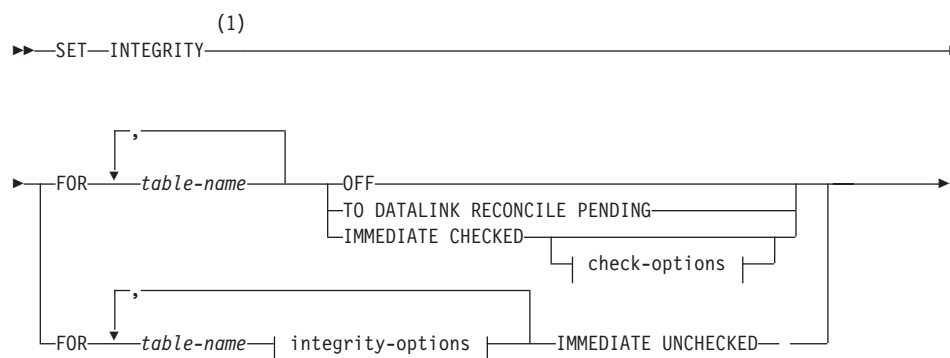


**check-options:**
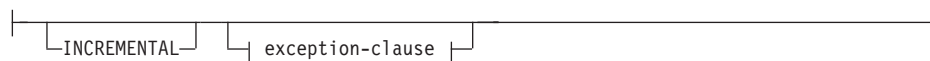


**exception-clause:**

**integrity-options:**

```
├──────ALL─────────────────────────────────────────────────────────────────────┤
       │   ┌─,──────────────────────────────────────┐
       │   │                                         │
       └───┴──┬──FOREIGN KEY───────────────────┬─────┘
       ▼      ├──CHECK─────────────────────────┤
              ├──DATALINK RECONCILE PENDING────┤
              └──SUMMARY───────────────────────┘
```
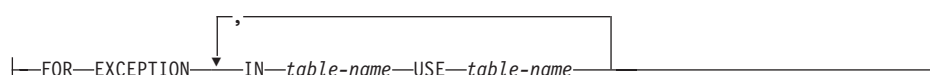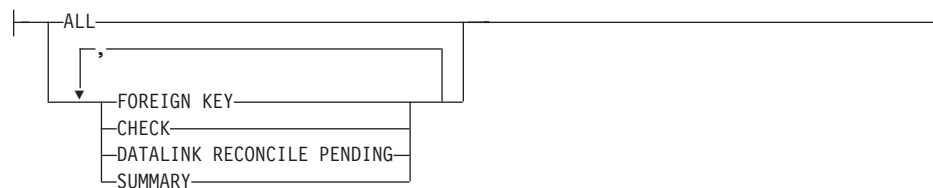
**Notes:**

**1**    For compatibility with previous versions, the keyword CONSTRAINTS will continue to be supported.

## Description

*table-name*

    Identifies a table for integrity processing. It must be a table described in the catalog and must not be a view, catalog table, or typed table. Only one summary table may be specified in the statement (SQLSTATE 42997).

**OFF**

    Specifies that the tables are to have their foreign key constraints and check constraints turned off and are, therefore to be placed into the check pending state. If it is a summary table, then immediate refreshing is turned off (if applicable) and the summary table is placed into check pending state.

    Note that it is possible that a table may already be in the check pending state with only one type of integrity checking turned off; in such a situation the other type of integrity checking will also be turned off.

    If any table in the list is a parent table, the check pending state for foreign key constraints is extended to all dependent and descendent tables.

    If any table in the list is an underlying table of a summary table, the check pending state is extended to such summary tables.

    Only very limited activity is allowed on a table that is in the check pending state. "Notes" on page 886 lists the restrictions.

**TO DATALINK RECONCILE PENDING**

    Specifies that the tables are to have DATALINK integrity constraint checking turned off and the tables placed in check pending state. If the table is already in DataLink Reconcile Not Possible (DRNP) state, it remains in this state with check pending. Otherwise, the table is set to DataLink Reconcile Pending (DRP) state.

    Dependent and descendent table are not affected when this option is specified.

**IMMEDIATE CHECKED**
Specifies that the table is to have its integrity checking turned on and that the integrity checking that was deferred is to be carried out. This is done in accordance with the information set in the STATUS and CONST_CHECKED columns of the SYSCAT.TABLES catalog. That is:

- The value in STATUS must be C (the table is in the check pending state) or an error (SQLSTATE 51027) is returned.
- The value in CONST_CHECKED indicates which integrity options are to be checked.

If it is a summary table, then the data is checked against the query and refreshed as necessary.

DATALINK values are not checked, even when the table is in DRP or DRNP state. The RECONCILE command or API should be used to perform the reconciliation of DATALINK values. The table will be taken out of check pending state but continue to have the DRP or DRNP flag set. This makes the table usable while the reconciliation of DATALINK values can be deferred to another time.

*check-options*

**INCREMENTAL**
Specifies the application of deferred integrity checks on the appended portion (if any) of the table. If such a request cannot be satisfied (i.e. the system detects that the whole table needs to be checked for data integrity), an error (SQLSTATE 55019) will be returned. If the attribute is not specified, the system will determine if incremental processing is possible; if not, the whole table will be checked. See Notes for situations in which system will favour full processing (checking whole table for integrity) over incremental processing.

If the table is not in the check pending state, an error (SQLSTATE 55019) is returned.

Note that incremental processing is the default behaviour even if the INCREMENTAL option is not specified. See Notes for situations in which the INCREMENTAL option is needed.

*exception-clause*

**FOR EXCEPTION**
Indicates that any row that is in violation of a foreign key constraint or a check constraint will be copied to an exception table and deleted from the original table. See "Appendix N. Exception Tables" on page 1175 for more information on these user-defined tables. Even if errors are detected the constraints are turned back on again and the

table is taken out of the check pending state. A warning (SQLSTATE 01603) is issued to indicate that one or more rows have been moved to the exception tables.

If the FOR EXCEPTION clause is not specified and any constraints are violated, then only the first violation detected is returned to the user (SQLSTATE 23514). In the case of a violation in any table, all the tables are left in the check pending state, as they were before the execution of the statement. This clause cannot be specified if the *table-name* is a summary table (SQLSTATE 42997).

**IN** *table-name*
Specifies the table from which rows that violate constraints are to be copied. There must be one exception table specified for each table being checked.

**USE** *table-name*
Specifies the exception table into which error rows are to be copied.

*integrity-options*
Used to define the integrity options that are set to IMMEDIATE UNCHECKED.

**ALL**
This indicates that all integrity-options are to be turned on.

**FOREIGN KEY**
This indicates that foreign key constraints are to be turned on.

**CHECK**
This indicates that check constraints are to be turned on.

**DATALINK RECONCILE PENDING**
This indicates that DATALINK integrity constraints are to be turned on.

**SUMMARY**
This indicates that immediate refreshing should be turned on for a summary table with the REFRESH IMMEDIATE attribute.

**IMMEDIATE UNCHECKED**
Specifies one of the following:
- The table is to have its integrity checking turned on (and, thus, are to be taken out of the check pending state) without having the table checked for integrity violations or the summary table is to have immediate refreshing turned on and be taken out of check pending state.

  This is specified for a given table either by specifying ALL, or by specifying CHECK when only check constraints are off for that table, or by specifying FOREIGN KEY when only foreign key constraints are off

for that table, or by specifying DATALINK RECONCILE PENDING when only DATALINK integrity constraints are off for that table or by specifying SUMMARY when only summary table query checking is off for that summary table.

- The table is to have one type of integrity turned on but is to be left in the check pending state.

  This is specified for a given table by specifying only CHECK, FOREIGN KEY, SUMMARY or DATALINK RECONCILE PENDING when any of those types of constraints are off for that table.

The state change is not extended to any tables not explicitly included in the list.

If the parent of a dependent table is in the check pending state, the foreign key constraints of a dependent table cannot be marked to bypass checking (the check constraints checking can be bypassed).

The implications with respect to data integrity should be considered before using this option. See "Notes".

### Notes

- Effects on tables in the check pending state:
  - Use of SELECT, INSERT, UPDATE, or DELETE is disallowed on a table that is either:
    - in the check pending state itself
    - or requires access to another table that is in the check pending state.

    For example, a DELETE of a row in a parent table that cascades to a dependent table that is in the check pending state is not allowed.
  - New constraints added to a table are normally enforced immediately. However, if the table is in check pending state the checking of any new constraints is deferred until the table is taken out of the check pending state.
  - The CREATE INDEX statement cannot reference any tables that are in the check pending state. Similarly, ALTER TABLE to add a primary key or unique constraint cannot reference any tables that are in the check pending state.
  - The utilities EXPORT, IMPORT, REORG, and REORGCHK are not allowed to operate on a table in the check pending state. Note that the IMPORT utility differs from the LOAD utility in that it always checks the constraints immediately.

- The utilities LOAD, BACKUP, RESTORE, ROLLFORWARD, UPDATE STATISTICS, RUNSTATS, LIST HISTORY, and ROLLFORWARD are allowed on a table in the check pending state.
- The statements ALTER TABLE, COMMENT ON, DROP TABLE, CREATE ALIAS, CREATE TRIGGER, CREATE VIEW, GRANT, REVOKE, and SET INTEGRITY can reference a table that is in the check pending state.
- Packages, views and any other objects that depend on a table that is in the check pending state will return an error when the table is accessed at run time.

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.

- Because incremental processing is the default behaviour, the INCREMENTAL option is not needed in most cases. It is needed, however, in two cases:
  - To force incremental processing on a table that was previously taken out of the check pending state with the IMMEDIATE UNCHECKED option. By default, the system chooses full processing to verify integrity of ALL data. This default behaviour can be overridden by specifying the INCREMENTAL option to check only the newly appended portion. (Refer to the bullet "Warning about the use of IMMEDIATE UNCHECKED clause" for further details.)
  - To ensure that integrity checks are indeed processed incrementally. By specifying the INCREMENTAL option, the system returns an error (SQLSTATE 55019) when the system detects that full processing is needed to ensure data integrity.
- 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.

  The fact that integrity was turned on without doing deferred checking will be recorded in the catalog (the value 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 until either:
  - The table is put back into the check pending state (by referencing the table in a SET INTEGRITY statement with the OFF clause), at which time the 'U' values in the CONST_CHECKED column will be changed to the '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.
  - A REFRESH TABLE statement is issued for a summary table.

The 'W' state differs from the 'N' state in that it records the fact that the integrity was previously checked by the user and not yet by the system, and if given a choice, the systems rechecks the whole table for data integrity and then changes it to the 'Y' state. If no choice is given (e.g. when IMMEDIATE UNCHECKED or INCREMENTAL is specified) it is changed back to the 'U' state to record that some data is still not verified by the system. In the latter (INCREMENTAL) case, a warning (SQLSTATE 01636) is returned.

- After appending data using Load Insert, the SET INTEGRITY ... IMMEDIATE CHECKED statement checks the table for constraint violation and then brings the table out of the check pending state. The system determines if incrementally processing on the table is possible. If so, only the appended portion is checked for integrity violations. If not, the system will check the whole table for integrity violations (see below for situations when the system favours full processing).

- Situations where the system checks the whole table for integrity when the user did not specify the INCREMENTAL option for the statement SET INTEGRITY for T IMMEDIATE CHECKED are:

  1. when the table T has one or more 'W' values in its CONST_CHECKED column in the SYSCAT.TABLES catalog.

- Situations where the system must check the whole table for integrity (INCREMENTAL option cannot be specified) for the statement SET INTEGRITY for T IMMEDIATE CHECKED are:

  1. when new constraints have been added to T itself or any of its parents which is in check pending

  2. when a Load Replace has taken place into T after the last integrity check on T

  3. (cascading effect of full processing) when any parent of T has been checked for integrity non-incrementally.

- A table that is in DataLink Reconcile Not Possible (DRNP) state requires corrective action to be taken (possibly outside of the database). Once corrective action is completed, the table is taken out of DRNP state using the IMMEDIATE UNCHECKED option. The RECONCILE command or API should then be used to check the DATALINK integrity constraints. For more details refer on removing a table from DataLink Reconcile Not Possible state refer to *Administration Guide, Design and Implementation*.

- While integrity is being checked an exclusive lock is held on each table specified in the SET INTEGRITY invocation.

- A shared lock is acquired on each table that is not listed in the SET INTEGRITY invocation but is a parent table of one of the dependent tables being checked.

- If an error occurs during integrity checking, all the effects of the checking including deleting from the original and inserting into the exception tables will be rolled back.

## Example

*Example 1:* The following is an example of a query that gives us information about the check pending state of tables. SUBSTR is used to extract the first 2 bytes of the CONST_CHECKED column of SYSCAT.TABLES. The first byte represents foreign key constraints, and the second byte represents check constraints.

```
SELECT TABNAME,
  SUBSTR( CONST_CHECKED, 1, 1 ) AS FK_CHECKED,
  SUBSTR( CONST_CHECKED, 2, 1 ) AS CC_CHECKED
  FROM SYSCAT.TABLES
  WHERE STATUS = 'C';
```

*Example 2:* Set tables T1 and T2 in the check pending state:

```
SET INTEGRITY FOR T1, T2 OFF;
```

*Example 3:* Check the integrity for T1 and get the first violation only:

```
SET INTEGRITY FOR T1 IMMEDIATE CHECKED
```

*Example 4:* Check the integrity for T1 and T2 and put the violating rows into exception tables E1 and E2:

```
SET INTEGRITY FOR T1, T2 IMMEDIATE CHECKED
   FOR EXCEPTION IN T1 USE E1,
                 IN T2 USE E2;
```

*Example 5:* Enable FOREIGN KEY constraint checking in T1 and CHECK constraint checking in T2 to be bypassed with the IMMEDIATE CHECKED option:

```
SET INTEGRITY FOR T1 FOREIGN KEY,
                 T2 CHECK IMMEDIATE UNCHECKED;
```

*Example 6:* Add a check constraint and a foreign key to the EMP_ACT table, using two ALTER TABLE statements. To perform constraint checking in a single pass of the table, integrity checking is turned off before the ALTER statements and checked after execution.

```
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;
```

## 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

```
►►──SET PASSTHRU──┬──server-name──┬─────────────────────────────►◄
                  └──RESET────────┘
```

### 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

Refer to "Pass-Through Facility Processing" on page 1056 for guidelines and restrictions on using pass-through.

### 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. 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

No authorization is required to execute this statement.

### Syntax

```
                   ┌─FUNCTION─┐
>>──SET──┬─CURRENT─┴──────────┴─┬──PATH──┬───┬────────────────────>
                                         └─=─┘

      ┌────────────,─────────────┐
>──────┬─schema-name──────────────┬─────────────────────────────><
       ├─SYSTEM PATH──────────────┤
       ├─USER─────────────────────┤
       │           ┌─FUNCTION─┐   │
       ├─CURRENT───┴──────────┴─PATH─┤
       ├─host-variable────────────┤
       └─string-constant──────────┘
```
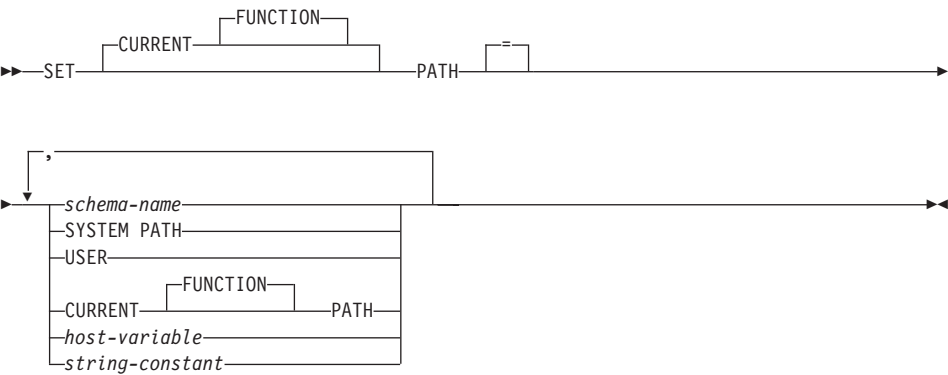
### Description

*schema-name*
    This one-part name identifies a schema that exists at the application server. It must be a short SQL identifier (SQLSTATE 42815). No validation that the schema exists is made at the time that the path is set. If a *schema-name* is, for example, misspelled, it 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"*.

**USER**
    The value in the USER special register.

**CURRENT PATH**
    The value of the CURRENT PATH before the execution of this statement. CURRENT FUNCTION PATH may also be specified.

*host-variable*
> A variable of type CHAR or VARCHAR. The length of the contents of the *host-variable* must not exceed 8 (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 justified. 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 8.

## Rules

- A schema name cannot appear more than once in the function path (SQLSTATE 42732).
- 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 quotes, doubling quotes within the schema name as necessary, and then separating each schema name by a comma. The length of the resulting string cannot exceed 254 bytes (SQLSTATE 42907).

## Notes

- The initial value of the CURRENT PATH special register is "SYSIBM","SYSFUN","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 user-defined data types, procedures and functions in dynamic SQL statements. The FUNCPATH bind option specifies the SQL path to be used for resolving user-defined data types and functions in static SQL statements. See the *Command Reference* for further information on the use of FUNCPATH option in BIND command.

## Example

*Example 1:*  The following statement sets the CURRENT FUNCTION 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;
```

## SET PATH

The value would be ″FERMAT″,″McDrw #8″,″SYSIBM″ if set by the previous example.

## SET SCHEMA

The SET SCHEMA statement changes the value of the CURRENT SCHEMA special register. It is not under transaction control. If the package is bound with DYNAMICRULES BIND option, this statement has no effect.

### Invocation

The statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

### Authorization

No authorization is required to execute this statement.

### Syntax

```
>>─SET──┬─CURRENT─┬──SCHEMA──┬───┬──┬─ schema-name ──────┬──><
        └─────────┘          └─=─┘  ├─ USER ─────────────┤
                                    ├─ host-variable ────┤
                                    └─ string-constant ──┘
```
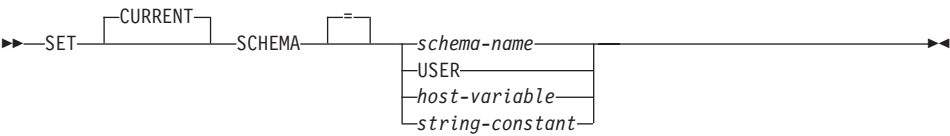
### Description

*schema-name*
> This one-part name identifies a schema that exists at the application server. It must be a short SQL identifier (SQLSTATE 42815). No validation that the schema exists is made at the time that the schema is set. If a *schema-name* is misspelled, it will not be caught, and it could affect the way subsequent SQL operates.

**USER**
> The value in the USER special register.

*host-variable*
> A variable of type CHAR or VARCHAR. The length of the contents of the *host-variable* must not exceed 8 (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 justified. 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 8.

## SET SCHEMA

### 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 (see the *Command Reference* for further information on use of the QUALIFIER option).

### Notes

- The initial value of the CURRENT SCHEMA special register is equivalent to USER.
- Setting the CURRENT SCHEMA special register does not effect 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

The authorization ID of the statement must have either SYSADM or DBADM authority on the federated database.

### Syntax

```
►►──SET SERVER OPTION──server-option-name──TO──string-constant──────────────────►

►──FOR──SERVER──server-name───────────────────────────────────────►◄
```

### Description

*server-option-name*
> Names the server option that is to be set. Refer to"Server Options" on page 1049 for descriptions of the server options.

**TO** *string-constant*
> Specifies the setting for *server-option-name* as a character string constant. Refer to "Server Options" on page 1049 for descriptions of possible settings.

**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.

**SET SERVER OPTION**

## Examples

*Example 1:* An Oracle data source called RATCHIT is defined to a federated database called DJDB. RATCHIT 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 ratchit");
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' (yes, validate 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 transition-variable

The SET transition-variable statement assigns values to new transition variables. It is under transaction control.

### Invocation

This statement can only be used as a triggered SQL statement in the triggered action of a BEFORE trigger whose granularity is FOR EACH ROW (see "CREATE TRIGGER" on page 690).

### Authorization

The privileges held by the authorization ID of the creator of the trigger must include at least one of the following:
- UPDATE of the columns referenced on the left hand side of the assignment and SELECT for any columns referenced on the right hand side.
- CONTROL privilege on the table (subject table of the trigger)
- SYSADM or DBADM authority.

To execute this statement with a *row-fullselect* as the right hand side of the assignment, the privileges held by the authorization ID of the creator of the trigger must also include at least one of the following for each table or view referenced:
- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM.

### Syntax

►►──SET────────────────────────────────────────────────────────────►

```
         ┌──────────────────────,──────────────────────────────────┐
         ▼                                                          │
    ──┬────┤ transition-variable ├──┬──=──┬──expression──┬──────────┬─►◄
      │                             └─────┬─NULL─────────┤
      │                                   └─DEFAULT──────┘
      │        ┌────,────┐                            ┌────,────┐        (1)
      └──(───▼─┤ transition-variable ├──┬──)──=──(──▼──┬──expression──┬──┬──)
               └────────────────────────┘             ├─NULL─────────┤
                                                       └─DEFAULT──────┘
                                                                     (2)
                                              └──row-fullselect──────┘
```
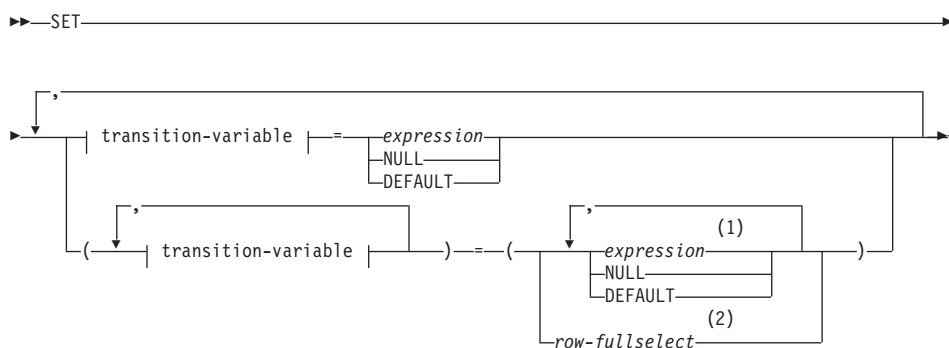
**transition-variable:**

## SET transition-variable
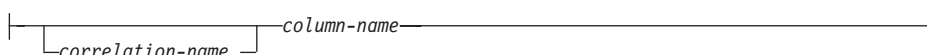
```
├──┬──────────────────┬──column-name────────────────────────────┤
   └─correlation-name.─┘
```

**Notes:**

**1** The number of expressions, NULLs and DEFAULTs must match the number of *-transition-variable*s.

**2** The number of columns in the select list must match the number of *-transition-variable*s.

## Description

**transition-variable**

Identifies a column in the set of affected rows for the trigger.

*correlation-name*

The *correlation-name* given for referencing the NEW transition variables. This *correlation-name* must match the correlation name specified following NEW in the REFERENCING clause of the CREATE TRIGGER.

If OLD is not specified in the REFERENCING clause, the *correlation-name* will default to the *correlation-name* specified following NEW. If both NEW and OLD are specified in the REFERENCING clause, then a *correlation-name* is required with each *column-name* (SQLSTATE 42702).

*column-name*

Identifies the column to be updated. The *column-name* must identify a column of the subject table of the trigger (SQLSTATE 42703). A column must not be specified more than once (SQLSTATE 42701).

*expression*

Indicates the new value of the column. The expression is any expression of the type described in "Expressions" on page 146. The expression can not include a column function except when it occurs within a scalar fullselect (SQLSTATE 42903). An *expression* may contain references to OLD and NEW transition variables and must be qualified by the *correlation-name* to specify which transition variable (SQLSTATE 42702).

**NULL**

Specifies the null value and can only be specified for nullable columns (SQLSTATE 23502).

**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 using the WITH DEFAULT clause, then the value is set to the default defined for the column (see default-clause in "ALTER TABLE" on page 449).
- If the column was defined without specifying the WITH DEFAULT clause or the NOT NULL clause, then the value inserted is NULL.
- If the column was defined using the NOT NULL clause and the WITH DEFAULT clause was not used or DEFAULT NULL was used, the DEFAULT keyword cannot be specified for that column (SQLSTATE 23502).

*row-fullselect*
  A fullselect that returns a single row with the number of columns corresponding to the number of column-names specified for assignment. The values are assigned to each corresponding column-name. If the result of the row-fullselect is no rows, then null values are assigned. A *row-fullselect* may contain references to OLD and NEW transition variables which must be qualified by the *correlation-name* to specify which transition variable to use. (SQLSTATE 42702). An error is returned if there is more than one row in the result (SQLSTATE 21000).

## Rules

- The number of values to be assigned from expressions, NULLs and DEFAULTs or the *row-fullselect* must match the number of columns specified for assignment (SQLSTATE 42802).
- If the statement is used in a BEFORE UPDATE trigger, the *column-name* specified as a *transition-variable* cannot be a partitioning key column (SQLSTATE 42997).

## Notes

- If more than one assignment is included, all the *expression*s and *row-fullselect*s are evaluated before the assignments are performed. Thus references to columns in an expression or row fullselect are always the value of the transition variable prior to any assignment in the single SET transition-variable statement.

## 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);
```

## SET transition-variable

*Example 3:* Set the salary and the commission column of the row for which the trigger action is currently executing to the average of the salary and of the commission of the employees of the updated row's department respectively.

```
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 respectively (i.e., before the SET statement was executed).

```
SET NEW_VAR.SALARY = 10000, NEW_VAR.COMM = NEW_VAR.SALARY;
 or
 SET (NEW_VAR.SALARY, NEW_VAR.COMM) = (10000, NEW_VAR.SALARY);
```

## SIGNAL SQLSTATE

The SIGNAL SQLSTATE statement is used to signal an error. It causes an error to be returned with the specified SQLSTATE and the specified *diagnostic-string*.

### Invocation

The SIGNAL SQLSTATE statement can only be used as a triggered SQL statement within a trigger.

### Authorization

No authorization is required to execute this statement.

### Syntax

```
►►──SIGNAL──SQLSTATE──string-constant──(──diagnostic-string──)──────────────►◄
```

### Description

*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 application-defined SQLSTATEs as follows:
>
> - 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' since 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 with a type of CHAR or VARCHAR that returns a character string of up to 70 bytes that describes the error condition. If the string is longer than 70 bytes, it will be truncated.

## SIGNAL SQLSTATE

### Example

Consider an order system that records orders in an ORDERS table
(ORDERNO, CUSTNO, PARTNO, QUANTITY) only if there is sufficient stock
in the PARTS tables.

```
CREATE TRIGGER check_avail
       NO CASCADE BEFORE INSERT ON orders
       REFERENCING NEW AS new_order
       FOR EACH ROW MODE DB2SQL
       WHEN (new_order.quantity > (SELECT on_hand FROM parts
                                   WHERE new_order.partno=parts.partno))
         BEGIN ATOMIC
           SIGNAL SQLSTATE '75001' ('Insufficient stock for order');
         END
```

## UPDATE

The UPDATE statement updates the values of specified columns in rows of a table or view. Updating a row of a view updates a row of its base table.

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:
- UPDATE privilege on the table or view where rows are to be updated
- UPDATE privilege on each of the columns to be updated.
- CONTROL privilege on the table or view where rows are to be updated
- SYSADM or DBADM authority.
- If a *row-fullselect* is included in the assignment, at least one of the following for each referenced table or view:
  - SELECT privilege
  - CONTROL privilege
  - SYSADM or DBADM authority.

For each table or view referenced by a subquery, the privileges held by the authorization ID of the statement must also include at least one of the following:
- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM authority.

When the package is precompiled with SQL92 rules[94] and the searched form of an UPDATE includes a reference to a column of the table or view 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:

---

94. The package used to process the statement is precompiled using option LANGLEVEL with value SQL92E or MIA.

## UPDATE

- SELECT privilege
- CONTROL privilege
- SYSADM or DBADM authority.

When 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 UPDATE statements.

### Syntax

**Searched UPDATE:**
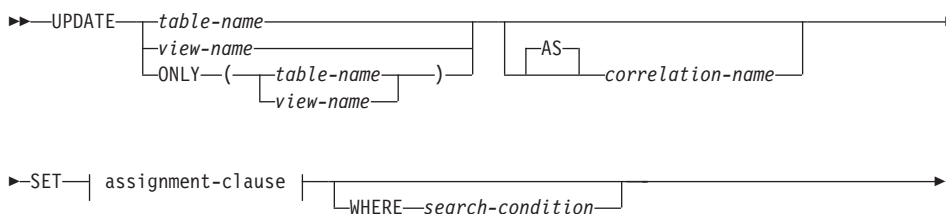


**Positioned UPDATE:**
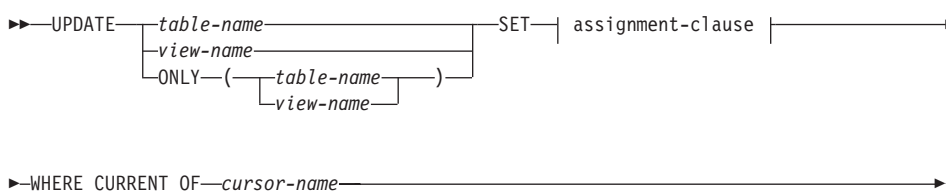


**assignment-clause:**
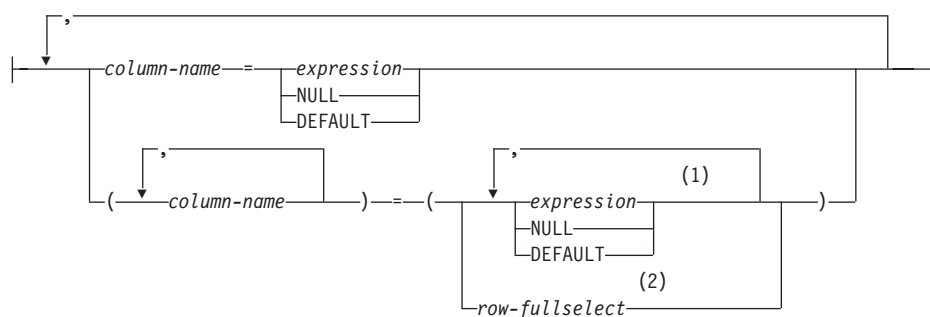
**Notes:**

1   The number of expressions, NULLs and DEFAULTs must match the number of *-column-name*s.

2   The number of columns in the select list must match the number of *-column-name*s.

## Description

*table-name* or *view-name*
>   Is the name of the table or view to be updated. The name must identify a table or view described in the catalog, but not a catalog table, a view of a catalog table (unless it is one of the updatable SYSSTAT views), a summary table, read-only view, or a nickname. (For an explanation of read-only views, see "CREATE VIEW" on page 714. For an explanation of updatable catalog views, see "Appendix D. Catalog Views" on page 945.)

>   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 or view in the FROM clause without using ONLY.

**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.

**AS**
>   Optional keyword to introduce the *correlation-name*.

*correlation-name*
>   May be used within search-condition to designate the table or view. (For an explanation of correlation-name, see "Correlation Names" on page 124.)

**SET**
>   Introduces the assignment of values to column names.

*assignment-clause*

# UPDATE

    *column-name*

Identifies a column to be updated. The *column-name* must identify an updatable column of the specified table or view.[95] The object ID column of a typed table is not updatable (SQLSTATE 428DZ). A column must not be specified more than once (SQLSTATE 42701).

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.

    *expression*

Indicates the new value of the column. The expression is any expression of the type described in "Expressions" on page 146. The expression can not include a column 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.

**NULL**

Specifies the null value and can only be specified for nullable columns (SQLSTATE 23502).

**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 using the WITH DEFAULT clause, then the value is set to the default defined for the column (see default-clause in "ALTER TABLE" on page 449).

- If the column was defined without specifying the WITH DEFAULT clause or the NOT NULL clause, then the value inserted is NULL.

---

95. A column of a partitioning key is not updatable (SQLSTATE 42997). The row of data must be deleted and inserted to change columns in a partitioning key.

- If the column was defined using the NOT NULL clause and the WITH DEFAULT clause was not used or DEFAULT NULL was used, the DEFAULT keyword cannot be specified for that column (SQLSTATE 23502).

*row-fullselect*

A fullselect that returns a single row with the number of columns corresponding to the number of *column-name*s specified for assignment. The values are assigned to each corresponding *column-name*. If the result of the *row-fullselect* is no rows, then null values are assigned.

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).

**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 or view are updated.

*search-condition*

Is any search condition as described in "Chapter 3. Language Elements" on page 61. Each *column-name* in the search condition, other than in a subquery, must name a column of the table or view. 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 or view 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 as explained in "DECLARE CURSOR" on page 731. The DECLARE CURSOR statement must precede the UPDATE statement in the program.

The table or view 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" on page 731.)

When the UPDATE statement is executed, the cursor must be positioned on a row; that row is updated.

### Rules

- *Assignment*: Update values are assigned to columns under the assignment rules described in Chapter 3..
- *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" on page 714.
- *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.

### 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.
- When an UPDATE statement completes execution, the value of SQLERRD(3) in the SQLCA is the number of rows updated. The SQLERRD(5) field contains the number of rows inserted, deleted, or updated by all activated triggers. (For a description of the SQLCA, see "Appendix B. SQL Communications (SQLCA)" on page 927.)
- 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.

- If the URL value of a DATALINK column is updated, this is the same as deleting the old DATALINK value then inserting the new one. First, if the old value was linked to a file, that file is unlinked. Then, unless the linkage attributes of the DATALINK value are empty, the specified file is linked to that column.

  The comment value of a DATALINK column can be updated without relinking the file by specifying an empty string as the URL path (for example, as the *data-location* argument of the DLVALUE scalar function or by specifying the new value to be the same as the old value).

  If a DATALINK column is updated with a null, it is the same as deleting the existing DATALINK value.

  An error may occur when attempting to update a DATALINK value if the file server of either the existing value or the new value is no longer registered with the database server (SQLSTATE 55022).

- When updating the column distribution statistics for a typed table, the subtable that first introduced the column must be specified.

## 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 NULL 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'
  ```

## UPDATE

- *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 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;
```

## VALUES

The VALUES statement is a form of query. It can be embedded in an
application program or issued interactively. For detailed information, see
"fullselect" on page 407.

## 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

None required.

### Syntax

```
>>──VALUES──┬─expression───────────────┬──INTO──┬─host-variable──┬──><
            │                          │        └──────,◄────────┘
            │      ┌──,◄──────┐        │
            └──(───┴─expression─┴───)──┘
```
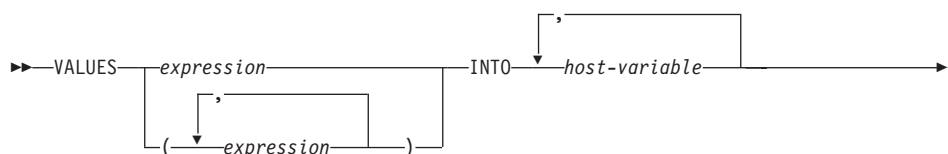
### Description

**VALUES**
Introduces a single row consisting of one of 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.

**INTO**
Introduces a list of host variables.

*host-variable*
Identifies a variable that is described in the program under the rules for declaring host variables.

The first value in the result row is assigned to the first variable in the list, the second value to the second variable, and so on. If the number of host variables is less than the number of column values, the value 'W' is assigned to the SQLWARN3 field of the SQLCA. (See "Appendix B. SQL Communications (SQLCA)" on page 927.)

Each assignment to a variable is made according to the rules described in "Assignments and Comparisons" on page 91. Assignments are made in sequence through the list.

If an error occurs, no value is assigned to any host variable.

## 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 :hv1;
```

*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;
```

## 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

```
>>-WHENEVER---NOT FOUND-----CONTINUE---------------------------------------><
             |-SQLERROR--|  |-GOTO--|                                  
             '-SQLWARNING-' '-GO TO-' '-:-'     host-label
```
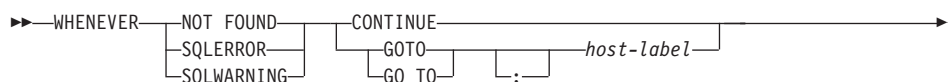
### Description

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.

**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.

**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;
```

**WHENEVER**

# Appendix A. SQL Limits

The following tables describe certain SQL limits. Adhering to the most restrictive case can help the programmer design application programs that are easily portable.

*Table 26. Identifier Length Limits*

| | Description | Limit in Bytes |
|---|---|---|
| 1 | Longest authorization name (can only be single-byte characters) | 8 |
| 2 | Longest constraint name | 18 |
| 3 | Longest correlation name | 128 |
| 4 | Longest cursor name | 18 |
| 5 | Longest external program name | 8 |
| 6 | Longest host identifier [a] | 255 |
| 7 | Longest schema name | 8 |
| 8 | Longest server (database alias) name | 8 |
| 9 | Longest statement name | 18 |
| 10 | Longest unqualified column name | 30 |
| 11 | Longest unqualified package name | 8 |
| 12 | Longest unqualified user-defined type, user-defined function, stored procedure, buffer pool, table space, nodegroup, trigger, index, or index specification name | 18 |
| 13 | Longest unqualified table name, view name, nickname, or alias | 128 |
| 14 | Longest wrapper name | 128 |
| 15 | Longest identifier of a data source user (*remote-authorization-name*) | 30 |
| 16 | Longest password to access a data source | 32 |
| 17 | Longest data source name | 18 |
| 18 | Longest data source table name (*remote-table-name*) | 128 |
| 19 | Longest data source column name | 128 |
| 20 | Longest data source index name | 128 |

## SQL Limits

*Table 26. Identifier Length Limits  (continued)*

| Description | Limit in Bytes |
|---|---|

**Notes:**

a    Individual host language compilers may have a more restrictive limit on variable names.

*Table 27. Numeric Limits*

| | Description | Limit |
|---|---|---|
| 1 | Smallest INTEGER value | −2 147 483 648 |
| 2 | Largest INTEGER value | +2 147 483 647 |
| 3 | Smallest BIGINT value | −9 223 372 036 854 775 808 |
| 4 | Largest BIGINT value | +9 223 372 036 854 775 807 |
| 5 | Smallest SMALLINT value | −32 768 |
| 6 | Largest SMALLINT value | +32 767 |
| 7 | Largest decimal precision | 31 |
| 8 | Smallest DOUBLE value | −1.79769E+308 |
| 9 | Largest DOUBLE value | +1.79769E+308 |
| 10 | Smallest positive DOUBLE value | +2.225E−307 |
| 11 | Largest negative DOUBLE value | −2.225E−307 |
| 12 | Smallest REAL value | −3.402E+38 |
| 13 | Largest REAL value | +3.402E+38 |
| 14 | Smallest positive REAL value | +1.175E−37 |
| 15 | Largest negative REAL value | −1.175E−37 |

*Table 28. String Limits*

| | Description | Limit |
|---|---|---|
| 1 | Maximum length of CHAR (in bytes) | 254 |
| 2 | Maximum length of VARCHAR (in bytes) | 32 672 |
| 3 | Maximum length of LONG VARCHAR (in bytes) | 32 700 |
| 4 | Maximum length of CLOB (in bytes) | 2 147 483 647 |
| 5 | Maximum length of GRAPHIC (in characters) | 127 |
| 6 | Maximum length of VARGRAPHIC (in characters) | 16 336 |
| 7 | Maximum length of LONG VARGRAPHIC (in characters) | 16 350 |
| 8 | Maximum length of DBCLOB (in characters) | 1 073 741 823 |

*Table 28. String Limits (continued)*

|  | Description | Limit |
|---|---|---|
| 9 | Maximum length of BLOB (in bytes) | 2 147 483 647 |
| 10 | Maximum length of character constant | 32 672 |
| 11 | Maximum length of graphic constant | 16 336 |
| 12 | Maximum length of concatenated character string | 2 147 483 647 |
| 13 | Maximum length of concatenated graphic string | 1 073 741 823 |
| 14 | Maximum length of concatenated binary string | 2 147 483 647 |
| 15 | Maximum number of hex constant digits | 16 336 |
| 16 | Maximum size of a catalog comment (in bytes) | 254 |

*Table 29. Datetime Limits*

|  | Description | Limit |
|---|---|---|
| 1 | Smallest DATE value | 0001-01-01 |
| 2 | Largest DATE value | 9999-12-31 |
| 3 | Smallest TIME value | 00:00:00 |
| 4 | Largest TIME value | 24:00:00 |
| 5 | Smallest TIMESTAMP value | 0001-01-01-00.00.00.000000 |
| 6 | Largest TIMESTAMP value | 9999-12-31-24.00.00.000000 |

*Table 30. Database Manager Limits*

|  | Description | Limit |
|---|---|---|
| 1 | Most columns in a table [g] | 1 012 |
| 2 | Most columns in a view [a] | 5 000 |
| 3 | Maximum length of a row including all overhead [b] [g] | 32 677 |
| 4 | Maximum size of a table per partition (in gigabytes) [c] [g] | 512 |
| 5 | Maximum size of an index per partition (in gigabytes) | 512 |
| 6 | Most rows in a table per partition | $4 \times 10^9$ |
| 7 | Longest index key including all overhead (in bytes) | 1 024 |
| 8 | Most columns in an index key | 16 |
| 9 | Most indexes on a table | 32 767 or storage |

# SQL Limits

*Table 30. Database Manager Limits  (continued)*

|  | Description | Limit |
|---|---|---|
| 10 | Most tables referenced in an SQL statement or a view | storage |
| 11 | Most host variable declarations in a precompiled program [c] | storage |
| 12 | Most host variable references in an SQL statement | 32 767 |
| 13 | Longest host variable value used for insert or update (in bytes) | 2 147 483 647 |
| 14 | Longest SQL statement (in bytes) | 65 535 |
| 15 | Most elements in a select list[g] | 1 012 |
| 16 | Most predicates in a WHERE or HAVING clause | storage |
| 17 | Maximum number of columns in a GROUP BY clause [g] | 1 012 |
| 18 | Maximum total length of columns in a GROUP BY clause (in bytes)[g] | 32 677 |
| 19 | Maximum number of columns in an ORDER BY clause [g] | 1 012 |
| 20 | Maximum total length of columns in an ORDER BY clause (in bytes) [g] | 32 677 |
| 21 | Maximum size of an SQLDA (in bytes) | storage |
| 22 | Maximum number of prepared statements | storage |
| 23 | Most declared cursors in a program | storage |
| 24 | Maximum number of cursors opened at one time | storage |
| 25 | Most tables in an SMS table space | 65 534 |
| 26 | Maximum number of constraints on a table | storage |
| 27 | Maximum level of subquery nesting | storage |
| 28 | Maximum number of subqueries in a single statement | storage |
| 29 | Most values in an INSERT statement [g] | 1 012 |
| 30 | Most SET clauses in a single UPDATE statement [g] | 1 012 |
| 31 | Most columns in a UNIQUE constraint (supported via a UNIQUE index) | 16 |

*Table 30. Database Manager Limits  (continued)*

|  | Description | Limit |
|---|---|---|
| 32 | Maximum combined length of columns in a UNIQUE constraint (supported via a UNIQUE index) (in bytes) | 1 024 |
| 33 | Most referencing columns in a foreign key | 16 |
| 34 | Maximum combined length of referencing columns in a foreign key (in bytes) | 1 024 |
| 35 | Maximum length of a check constraint specification (in bytes) | 65 535 |
| 36 | Maximum number of columns in a partitioning key [e] | 500 |
| 37 | Maximum number of rows changed in a unit of work | storage |
| 38 | Maximum number of packages | storage |
| 39 | Most constants in a statement | storage |
| 40 | Maximum concurrent users of server [d] | 64 000 |
| 41 | Maximum number of parameters in a stored procedure | 32 767 |
| 42 | Maximum number of parameters in a user defined function | 90 |
| 43 | Maximum run-time depth of cascading triggers | 16 |
| 44 | Maximum number of simultaneously active event monitors | 32 |
| 45 | Maximum size of a regular DMS table space (in gigabytes)[c] [g] | 512 |
| 46 | Maximum size of a long DMS table space (in terabytes)[c] | 2 |
| 47 | Maximum size of a temporary DMS table space (in terabytes)[c] | 2 |
| 48 | Maximum number of databases per instance concurrently in use | 256 |
| 49 | Maximum number of concurrent users per instance | 64 000 |
| 50 | Maximum number of concurrent applications per database | 1 000 |
| 51 | Maximum depth of cascaded triggers | 16 |
| 52 | Maximum partition number | 999 |

## SQL Limits

*Table 30. Database Manager Limits (continued)*

| | Description | Limit |
|---|---|---|
| 53 | Most table objects in DMS table space [f] | 51 000 |
| 54 | Longest variable index key part (in bytes) | 255 |
| 55 | Maximum number of columns in a data source table or view that is referenced by a nickname | 5 000 |

**Notes:**

**a**    This maximum can be achieved 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.

**b**    The actual data for BLOB, CLOB, LONG VARCHAR, DBCLOB, and LONG VARGRAPHIC columns is not included in this count. However information about the location of that data does take up some space in the row.

**c**    The numbers shown are architectural limits and approximations. The practical limits may be less.

**d**    The actual value will be the value of the MAXAGENTS configuration parameter. See the *Administration Guide* for information on MAXAGENTS.

**e**    This is an architectural limit. The limit on the most columns in an index key should be used as a practical limit.

**f**    Table objects include data, indexes, LONG VARCHAR/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.

**g**    For page size specific values please refer to Table 31.

*Table 31. Database Manager Page Size Specific Limits*

| | Description | 4K page size limit | 8K page size limit | 16K page size limit | 32K page size limit |
|---|---|---|---|---|---|
| 1 | Most columns in a table | 500 | 1 012 | 1 012 | 1 012 |
| 3 | Maximum length of a row including all overhead | 4 005 | 8 101 | 16 293 | 32 677 |
| 4 | Maximum size of a table per partition (in gigabytes) | 64 | 128 | 256 | 512 |
| 5 | Maximum size of an index per partition (in gigabytes) | 64 | 128 | 256 | 512 |
| 15 | Most elements in a select list | 500 | 1 012 | 1 012 | 1 012 |

*Table 31. Database Manager Page Size Specific Limits  (continued)*

|  | Description | 4K page size limit | 8K page size limit | 16K page size limit | 32K page size limit |
|---|---|---|---|---|---|
| 17 | Maximum number of columns in a GROUP BY clause | 500 | 1 012 | 1 012 | 1 012 |
| 18 | Maximum total length of columns in a GROUP BY clause (in bytes) | 4 005 | 8 101 | 16 293 | 32 677 |
| 19 | Maximum number of columns in an ORDER BY clause | 500 | 1 012 | 1 012 | 1 012 |
| 20 | Maximum total length of columns in an ORDER BY clause (in bytes) | 4 005 | 8 101 | 16 293 | 32 677 |
| 29 | Most values in an INSERT statement | 500 | 1 012 | 1 012 | 1 012 |
| 30 | Most SET clauses in a single UPDATE statement | 500 | 1 012 | 1 012 | 1 012 |
| 45 | Maximum size of a regular DMS table space (in gigabytes) | 64 | 128 | 256 | 512 |

**SQL Limits**

# Appendix B. SQL Communications (SQLCA)

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 (except for DECLARE, INCLUDE, and WHENEVER) 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.

## Viewing the SQLCA Interactively

To display the SQLCA after each command you use in the command line processor, use 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 32. Fields of SQLCA*

| Name[96] | Data Type | Field Values |
|---|---|---|
| sqlcaid | CHAR(8) | An "eye catcher" for storage dumps containing 'SQLCA'. |
| sqlcabc | INTEGER | Contains the length of the SQLCA, 136. |

---

96. The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement.

## SQLCA

*Table 32. Fields of SQLCA  (continued)*

| Name[96] | Data Type | Field Values |
|---|---|---|
| sqlcode | INTEGER | Contains the SQL return code. For specific meanings of SQL return codes, see the message section of the *Message Reference*. |
| | | **Code**    **Means**<br>**0**           Successful execution (although one or more SQLWARN indicators may be set).<br>**positive**<br>               Successful execution, but with a warning condition.<br>**negative**<br>               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', that are substituted for variables in the descriptions of error conditions.<br><br>This field is also used when a successful connection is completed.<br><br>When a NOT ATOMIC compound SQL statement is issued, it may contain information on up to 7 errors.<br><br>For specific meanings of SQL return codes, see the message section of the *Message Reference*. |
| sqlerrp | CHAR(8) | Begins with a three-letter identifier indicating the product, followed by five digits indicating the version, release, and modification level of the product. For example, SQL06010 means DB2 Universal Database versions for Version 6 Release 1 Modification level 0.<br><br>If SQLCODE indicates an error condition, then this field identifies the module that returned the error.<br><br>This field is also used when a successful connection is completed. |
| 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. |

*Table 32. Fields of SQLCA (continued)*

| Name[96] | Data Type | Field Values |
|---|---|---|
| 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. [a] |
| 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. [a] 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, and DELETE, contains the actual number of rows affected. 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; 2 if the database is read only. |
| 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 down-level client; 1 for a one-phase commit; 2 for a one-phase, read-only commit; and 3 for a two-phase commit. |

# SQLCA

*Table 32. Fields of SQLCA  (continued)*

| Name[96] | Data Type | Field Values |
|---|---|---|
| sqlerrd(5) | INTEGER | Contains the total number of rows deleted, inserted, or updated as a result of both:<br><br>• The enforcement of constraints after a successful delete operation<br>• The processing of triggered SQL statements from activated triggers.<br><br>If compound SQL is invoked, contains an accumulation of the number of such rows for all substatements. 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 of 0 for a server authentication; 1 for client authentication; 2 for authentication using DB2 Connect; 3 for DCE security services authentication; 255 for unspecified authentication. |
| sqlerrd(6) | INTEGER | For a partitioned database, contains the partition number of the partition that encountered the error or warning. If no errors or warnings were encountered, this field contains the partition number of the coordinator node. The number in this field is the same as that specified for the 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 substatements. |
| 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. |
| sqlwarn2 | CHAR(1) | Contains W if null values were eliminated from the argument of a function. [b] |
| sqlwarn3 | CHAR(1) | Contains W if the number of columns is not equal to the number of host variables. |
| sqlwarn4 | CHAR(1) | Contains W if a prepared UPDATE or DELETE statement does not include a WHERE clause. |
| sqlwarn5 | CHAR(1) | Reserved for future use. |
| sqlwarn6 | CHAR(1) | Contains W if the result of a date calculation was adjusted to avoid an impossible date. |
| sqlwarn7 | CHAR(1) | Reserved for future use. |
| sqlwarn8 | CHAR(1) | Contains W if a character that could not be converted was replaced with a substitution character. |

*Table 32. Fields of SQLCA  (continued)*

| Name[96] | Data Type | Field Values |
|---|---|---|
| sqlwarn9 | CHAR(1) | Contains W if arithmetic expressions with errors were ignored during column 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. |

**Note:**

**a**     See the "Character Conversion Expansion Factor" section of the "Programming in Complex Environments" chapter in the *Application Development Guide* for details.

**b**     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.

## Order of Error Reporting

The order of error reporting is as follows:

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.

   For DB2 Enterprise - Extended Edition, the exception to this rule occurs if a data manipulation operation is issued on a table that is empty on one partition, but has data on other nodes. The SQLCODE +100 is only returned to the application if agents from all partitions return SQL0100W, either because the table is empty on all partitions or there are no rows that satisfy the WHERE clause in an UPDATE statement.

## DB2 Enterprise - Extended Edition Usage of the SQLCA

In DB2 Universal Database Enterprise - Extended Edition, one SQL statement may be executed by a number of agents on different 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. For example:

## SQLCA

- 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 that SQLSTATE 09000 may not be returned in all cases of an error occurring while processing a triggered SQL statement.

# Appendix C. SQL Descriptor Area (SQLDA)

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, EXECUTE, and CALL 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 statement.

SQLDAs are supported for all languages, but predefined declarations are provided only for C, REXX, FORTRAN, and COBOL. In REXX, the SQLDA is somewhat different than in the other languages; for information on the use of SQLDAs in REXX see the *Application Development Guide*

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, FETCH, and CALL, an SQLDA describes host variables.

In DESCRIBE and PREPARE, if any one of the columns being described is either a LOB type, [97] reference type or a distinct 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, OPEN, and CALL, if any one of the variables being described is a LOB type, [97] the number of SQLVAR entries for the entire SQLDA needs to be doubled.[98]

## 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, EXECUTE, and CALL each occurrence of SQLVAR describes a host variable. In DESCRIBE and PREPARE, each occurrence of SQLVAR describes a column of a result table. There are two types of SQLVAR entries:

---

97. LOB locators and file reference variables do not require doubled SQLDAs.

98. Distinct types are not relevant in these cases since a host variable cannot be a distinct type.

1. **Base SQLVARs:** These entries are always present. They contain the base information about the column or host variable such as data type code, length attribute, column name, host variable address, and indicator variable address.

2. **Secondary SQLVARs:** These entries are only present if the number of SQLVAR entries is doubled as per the rules outlined above. For distinct types they contain the distinct type name. For reference types, they contain that 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.[99] 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 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 939.

### Fields in the SQLDA Header

*Table 33. 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, EXECUTE, and CALL (set by the application prior to 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 939 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 BLOB, CLOB, or DBCLOB, the seventh byte must be set to the character '2'; otherwise it can be set to any character but we advise the use of a blank.

When used with the CALL statement and one or more SQLVARs define of data field as FOR BIT DATA, the sixth byte must be set to the '+' character; otherwise it can be set to any character but we advise the use of blank. |

---

99. 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.

*Table 33. Fields in the SQLDA Header  (continued)*

| C Name | SQL Data Type | Usage in DESCRIBE and PREPARE (set by the database manager except for SQLN) | Usage in FETCH, OPEN, EXECUTE, and CALL (set by the application prior to executing the statement) |
|---|---|---|---|
| sqldabc | INTEGER | Length of the SQLDA, equal to SQLN*44+16. | Length of the SQLDA, >= to SQLN*44+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 zero if the statement being described is not a select-statement). | The number of host variables described by occurrences of SQLVAR. |

## Fields in an Occurrence of a Base SQLVAR

*Table 34. Fields in a Base SQLVAR*

| Name | Data Type | Usage in DESCRIBE and PREPARE | Usage in FETCH, OPEN, EXECUTE, and CALL |
|---|---|---|---|
| sqltype | SMALLINT | Indicates the data type of the column and whether it can contain nulls. Table 36 on page 940 lists the allowable values and their meanings.<br><br>Note that for a distinct or reference type, the data type of the base type is placed into this field. There is no indication in the Base SQLVAR that it is part of the description of a distinct 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. For datetime columns, the length of the string representation of the values. See Table 36 on page 940.<br><br>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 36 on page 940.<br><br>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. |

## SQLDA

*Table 34. Fields in a Base SQLVAR  (continued)*

| Name | Data Type | Usage in DESCRIBE and PREPARE | Usage in FETCH, OPEN, EXECUTE, and CALL |
|------|-----------|-------------------------------|------------------------------------------|
| sqldata | pointer | For character-string SQLVARs, sqldata contains 0 if the column is defined with the FOR BIT DATA attribute. If the column does not have the FOR BIT DATA attribute, the value depends on the encoding of the data. For single-byte SBCS encoded data, sqldata contains the SBCS code page. For mixed DBCS encoded data, sqldata contains the SBCS code page associated with the composite DBCS code page. For Japanese or Traditional-Chinese EUC encoded data, sqldata contains the composite EUC code page. | Contains the address of the host variable (where the fetched data will be stored). |
| | | For all other column types, sqldata is undefined. | |
| sqlind | pointer | For character-string SQLVARs, sqlind contains 0 except for mixed DBCS encoded data when sqlind contains the DBCS code page associated with the composite DBCS code page. | 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. |
| | | For all other column types, sqlind is undefined. | |

*Table 34. Fields in a Base SQLVAR (continued)*

| Name | Data Type | Usage in DESCRIBE and PREPARE | Usage in FETCH, OPEN, EXECUTE, and CALL |
|------|-----------|-------------------------------|------------------------------------------|
| sqlname | VARCHAR (30) | Contains the unqualified name of the column.<br><br>For columns that have a system generated name (the result column was not directly derived from a single column and did not specify a name using the AS clause), the thirtieth byte is set to X'FF'. For column names specified by the AS clause, this byte is X'00'. | When used with the CALL statement to access a DRDA application server, sqlname can be set to indicate a FOR BIT DATA string as follows:<br><br>• the length of sqlname is 8<br>• the first four bytes of sqlname are X'00000000'<br>• the remaining four bytes of sqlname are reserved (and currently ignored).<br><br>In addition, the sqltype must indicate a CHAR, VARCHAR or LONG VARCHAR and the sixth byte of the sqldaid field is set to the '+' character.<br><br>This technique can also be used with OPEN and EXECUTE when using DB2 Connect to access the server. |

## Fields in an Occurrence of a Secondary SQLVAR

*Table 35. Fields in a Secondary SQLVAR*

| Name | Data Type | Usage in DESCRIBE and PREPARE | Usage in FETCH, OPEN, EXECUTE, and CALL |
|------|-----------|-------------------------------|------------------------------------------|
| len.sqllonglen | INTEGER | The length attribute of a BLOB, CLOB, or DBCLOB column. | 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 characters for a DBCLOB. |
| reserve2 | CHAR(3) | 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. 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. Otherwise, the value is X'00'. |

*Table 35. Fields in a Secondary SQLVAR (continued)*

| Name | Data Type | Usage in DESCRIBE and PREPARE | Usage in FETCH, OPEN, EXECUTE, and CALL |
|---|---|---|---|
| sqldatalen | pointer | Not used. | Used for BLOB, CLOB, and DBCLOB host variables only. |
| | | | If this field is NULL, then the actual length (in 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 NULL, 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 distinct type column, the database manager sets this to the fully qualified distinct 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. | Not used. |
| reserved | CHAR(3) | Not used. | Not used. |

*Table 35. Fields in a Secondary SQLVAR (continued)*

| Name | Data Type | Usage in DESCRIBE and PREPARE | Usage in FETCH, OPEN, EXECUTE, and CALL |
|------|-----------|-------------------------------|------------------------------------------|

**Note:**

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 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 or PREPARE INTO statement, the database manager always sets SQLD to the number of columns in the result set.

The SQLVARs in the SQLDA are set in the following cases:

- SQLN >= SQLD and no column is either a LOB, distinct type or reference type

  The first SQLD SQLVAR entries are set and SQLDOUBLED is set to blank.

- SQLN >= 2*SQLD and at least one column is a LOB, distinct 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 column is a distinct type or reference type but there are no LOB columns

  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 column is either a LOB, distinct 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.

# SQLDA

- SQLN < SQLD and at least one column is a distinct type or reference type but there are no LOB columns

  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 types and target types of reference types.

- SQLN < 2*SQLD and at least one column is a LOB

  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 above lists to LOB columns include distinct type columns 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 columns 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 column in the result set.

## SQLTYPE and SQLLEN

Table 36 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 the column does not allow nulls, and an odd value means the column does allow nulls. In FETCH, OPEN, EXECUTE, and CALL, an even value of SQLTYPE means no indicator variable is provided, and an odd value means that SQLIND contains the address of an indicator variable.

*Table 36. SQLTYPE and SQLLEN values for DESCRIBE, FETCH, OPEN, EXECUTE, and CALL*

| | For DESCRIBE and PREPARE INTO | | For FETCH, OPEN, EXECUTE, and CALL | |
|---|---|---|---|---|
| **SQLTYPE** | **Column Data Type** | **SQLLEN** | **Host Variable Data Type** | **SQLLEN** |
| 384/385 | date | 10 | fixed-length character string representation of a date | length attribute of the host variable |

*Table 36. SQLTYPE and SQLLEN values for DESCRIBE, FETCH, OPEN, EXECUTE, and CALL  (continued)*

| SQLTYPE | For DESCRIBE and PREPARE INTO | | For FETCH, OPEN, EXECUTE, and CALL | |
|---|---|---|---|---|
| | Column Data Type | SQLLEN | Host Variable Data Type | SQLLEN |
| 388/389 | time | 8 | fixed-length character string representation of a time | length attribute of the host variable |
| 392/393 | timestamp | 26 | fixed-length character string representation of a timestamp | length attribute of the host variable |
| 396/397 | DATALINK | length attribute of the column | DATALINK | length attribute of the host variable |
| 400/401 | N/A | N/A | NUL-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 |
| 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 | N/A | N/A | NUL-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 |

## SQLDA

*Table 36. SQLTYPE and SQLLEN values for DESCRIBE, FETCH, OPEN, EXECUTE, and CALL  (continued)*

| | For DESCRIBE and PREPARE INTO | | For FETCH, OPEN, EXECUTE, and CALL | |
|---|---|---|---|---|
| SQLTYPE | Column Data Type | SQLLEN | Host Variable Data Type | SQLLEN |
| 496/497 | large integer | 4 | large integer | 4 |
| 500/501 | small integer | 2 | small integer | 2 |
| 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 |

**Note:**

- The len.sqllonglen field in the secondary SQLVAR contains the length attribute of the column.
- The SQLTYPE has changed from the previous version for portability in DB2. The values from the previous version (see previous version SQL Reference) will 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 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.

---

100. ROWID is supported by DB2 Universal Database for OS/390 Version 6.

| Data Type | Compatible Data Type |
|-----------|---------------------|
| BIGINT | DECIMAL(19, 0) |
| ROWID | VARCHAR(40) FOR BIT DATA [100] |

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 37. Values for Sign Indicator of a Packed Decimal Number*

| Sign | Representation | | |
|------|--------|---------|-------------|
| | Binary | Decimal | Hexadecimal |
| Positive (+) | 1100 | 12 | C |
| Negative (-) | 1101 | 13 | D |

In summary:
1. To store any value, allocate $p/2+1$ bytes, where $p$ is precision.
2. 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.
3. The sign nybble will be the second nybble of the last byte.

There is an alternative way to perform packed decimal conversions, see "CHAR" on page 237.

For example:

| Column | Value | Nybbles in Hexadecimal Grouped by Bytes |
|--------|-------|------------------------------------------|
| DEC(8,3) | 6574.23 | 00 65 74 23 0C |

| Column | Value | Nybbles in Hexadecimal Grouped by Bytes |
|--------|------:|------------------------------------------|
| 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;
```

# Appendix D. Catalog Views

The database manager creates and maintains two sets of system catalog views. This appendix contains a description of each system catalog view, including column names and data types. All the system catalog views are created when a database is created with the CREATE DATABASE command. The catalog views cannot be explicitly created or dropped. The system catalog views are updated during normal operation in response to SQL data definition statements, environment routines, and certain utilities. Data in the system catalog views is available through normal SQL query facilities. The system catalog views cannot be modified using normal SQL data manipulation commands with the exception of some specific updatable catalog views.

The catalog views are supported in addition to the catalog base tables from Version 1. The views are within the SYSCAT schema and SELECT privilege on all views is granted to PUBLIC by default. Application programs should be written to these views rather than the base catalog tables. A second set of views formed from a subset of those within the SYSCAT schema, contain statistical information used by the optimizer. The views within the SYSSTAT schema contain some updatable columns.

Warning:  The intention is to enable applications to update certain columns using the SYSSTAT views, but have the SYSCAT views read only. Currently, the SYSCAT views are not read only. Applications developers are warned to ensure that applications are written to only update catalog information using the SYSSTAT views. The SYSCAT views will become read only views after the next version migration.

The catalog views are designed to use more consistent conventions than the underlying catalog base tables. As such, the order of columns may change from release to release. To protect from this affecting programming logic, always specify explicitly the columns in a select list rather then letting them default by using SELECT *. Columns have consistent names based on the type of objects that they describe:

| Described Object | Column Names |
|---|---|
| **Table** | TABSCHEMA, TABNAME |
| **Index** | INDSCHEMA, INDNAME |
| **View** | VIEWSCHEMA, VIEWNAME |
| **Constraint** | CONSTSCHEMA, CONSTNAME |
| **Trigger** | TRIGSCHEMA, TRIGNAME |
| **Package** | PKGSCHEMA, PKGNAME |

## Catalog Views

| | |
|---|---|
| **Type** | TYPESCHEMA, TYPENAME, TYPEID |
| **Function** | FUNCSCHEMA, FUNCNAME, FUNCID |
| **Column** | COLNAME |
| **Schema** | SCHEMANAME |
| **Table Space** | TBSPACE |
| **Nodegroup** | NGNAME |
| **Buffer pool** | BPNAME |
| **Event Monitor** | EVMONNAME |
| **Creation Timestamp** | CREATE_TIME |

- "Updatable Catalog Views"

- ""Roadmap" to Catalog Views"

## Updatable Catalog Views

The updatable views contain statistical information used by the optimizer. Some columns in these views may be changed to investigate the performance of hypothetical databases. An object (table, column, function, or index) will appear in the updatable catalog view for a given user only if that user created the object, holds CONTROL privilege on the object, or holds explicit DBADM privilege. These views are found in the SYSSTAT schema. They are defined on top of the system catalog base tables.

Before changing any statistics for the first time, it is advised to issue the RUNSTATS command so that all statistics will reflect the current state.

## "Roadmap" to Catalog Views

| Description | Catalog View | Page |
|---|---|---|
| attributes of structured data types | SYSCAT.ATTRIBUTES | 950 |
| authorities on database | SYSCAT.DBAUTH | 968 |
| buffer pool configuration on nodegroup | SYSCAT.BUFFERPOOLS | 953 |
| buffer pool size on node | SYSCAT.BUFFERPOOLNODES | 952 |
| cast functions | SYSCAT.CASTFUNCTIONS | 954 |
| check constraints | SYSCAT.CHECKS | 955 |
| column privileges | SYSCAT.COLAUTH | 956 |
| columns | SYSCAT.COLUMNS | 960 |
| columns referenced by check constraints | SYSCAT.COLCHECKS | 957 |

| Description | Catalog View | Page |
|---|---|---|
| columns used in keys | SYSCAT.KEYCOLUSE | 991 |
| detailed column options | SYSCAT.COLOPTIONS | 959 |
| detailed column statistics | SYSCAT.COLDIST | 958 |
| constraint dependencies | SYSCAT.CONSTDEP | 965 |
| datatypes | SYSCAT.DATATYPES | 966 |
| event monitor definitions | SYSCAT.EVENTMONITORS | 969 |
| events currently monitored | SYSCAT.EVENTS | 971 |
| hierarchies(types, tables,views) | SYSCAT.FULLHIERARCHIES | 972 |
| function dependencies | SYSCAT.FUNCDEP | 973 |
| function mapping | SYSCAT.FUNCMAPPINGS | 976 |
| function mapping options | SYSCAT.FUNCMAPOPTIONS | 974 |
| function mapping parameter options | SYSCAT.FUNCMAPPARMOPTIONS | 975 |
| function parameters | SYSCAT.FUNCPARMS | 977 |
| hierarchies (types, tables, views) | SYSCAT.HIERARCHIES | 983 |
| Index columns | SYSCAT.INDEXCOLUSE | 985 |
| index dependencies | SYSCAT.INDEXDEP | 986 |
| index privileges | SYSCAT.INDEXAUTH | 984 |
| indexes | SYSCAT.INDEXES | 987 |
| index options | SYSCAT.INDEXOPTIONS | 990 |
| nodegroup definitions | SYSCAT.NODEGROUPS | 994 |
| nodegroup nodes | SYSCAT.NODEGROUPDEF | 993 |
| object mapping | SYSCAT.NAMEMAPPINGS | 992 |
| package dependencies | SYSCAT.PACKAGEDEP | 996 |
| package privileges | SYSCAT.PACKAGEAUTH | 995 |
| packages | SYSCAT.PACKAGES | 997 |
| partitioning maps | SYSCAT.PARTITIONMAPS | 1001 |
| pass-through privileges | SYSCAT.PASSTHRUAUTH | 1002 |
| procedure options | SYSCAT.PROCOPTIONS | 1005 |
| procedure parameter options | SYSCAT.PROCPARMOPTIONS | 1006 |
| procedure parameters | SYSCAT.PROCPARMS | 1007 |
| provides DB2 Universal Database for OS/390 compatibility | SYSIBM.SYSDUMMY1 | 949 |
| referential constraints | SYSCAT.REFERENCES | 1009 |

# Catalog Views

## "Roadmap" to Updatable Catalog Views

## SYSIBM.SYSDUMMY1

Contains one row. This view is available for applications that require compatibility with DB2 Universal Database for OS/390.

*Table 38. SYSCAT.DUMMY1 Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| IBMREQD | CHAR(1) | Y | |

## SYSCAT.ATTRIBUTES

Contains one row for each attribute (including inherited attributes where applicable) that is defined for a user-defined structured data type.

*Table 39. SYSCAT.ATTRIBUTES Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| TYPESCHEMA | VARCHAR(128) | | Qualified name of the strucutred data type that includes the attribute. |
| TYPENAME | VARCHAR(18) | | |
| ATTR_NAME | VARCHAR(18) | | Attribute name. |
| ATTR_TYPESCHEMA | VARCHAR(128) | | Contains the qualified name of the type of the attribute. |
| ATTR_TYPENAME | VARCHAR(18) | | |
| TARGET_TYPESCHEMA | VARCHAR(128) | | Qualified name of the target type, if the type of the attribute is REFERENCE. Null value if the type of the attribute is not REFERENCE. |
| TARGET_TYPENAME | VARCHAR(18) | | |
| SOURCE_TYPESCHEMA | VARCHAR(128) | | Qualified name of the data type in the data type hierarchy where the attribute was introduced. For non-inherited attributes, these columns are the same as TYPESCHEMA and TYPENAME. |
| SOURCE_TYPENAME | VARCHAR(18) | | |
| ORDINAL | SMALLINT | | Position of the attribute in the definition of the structured data type starting with zero. |
| LENGTH | INTEGER | | Maximum length of data. 0 for distinct types. The LENGTH column indicates precision for DECIMAL fields. |
| SCALE | SMALLINT | | Scale for DECIMAL fields; 0 if not DECIMAL. |
| CODEPAGE | SMALLINT | | Code page of the attribute. For character-string attributes not defined with FOR BIT DATA, the value is the database code page. For graphic-string attributes, the value is the DBCS code page implied by the (composite) database code page. Otherwise, the value is 0. |
| LOGGED | CHAR(1) | | Applies only to attributes whose type is LOB or distinct based on LOB (blank otherwise). Y = Attribute is logged. N = Attribute is not logged. |
| COMPACT | CHAR(1) | | Applies only to attributes whose type is LOB or distinct based on LOB (blank otherwise). Y = Attribute is compacted in storage. N = Attribute is not compacted. |

*Table 39. SYSCAT.ATTRIBUTES Catalog View (continued)*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| DL_FEATURES | CHAR(10) | | Applies to DATALINK type attributes only. Blank for REFERENCE type attributes. Null otherwise. Encodes various DATALINK features such as linktype, control mode, recovery, and unlink properties. |

## SYSCAT.BUFFERPOOLNODES

Contains a row for each node in the buffer pool for which the size of the
buffer pool on the node is different from the default size in
SYSCAT.BUFFERPOOLS column NPAGES.

*Table 40. SYSCAT.BUFFERPOOLNODES Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| BUFFERPOOLID | INTEGER | | Internal buffer pool identifier |
| NODENUM | SMALLINT | | Node Number |
| NPAGES | INTEGER | | Number of pages in this buffer pool on this node |

## SYSCAT.BUFFERPOOLS

Contains a row for every buffer pool in every nodegroup.

*Table 41. SYSCAT.BUFFERPOOLS Catalog View*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| BPNAME | VARCHAR(18) | | Name of buffer pool |
| BUFFERPOOLID | INTEGER | | Internal buffer pool identifier |
| NGNAME | VARCHAR(18) | Yes | Nodegroup name (NULL if the buffer pool exists on all nodes in the database) |
| NPAGES | INTEGER | | Number of pages in the buffer pool |
| PAGESIZE | INTEGER | | Pagesize for this buffer pool |
| ESTORE | CHAR(1) | | N = This buffer pool does not use extended storage. <br> Y = This buffer pool uses extended storage. |

## SYSCAT.CASTFUNCTIONS

Contains a row for each cast function. It does not include built-in cast functions.

*Table 42. SYSCAT.CASTFUNCTIONS Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| FROM_TYPESCHEMA | VARCHAR(128) | | Qualified name of the data type of the parameter. |
| FROM_TYPENAME | VARCHAR(18) | | |
| TO_TYPESCHEMA | VARCHAR(128) | | Qualified name of the data type of the result after casting. |
| TO_TYPENAME | VARCHAR(18) | | |
| FUNCSCHEMA | VARCHAR(128) | | Qualified name of the function. |
| FUNCNAME | VARCHAR(18) | | |
| SPECIFICNAME | VARCHAR(18) | | The name of the function instance. |
| ASSIGN_FUNCTION | CHAR(1) | | Y = Implicit assignment function<br>N = Not an assignment function |

## SYSCAT.CHECKS

Contains one row for each CHECK constraint.

Table 43. SYSCAT.CHECKS Catalog View

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| CONSTNAME | VARCHAR(18) | | Name of the check constraint (unique within a table.) |
| DEFINER | VARCHAR(128) | | Authorization ID under which the check constraint was defined. |
| TABSCHEMA | VARCHAR(128) | | Qualified name of the table to which this constraint applies. |
| TABNAME | VARCHAR(128) | | |
| CREATE_TIME | TIMESTAMP | | The time at which the constraint was defined. Used in resolving functions that are used in this constraint. No functions will be chosen that were created after the definition of the constraint. |
| QUALIFIER | VARCHAR(128) | | Value of the default schema at time of object definition. Used to complete any unqualified references. |
| TYPE | CHAR(1) | | Type of check constraint:<br><br>A = System generated check constraint for generated Always column<br><br>C = Check constraint |
| FUNC_PATH | VARCHAR(254) | | The current SQL path that was used when the constraint was created. |
| TEXT | CLOB(64K) | | The text of the CHECK clause. |

## SYSCAT.COLAUTH

Contains one or more rows for each user or group who is granted a column level privilege, indicating the type of privilege and whether or not it is grantable.

*Table 44. SYSCAT.COLAUTH Catalog View*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| GRANTOR | VARCHAR(128) | | Authorization ID of the user who granted the privileges or SYSIBM. |
| GRANTEE | VARCHAR(128) | | Authorization ID of the user or group who holds the privileges. |
| GRANTEETYPE | CHAR(1) | | U = Grantee is an individual user.<br>G = Grantee is a group. |
| TABSCHEMA | VARCHAR(128) | | Qualified name of the table or view. |
| TABNAME | VARCHAR(128) | | |
| COLNAME | VARCHAR(128) | | Name of the column to which this privilege applies. |
| COLNO | SMALLINT | | Number of this column in the table or view. |
| PRIVTYPE | CHAR(1) | | Indicates the type of privilege held on the table or view:<br>U = Update privilege<br>R = Reference privilege |
| GRANTABLE | CHAR(1) | | Indicates if the privilege is grantable.<br>G = Grantable<br>N = Not grantable |

## SYSCAT.COLCHECKS

Each row represents some column that is referenced by a CHECK constraint.

*Table 45. SYSCAT.COLCHECKS Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| CONSTNAME | VARCHAR(18) | | Name of the check constraint. (Unique within a table. May be system generated.) |
| TABSCHEMA | VARCHAR(128) | | Qualified name of table containing referenced column. |
| TABNAME | VARCHAR(128) | | |
| COLNAME | VARCHAR(128) | | Name of column. |
| USAGE | CHAR(1) | | R = Column is referenced in the check constraint. |
| | | | S = Column is a source column in the system generated column check constraint that supports a summary table. |
| | | | T = Column is a target column in the system generated column check constraint that supports a summary table. |

## SYSCAT.COLDIST

Contains detailed column statistics for use by the optimizer. Each row describes the Nth-most-frequent value of some column.

*Table 46. SYSCAT.COLDIST Catalog View*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| TABSCHEMA | VARCHAR(128) | | Qualified name of the table to which this entry applies. |
| TABNAME | VARCHAR(128) | | |
| COLNAME | VARCHAR(128) | | Name of the column to which this entry applies. |
| TYPE | CHAR(1) | | F = Frequency (most frequent value)<br>Q = Quantile value |
| SEQNO | SMALLINT | | • If TYPE = F, then N in this column identifies the Nth most frequent value.<br>• If TYPE = Q, then N in this column identifies the Nth quantile value. |
| COLVALUE | VARCHAR(254) | Yes | The data value, as a character literal or a null value. |
| VALCOUNT | BIGINT | | • If TYPE = F, then VALCOUNT is the number of occurrences of COLVALUE in the column.<br>• If TYPE = Q, then VALCOUNT is the number of rows whose value is less than or equal to COLVALUE. |
| DISTCOUNT | BIGINT | Yes | If TYPE = Q, this column records the number of distinct values that are less than or equal to COLVALUE (null if unavailable). |

## SYSCAT.COLOPTIONS

Each row contains column specific option values.

*Table 47. SYSCAT.COLOPTIONS Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| TABSCHEMA | VARCHAR(128) | | Qualifier of a nickname. |
| TABNAME | VARCHAR(128) | | Nickname for the column. |
| COLNAME | VARCHAR(128) | | Local column name. |
| OPTION | VARCHAR(128) | | Name of column option. |
| SETTING | VARCHAR(255) | | Values |

## SYSCAT.COLUMNS

Contains one row for each column (including inherited columns where applicable) that is defined for a table or view. All of the catalog views have entries in the SYSCAT.COLUMNS table.

*Table 48. SYSCAT.COLUMNS Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| TABSCHEMA | VARCHAR(128) | | Qualified name of the table or view that contains the column. |
| TABNAME | VARCHAR(128) | | |
| COLNAME | VARCHAR(128) | | Column name. |
| COLNO | SMALLINT | | Numerical place of column in table or view, beginning at zero. |
| TYPESCHEMA | VARCHAR(128) | | Contains the qualified name of the type, if the data type of the column is distinct. Otherwise TYPESCHEMA contains the value SYSIBM and TYPENAME contains the data type of the column (in long form, for example, CHARACTER). If FLOAT or FLOAT($n$) with $n$ greater than 24 is specified, TYPENAME is renamed to DOUBLE. If FLOAT($n$) with $n$ less than 25 is specified, TYPENAME is renamed to REAL. Also, NUMERIC is renamed to DECIMAL. |
| TYPENAME | VARCHAR(18) | | |
| LENGTH | INTEGER | | Maximum length of data. 0 for distinct types. The LENGTH column indicates precision for DECIMAL fields. |
| SCALE | SMALLINT | | Scale for DECIMAL fields; 0 if not DECIMAL. |
| DEFAULT | VARCHAR(254) | Yes | 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. May also be the keyword NULL. Values may be converted from what was specified as a default value. For example, date and time constants are presented in ISO format and cast-function names are qualified with schema name and the identifiers are delimited (see Note 3). Null value if a DEFAULT clause was not specified or the column is a view column. |

*Table 48. SYSCAT.COLUMNS Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| NULLS | CHAR(1) | | Y = Column is nullable.<br><br>N = Column is not nullable.<br><br>The value can be N for a view column that is derived from an expression or function. Nevertheless, such a column allows nulls when the statement using the view is processed with warnings for arithmetic errors.<br><br>See Note 1. |
| CODEPAGE | SMALLINT | | Code page of the column. For character-string columns not defined with the FOR BIT DATA attribute, the value is the database code page. For graphic-string columns, the value is the DBCS code page implied by the (composite) database code page. Otherwise, the value is 0. |
| LOGGED | CHAR(1) | | Applies only to columns whose type is LOB or distinct based on LOB (blank otherwise).<br><br>Y=Column is logged.<br><br>N=Column is not logged. |
| COMPACT | CHAR(1) | | Applies only to columns whose type is LOB or distinct based on LOB (blank otherwise).<br><br>Y = Column is compacted in storage.<br><br>N = Column is not compacted. |
| COLCARD | BIGINT | | Number of distinct values in the column; −1 if statistics are not gathered; −2 for inherited columns and columns of H-tables. |
| HIGH2KEY | VARCHAR(254) | Yes | Second highest value of the column. This field is empty if statistics are not gathered and for inherited columns and columns of H-tables. See Note 2. |
| LOW2KEY | VARCHAR(254) | Yes | Second lowest value of the column. This field is empty if statistics are not gathered and for inherited columns and columns of H-tables. See Note 2. |
| AVGCOLLEN | INTEGER | | Average column length. −1 if a long field or LOB, or statistics have not been collected; −2 for inherited columns and columns of H-tables. |

## SYSCAT.COLUMNS

*Table 48. SYSCAT.COLUMNS Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| KEYSEQ | SMALLINT | Yes | The column's numerical position within the table's primary key. This field is null for subtables and hierarchy tables. |
| PARTKEYSEQ | SMALLINT | Yes | The column's numerical position within the table's partitioning key. This field is null or 0 if the column is not part of the partitioning key. This field is also null for subtables and hierarchy tables. |
| NQUANTILES | SMALLINT | | Number of quantile values recorded in SYSCAT.COLDIST for this column; −1 if no statistics; −2 for inherited columns and columns of H-tables. |
| NMOSTFREQ | SMALLINT | | Number of most-frequent values recorded in SYSCAT.COLDIST for this column; −1 if no statistics; −2 for inherited columns and columns of H-tables. |
| NUMNULLS | BIGINT | | Contains the number of nulls in a column. −1 if statistics are not gathered. |
| TARGET_TYPESCHEMA | VARCHAR(128) | Yes | Qualified name of the target type, if the type of the column is REFERENCE. Null value if the type of the column is not REFERENCE. |
| TARGET_TYPENAME | VARCHAR(18) | Yes | |
| SCOPE_TABSCHEMA | VARCHAR(128) | Yes | Qualified name of the scope (target table), if the type of the column is REFERENCE. Null value if the type of the column is not REFERENCE or the scope is not defined. |
| SCOPE_TABNAME | VARCHAR(128) | Yes | |
| SOURCE_TABSCHEMA | VARCHAR(128) | | Qualified name of the table or view in the respective hierarchy where the column was introduced. For non-inherited columns, the values are the same as TBCREATOR and TBNAME. Null for columns of non-typed tables and views |
| SOURCE_TABNAME | VARCHAR(128) | | |

*Table 48. SYSCAT.COLUMNS Catalog View (continued)*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| DL_FEATURES | CHAR(10) | Yes | Applies to DATALINK type columns only. Null otherwise. Each character position is defined as follows:<br>1. Link type (U for URL)<br>2. Link control (F for file, N for no)<br>3. Integrity (A for all, N for none)<br>4. Read permission (F for file system, D for database)<br>5. Write permission (F for file system, B for blocked)<br>6. Recovery (Y for yes, N for no)<br>7. On unlink (R for restore, D for delete, N for not applicable)<br><br>Characters 8 through 10 are reserved for future use. |
| SPECIAL_PROPS | CHAR(8) | Yes | Applies to REFERENCE type columns only. Null otherwise. Each character position is defined as follows:<br>Object identifier (OID) column (Y for yes, N for no)<br>User generated or system generated (U for user, S for system) |
| HIDDEN | CHAR(1) | | Type of hidden column<br>S = System managed hidden column<br>Blank if column is not hidden |
| INLINE_LENGTH | INTEGER | | Any long VARCHAR or lob data smaller than or equal to INLINE_LENGTH is stored as VARCHAR data in the base table. Initial value is 0. |
| GENERATED | CHAR(1) | | Type of generated column<br>A = Column value is always generated<br>Blank if column is not generated |
| TEXT | CLOB(64K) | | Contains text of the definition of the generated column. |
| REMARKS | VARCHAR(254) | Yes | User-supplied comment. |

## SYSCAT.COLUMNS

*Table 48. SYSCAT.COLUMNS Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|

**Note:**

1. Starting with Version 2, value D (indicating not null with a default) is no longer used. Instead, use of WITH DEFAULT is indicated by a non-null value in the DEFAULT column.

2. Starting with Version 2, representation of numeric data has been changed to character literals. The size has been enlarged from 16 to 33 bytes.

3. For Version 2.1.0, cast-function names were not delimited and may still appear this way in the DEFAULT column. Also, some view columns included default values which will still appear in the DEFAULT column.

## SYSCAT.CONSTDEP

Contains a row for every dependency of a constraint on some other object.

*Table 49. SYSCAT.CONSTDEP Catalog View*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| CONSTNAME | VARCHAR(18) | | Name of the constraint. |
| TABSCHEMA | VARCHAR(128) | | Qualified name of the table to which the constraint applies. |
| TABNAME | VARCHAR(128) | | |
| BTYPE | CHAR(1) | | Type of object that the constraint depends on. Possible values: |
| | | | F = Function instance |
| | | | I = Index instance |
| | | | R = Structured type |
| BSCHEMA | VARCHAR(128) | | Qualified name of object that the constraint depends on. |
| BNAME | VARCHAR(18) | | |

## SYSCAT.DATATYPES

Contains a row for every data type, including built-in and user-defined types.

*Table 50. SYSCAT.DATATYPES Catalog View*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| TYPESCHEMA | VARCHAR(128) | | Qualified name of the data type (for built-in types, TYPESCHEMA is SYSIBM). |
| TYPENAME | VARCHAR(18) | | |
| DEFINER | VARCHAR(128) | | Authorization ID under which type was created. |
| SOURCESCHEMA | VARCHAR(128) | Yes | Qualified name of the source type for distinct types. Qualified name of the builtin type used as the reference type that is used as the representation for references to structured types. Null for other types. |
| SOURCENAME | VARCHAR(18) | Yes | |
| METATYPE | CHAR(1) | | S = System predefined type  T = Distinct type  R = Structured type |
| TYPEID | SMALLINT | | The system generated internal identifier of the data type. |
| SOURCETYPEID | SMALLINT | Yes | Internal type ID of source type (null for built-in types). For user-defined structured types, this is the internal type ID of the reference representation type. |
| LENGTH | INTEGER | | Maximum length of the type. 0 for system predefined parameterized types (for example, DECIMAL and VARCHAR). For user-defined structured types, this indicates the length of the reference representation type. |
| SCALE | SMALLINT | | Scale for distinct types or reference representation types based on the system predefined DECIMAL type. 0 for all other types (including DECIMAL itself). For user-defined structured types, this indicates the length of the reference representation type. |
| CODEPAGE | SMALLINT | | Code page for character and graphic distinct types or reference representation types; 0 otherwise. |
| CREATE_TIME | TIMESTAMP | | Creation time of the data type. |
| ATTRCOUNT | SMALLINT | | Number of attributes in data type. |
| INSTANTIABLE | CHAR(1) | | Y = Type can be instantiated.  N = Type can not be instantiated. |

*Table 50. SYSCAT.DATATYPES Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| WITH_FUNC_ACCESS | CHAR(1) | | Y = All the methods for this type can be invoked using function notation. |
| | | | N = Methods for this type can not be invoked using function notation. |
| FINAL | CHAR(1) | | Y = User-defined type can not have subtypes. |
| | | | N = User-defined type can have subtypes. |
| INLINE_LENGTH | INTEGER | | Length of structured type that can be kept with base table row. Always 0. |
| REMARKS | VARCHAR(254) | Yes | User-supplied comment, or null. |

## SYSCAT.DBAUTH

Records the database authorities held by users.

*Table 51. SYSCAT.DBAUTH Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| GRANTOR | VARCHAR(128) | | SYSIBM or authorization ID of the user who granted the privileges. |
| GRANTEE | VARCHAR(128) | | Authorization ID of the user or group who holds the privileges. |
| GRANTEETYPE | CHAR(1) | | U = Grantee is an individual user.<br>G = Grantee is a group. |
| DBADMAUTH | CHAR(1) | | Whether grantee holds DBADM authority over the database:<br>Y = Authority is held.<br>N = Authority is not held. |
| CREATETABAUTH | CHAR(1) | | Whether grantee can create tables in the database (CREATETAB):<br>Y = Privilege is held.<br>N = Privilege is not held. |
| BINDADDAUTH | CHAR(1) | | Whether grantee can create new packages in the database (BINDADD):<br>Y = Privilege is held.<br>N = Privilege is not held. |
| CONNECTAUTH | CHAR(1) | | Whether grantee can connect to the database (CONNECT):<br>Y = Privilege is held.<br>N = Privilege is not held. |
| NOFENCEAUTH | CHAR(1) | | Whether grantee holds privilege to create non-fenced functions.<br>Y = Privilege is held.<br>N = Privilege is not held. |
| IMPLSCHEMAAUTH | CHAR(1) | | Whether grantee can implicitly create schemas in the database (IMPLICIT_SCHEMA):<br>Y = Privilege is held.<br>N = Privilege is not held. |
| LOADAUTH | CHAR(1) | | Whether grantee has the privilege to use the LOAD utility:<br>Y = Privilege is held.<br>N = Privilege is not held. |

## SYSCAT.EVENTMONITORS

Contains a row for every event monitor that has been defined.

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| EVMONNAME | VARCHAR(18) | | Name of event monitor. |
| DEFINER | VARCHAR(128) | | Authorization ID of definer of event monitor. |
| TARGET_TYPE | CHAR(1) | | The type of the target to which event data is written. Values:<br>F = File<br>P = Pipe |
| TARGET | VARCHAR(246) | | Name of the target to which event data is written. Absolute pathname of file, or absolute name of pipe. |
| MAXFILES | INTEGER | Yes | Maximum number of event files that this event monitor permits in an event path. Null if there is no maximum, or if the target-type is not FILE. |
| MAXFILESIZE | INTEGER | Yes | Maximum size (in 4K pages) that each event file can reach before the event monitor creates a new file. Null if there is no maximum, or if the target-type is not FILE. |
| BUFFERSIZE | INTEGER | Yes | Size of buffers (in 4K pages) used by event monitors with file targets; otherwise null. |
| IO_MODE | CHAR(1) | Yes | Mode of file I/O.<br>B = Blocked<br>N = Not blocked<br>Null if target-type is not FILE. |
| WRITE_MODE | CHAR(1) | Yes | Indicates how this monitor handles existing event data when the monitor is activated. Values:<br>A = Append<br>R = Replace<br>Null if target-type is not FILE. |
| AUTOSTART | CHAR(1) | | The event monitor will be activated automatically when the database starts.<br>Y = Yes<br>N = No |
| NODENUM | SMALLINT | | The number of the partition (or node) on which the event monitor runs and logs events. |

**SYSCAT.EVENTMONITORS**

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| MONSCOPE | CHAR(1) | | Monitoring scope:<br>L = Local<br>G = Global |
| REMARKS | VARCHAR(254) | Yes | Reserved for future use. |

## SYSCAT.EVENTS

Contains a row for every event that is being monitored. An event monitor, in general, monitors multiple events.

*Table 52. SYSCAT.EVENTS Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| EVMONNAME | VARCHAR(18) | | Name of event monitor that is monitoring this event. |
| TYPE | VARCHAR(18) | | Type of event being monitored. Possible values: <br> DATABASE <br> CONNECTIONS <br> TABLES <br> STATEMENTS <br> TRANSACTIONS <br> DEADLOCKS <br> TABLESPACES |
| FILTER | CLOB(32K) | Yes | The full text of the WHERE-clause that applies to this event. |

## 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 53. SYSCAT.FULLHIERARCHIES Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| METATYPE | CHAR(1) | | Encodes the type of relationship:<br>R = Between structured types<br>U = Between typed tables<br>W = Between typed views |
| SUB_SCHEMA | VARCHAR(128) | | Qualified name of subtype, subtable or subview. |
| SUB_NAME | VARCHAR(128) | | |
| SUPER_SCHEMA | VARCHAR(128) | | Qualified name of supertype, supertable or superview. |
| SUPER_NAME | VARCHAR(128) | | |
| ROOT_SCHEMA | VARCHAR(128) | | Qualified name of the table, view or type that is at the root of the hierarchy. |
| ROOT_NAME | VARCHAR(128) | | |

## SYSCAT.FUNCDEP

Each row represents a dependency of a function on some other object.

*Table 54. SYSCAT.FUNCDEP Catalog View*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| FUNCSCHEMA | VARCHAR(128) | | Qualified name of the function which has dependencies on another object. |
| FUNCNAME | VARCHAR(18) | | |
| BTYPE | CHAR(1) | | Type of object that the function is dependent on.<br><br>A = Alias<br>F = Function instance<br>O = Privilege dependency on all subtables or subviews in a table or view hierarchy<br>R = Structured type<br>S = Summary table<br>T = Table<br>U = Typed table<br>V = View<br>W = Typed view<br>X = Index extension |
| BSCHEMA | VARCHAR(128) | | Qualified name of the object that the function has a dependency on. |
| BNAME | VARCHAR(128) | | |
| TABAUTH | SMALLINT | Yes | If BTYPE = O, S, T, U, V or W then it encodes the privileges on the table or view that are required by the dependent function. Otherwise null. |

## SYSCAT.FUNCMAPOPTIONS

Each row contains function mapping option values.

*Table 55. SYSCAT.FUNCMAPOPTIONS Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| FUNCTION_MAPPING | VARCHAR(18) | | Function mapping name. |
| OPTION | VARCHAR(128) | | Name of the function mapping option. |
| SETTING | VARCHAR(255) | | Value. |

## SYSCAT.FUNCMAPPARMOPTIONS

Each row contains function mapping parameter option values.

*Table 56. SYSCAT.FUNCMAPPARMOPTIONS Catalog View*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| FUNCTION_MAPPING | VARCHAR(18) | | Name of function mapping. |
| ORDINAL | SMALLINT | | Position of parameter |
| LOCATION | CHAR(1) | | L = Local<br>R = Remote |
| OPTION | VARCHAR(128) | | Name of the function mapping parameter option. |
| SETTING | VARCHAR(255) | | Value. |

## SYSCAT.FUNCMAPPINGS

Each row contains function mappings.

*Table 57. SYSCAT.FUNCMAPPINGS Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| FUNCTION_MAPPING | VARCHAR(18) | | Name of function mapping (may be system generated). |
| FUNCSCHEMA | VARCHAR(128) | Yes | Funcion schema. Null if system built-in function. |
| FUNCNAME | VARCHAR(1024) | Yes | Name of the local function (built-in or user-defined) |
| FUNCID | INTEGER | Yes | Internally assigned identifier. |
| SPECIFICNAME | VARCHAR(18) | Yes | Name of the local function instance. |
| DEFINER | VARCHAR(128) | | Authorization ID under which this mapping was created. |
| WRAPNAME | VARCHAR(128) | Yes | Wrapper name to which the mapping is applied. |
| SERVERNAME | VARCHAR(128) | Yes | Name of the data source. |
| SERVERTYPE | VARCHAR(30) | Yes | Type of data source to which mapping is applied. |
| SERVERVERSION | VARCHAR(18) | Yes | Version of the server type to which mapping is applied. |
| CREATE_TIME | TIMESTAMP | Yes | Time at which the mapping is created. |
| REMARKS | VARCHAR(254) | Yes | User supplied comment, or null. |

## SYSCAT.FUNCPARMS

Contains a row for every parameter or result of a function defined in SYSCAT.FUNCTIONS.

*Table 58. SYSCAT.FUNCPARMS Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| FUNCSCHEMA | VARCHAR(128) | | Qualified function name. |
| FUNCNAME | VARCHAR(18) | | |
| SPECIFICNAME | VARCHAR(18) | | The name of the function instance (may be system-generated). |
| ROWTYPE | CHAR(1) | | P = Parameter<br>R = Result before casting<br>C = Result after casting |
| ORDINAL | SMALLINT | | If ROWTYPE = P, the parameter's numerical position within the function signature. Otherwise 0. |
| PARMNAME | VARCHAR(128) | | Name of parameter or result column, or null if no name exists. |
| TYPESCHEMA | VARCHAR(128) | | Qualified name of data type of parameter or result. |
| TYPENAME | VARCHAR(18) | | |
| LENGTH | INTEGER | | Length of parameter or result. 0 if parameter or result is a distinct type. See Note 1. |
| SCALE | SMALLINT | | Scale of parameter or result. 0 if parameter or result is a distinct type. See Note 1. |
| CODEPAGE | SMALLINT | | Code page of parameter. 0 denotes either not applicable or a column for character data declared with the FOR BIT DATA attribute. |
| CAST_FUNCID | INTEGER | Yes | Internal function ID. |
| AS_LOCATOR | CHAR(1) | | Y = Parameter or result is passed in the form of a locator.<br>N = Not passed in the form of a locator. |
| TARGET_TYPESCHEMA | VARCHAR(128) | | Qualified name of the target type, if the type of the parameter or result is REFERENCE. |
| TARGET_TYPENAME | VARCHAR(18) | | Null value if the type of the parameter or result is not REFERENCE. |
| SCOPE_TABSCHEMA | VARCHAR(128) | | Qualified name of the scope (target table), if the type of the parameter or result is REFERENCE. Null value if the type of the parameter or result is not REFERENCE or the scope is not defined. |
| SCOPE_TABNAME | VARCHAR(128) | | |

## SYSCAT.FUNCPARMS

*Table 58. SYSCAT.FUNCPARMS Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| TRANSFORM_GRPNAME | VARCHAR(18) | | Name of transform group for a user-defined type function parameter. |

**Note:**

1. 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.FUNCTIONS

Contains a row for each user-defined function (scalar, table or source). Does not include built-in functions.

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| FUNCSCHEMA | VARCHAR(128) | | Qualified function name. |
| FUNCNAME | VARCHAR(18) | | |
| SPECIFICNAME | VARCHAR(18) | | The name of the function instance (may be system-generated). |
| DEFINER | VARCHAR(128) | | Authorization ID of function definer. |
| FUNCID | INTEGER | | Internally-assigned function ID. |
| RETURN_TYPE | SMALLINT | | Internal type code of return type of function. |
| ORIGIN | CHAR(1) | | B = Built-in<br>E = User-defined, external<br>U = User-defined, based on a source<br>S = System-generated |
| TYPE | CHAR(1) | | C = Column function<br>S = Scalar function<br>T = Table function |
| METHOD | CHAR(1) | | N = Not a method |
| EFFECT | CHAR(2) | | Blanks if not a method |
| PARM_COUNT | SMALLINT | | Number of function parameters. |
| PARM_SIGNATURE | VARCHAR(180) FOR BIT DATA | | Concatenation of up to 90 parameter types, in internal format. Zero length if function takes no parameters. |
| CREATE_TIME | TIMESTAMP | | Timestamp of function creation. Set to 0 for Version 1 functions. |
| QUALIFIER | VARCHAR(128) | | Value of default schema at object definition time. |
| WITH_FUNC_ACCESS | CHAR(1) | | Reserved for future use. |
| TYPE_PRESERVING | CHAR(1) | | Reserved for future use. |
| VARIANT | CHAR(1) | | Y = Variant (results may differ)<br>N = Invariant (results are consistent)<br>Blank if ORIGIN is not E |

## SYSCAT.FUNCTIONS

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| SIDE_EFFECTS | CHAR(1) | | E = Function has external side-effects (number of invocations is important)<br><br>N = No side-effects<br><br>Blank if ORIGIN is not E |
| FENCED | CHAR(1) | | Y = Fenced<br><br>N = Not fenced<br><br>Blank if ORIGIN is not E |
| NULLCALL | CHAR(1) | | Y = CALLED ON NULL INPUT<br><br>N = RETURNS NULL ON NULL INPUT (function result is implicitly null if operand(s) are null).<br><br>Blank if ORIGIN is not E. |
| CAST_FUNCTION | CHAR(1) | | Y = This is a cast function<br><br>N = This is not a cast function |
| ASSIGN_FUNCTION | CHAR(1) | | Y = Implicit assignment function<br><br>N = Not an assignment function |
| SCRATCHPAD | CHAR(1) | | Y = This function has a scratch pad.<br><br>N = This function does not have a scratch pad.<br><br>Blank if ORIGIN is not E |
| FINAL_CALL | CHAR(1) | | Y = Final call is made to this function at run time end-of-statement.<br><br>N = No final call is made.<br><br>Blank if ORIGIN is not E |
| PARALLELIZABLE | CHAR(1) | | Y = Function can be executed in parallel.<br><br>N = Function cannot be executed in parallel.<br><br>Blank if ORIGIN is not E |
| CONTAINS_SQL | CHAR(1) | | Indicates whether an external function contains SQL.<br><br>N = Function does not contain SQL statements.<br><br>Blank if ORIGIN is not E |

## SYSCAT.FUNCTIONS

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| DBINFO | CHAR(1) | | Indicates whether a DBINFO parameter is passed to an external function.<br><br>Y = DBINFO is passed.<br>N = DBINFO is not passed.<br>Blank if ORIGIN is not E |
| RESULT_COLS | SMALLINT | | For a table function (TYPE=T) contains the number of columns in the result table; otherwise contains 1. |
| LANGUAGE | CHAR(8) | | Implementation language of function body. Possible values are C, JAVA, OLE or OLEDB. Blank if ORIGIN is not E. |
| IMPLEMENTATION | VARCHAR(254) | Yes | If ORIGIN = E, identifies the path/module/function that implements this function. If ORIGIN = U and the source function is built-in, this column contains the name and signature of the source function. Null otherwise. |
| CLASS | VARCHAR(128) | Yes | If LANGUAGE = JAVA, identifies the class that implements this function. Null otherwise. |
| JAR_ID | VARCHAR(128) | Yes | If LANGUAGE = JAVA, identifies the jar file that implements this function. Null otherwise. |
| PARM_STYLE | CHAR(8) | | Indicates the parameter style declared in the CREATE FUNCTION statement. Values:<br><br>DB2SQL<br>DB2GENRL<br>JAVA<br>Blank if ORIGIN is not E |
| SOURCE_SCHEMA | VARCHAR(128) | Yes | If ORIGIN = U and the source function is a user-defined function, contains the qualified name of the source function. If ORIGIN = U and the source function is built-in, SOURCE_SCHEMA is 'SYSIBM' and SOURCE_SPECIFIC is 'N/A for built-in'. Null if ORIGIN is not U. |
| SOURCE_SPECIFIC | VARCHAR(18) | Yes | |
| IOS_PER_INVOC | DOUBLE | | Estimated number of I/Os per invocation; -1 if not known (0 default). |
| INSTS_PER_INVOC | DOUBLE | | Estimated number of instructions per invocation; -1 if not known (450 default). |
| IOS_PER_ARGBYTE | DOUBLE | | Estimated number of I/O's per input argument byte; -1 if not known (0 default). |

## SYSCAT.FUNCTIONS

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| INSTS_PER_ARGBYTE | DOUBLE | | Estimated number of instructions per input argument byte; -1 if not known (0 default). |
| PERCENT_ARGBYTES | SMALLINT | | Estimated average percent of input argument bytes that the function will actually read; -1 if not known (100 default). |
| INITIAL_IOS | DOUBLE | | Estimated number of I/O's performed the first/last time the function is invoked; -1 if not known (0 default). |
| INITIAL_INSTS | DOUBLE | | Estimated number of instructions executed the first/last time the function is invoked; -1 if not known (0 default). |
| CARDINALITY | BIGINT | | The predicted cardinality of a table function. −1 if not known or if function is not a table function. |
| IMPLEMENTED | CHAR(1) | | Reserved for future use. |
| SELECTIVITY | DOUBLE | | Reserved for future use. |
| OVERRIDEN_FUNCID | INTEGER | Yes | Reserved for future use. |
| SUBJECT_TYPESCHEMA | VARCHAR(128) | Yes | Subject type schema for the user defined method. |
| SUBJECT_TYPENAME | VARCHAR(18) | Yes | Subject type name for the user defined method. |
| FUNC_PATH | VARCHAR(254) | Yes | Function path at the time the function was defined. |
| BODY | CLOB(1M) | Yes | Reserved for future use. |
| REMARKS | VARCHAR(254) | Yes | User-supplied comment, or null. |

## 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 59. SYSCAT.HIERARCHIES Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| METATYPE | CHAR(1) | | Encodes the type of relationship: <br> R = Between structured types <br> U = Between typed tables <br> W = Between typed views |
| SUB_SCHEMA | VARCHAR(128) | | Qualified name of subtype, subtable, or subview. |
| SUB_NAME | VARCHAR(128) | | |
| SUPER_SCHEMA | VARCHAR(128) | | Qualified name of supertype, supertable, or superview. |
| SUPER_NAME | VARCHAR(128) | | |
| ROOT_SCHEMA | VARCHAR(128) | | Qualified name of the table, view or type that is at the root of the hierarchy. |
| ROOT_NAME | VARCHAR(128) | | |

## SYSCAT.INDEXAUTH

Contains a row for every privilege held on an index.

*Table 60. SYSCAT.INDEXAUTH Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| GRANTOR | VARCHAR(128) | | Authorization ID of the user who granted the privileges. |
| GRANTEE | VARCHAR(128) | | Authorization ID of the user or group who holds the privileges. |
| GRANTEETYPE | CHAR(1) | | U = Grantee is an individual user. G = Grantee is a group. |
| INDSCHEMA | VARCHAR(128) | | Name of the index. |
| INDNAME | VARCHAR(18) | | |
| CONTROLAUTH | CHAR(1) | | Whether grantee holds CONTROL privilege over the index: Y = Privilege is held. N = Privilege is not held. |

## SYSCAT.INDEXCOLUSE

Lists all columns that participate in an index.

*Table 61. SYSCAT.INDEXCOLUSE Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| INDSCHEMA | VARCHAR(128) | | Qualified name of the index. |
| INDNAME | VARCHAR(18) | | |
| COLNAME | VARCHAR(128) | | Name of the column. |
| COLSEQ | SMALLINT | | Numeric position of the column in the index (initial position = 1). |
| COLORDER | CHAR(1) | | Order of the values in this column in the index. Values:<br><br>A = Ascending<br><br>D = Descending<br><br>I = INCLUDE column(ordering ignored) |

## SYSCAT.INDEXDEP

Each row represents a dependency of a function on some other object.

*Table 62. SYSCAT.INDEXDEP Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| INDSCHEMA | VARCHAR(128) | | Qualified name of the index which has dependencies on another object. |
| INDNAME | VARCHAR(18) | | |
| BTYPE | CHAR(1) | | Type of object that the index is dependent on.<br><br>A = Alias<br><br>F = Function instance<br><br>O = Privilege dependency on all subtables or subviews in a table or view hierarchy<br><br>R = Structured type<br><br>S = Summary table<br><br>T = Table<br><br>U = Typed table<br><br>V = View<br><br>W = Typed view<br><br>X = Index extension |
| BSCHEMA | VARCHAR(128) | | Qualified name of the object that the index has a dependency on. |
| BNAME | VARCHAR(128) | | |
| TABAUTH | SMALLINT | Yes | If BTYPE = O, S, T, U, V or W then it encodes the privileges on the table or view that are required by the dependent index. Otherwise null. |

## SYSCAT.INDEXES

Contains one row for each index (including inherited indexes where applicable) that is defined for a table.

*Table 63. SYSCAT.INDEXES Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| INDSCHEMA | VARCHAR(128) | | Name of the index. |
| INDNAME | VARCHAR(18) | | |
| DEFINER | VARCHAR(128) | | User who created the index. |
| TABSCHEMA | VARCHAR(128) | | Qualified name of the table or nickname on which the index is defined. |
| TABNAME | VARCHAR(128) | | |
| COLNAMES | VARCHAR(640) | | List of column names, each preceded by + or − to indicate ascending or descending order respectively. Warning: This column will be removed in the future. Use "SYSCAT.INDEXCOLUSE" on page 985 for this information. |
| UNIQUERULE | CHAR(1) | | Unique rule:<br>D = Duplicates allowed<br>P = Primary index<br>U = Unique entries only allowed |
| MADE_UNIQUE | CHAR(1) | | 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 will revert to non-unique.<br>N = Index remains as it was created. |
| COLCOUNT | SMALLINT | | Number of columns in the key plus the number of include columns if any. |
| UNIQUE_COLCOUNT | SMALLINT | | 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) |
| INDEXTYPE | CHAR(4) | | Type of index.<br>CLUS = Clustering<br>REG = Regular |
| ENTRYTYPE | CHAR(1) | | H = An index on a hierarchy table (H-table)<br>L = Logical index on a typed table<br>blank if an index on an untyped table |

## SYSCAT.INDEXES

*Table 63. SYSCAT.INDEXES Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| PCTFREE | SMALLINT | | 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. |
| IID | SMALLINT | | Internal index ID. |
| NLEAF | INTEGER | | Number of leaf pages; −1 if statistics are not gathered. |
| NLEVELS | SMALLINT | | Number of index levels; −1 if statistics are not gathered. |
| FIRSTKEYCARD | BIGINT | | Number of distinct first key values; −1 if statistics are not gathered. |
| FIRST2KEYCARD | BIGINT | | Number of distinct keys using the first two columns of the index (−1 if no statistics or inapplicable) |
| FIRST3KEYCARD | BIGINT | | Number of distinct keys using the first three columns of the index (−1 if no statistics or inapplicable) |
| FIRST4KEYCARD | BIGINT | | Number of distinct keys using the first four columns of the index (−1 if no statistics or inapplicable) |
| FULLKEYCARD | BIGINT | | Number of distinct full key values; −1 if statistics are not gathered. |
| CLUSTERRATIO | SMALLINT | | 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). |
| CLUSTERFACTOR | DOUBLE | | 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. |
| SEQUENTIAL_PAGES | INTEGER | | 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 | | 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.) |
| USER_DEFINED | SMALLINT | | 1 if this index was defined by a user and has not been dropped; otherwise 0. |

*Table 63. SYSCAT.INDEXES Catalog View (continued)*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| SYSTEM_REQUIRED | SMALLINT | | 1 if this index is required for primary key or unique key constraint, OR if this is the index on the object identifier (OID) column of a typed table.<br><br>2 if this index is required for primary key or unique key constraint, AND this is the index on the object identifier (OID) column of a typed table.<br><br>0 otherwise. |
| CREATE_TIME | TIMESTAMP | | Time when the index was created. |
| STATS_TIME | TIMESTAMP | Yes | Last time when any change was made to recorded statistics for this index. Null if no statistics available. |
| PAGE_FETCH_PAIRS | VARCHAR(254) | | 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.) |
| MINPCTUSED | SMALLINT | | If not zero, then on-line index reorganization is enabled and the value is the threshold of minimum used space before merging pages. |
| REVERSE_SCANS | CHAR(1) | | Y = Index supports reverse scans<br>N = Index does not support reverse scans |
| INTERNAL_FORMAT | CHAR(1) | | Encodes the internal representation of the index. |
| REMARKS | VARCHAR(254) | Yes | User-supplied comment, or null. |

## SYSCAT.INDEXOPTIONS

Each row contains index specific option values.

*Table 64. SYSCAT.INDEXOPTIONS Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| INDSCHEMA | VARCHAR(128) | | Schema name of the index. |
| INDNAME | VARCHAR(18) | | Local name of the index. |
| OPTION | VARCHAR(128) | | Name of the index option. |
| SETTING | VARCHAR(255) | | Value. |

## SYSCAT.KEYCOLUSE

Lists all columns that participate in a key (including inherited primary or unique keys where applicable) defined by a unique, primary key, or foreign key constraint.

*Table 65. SYSCAT.KEYCOLUSE Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| CONSTNAME | VARCHAR(18) | | Name of the constraint (unique within a table). |
| TABSCHEMA | VARCHAR(128) | | Qualified name of the table containing the column. |
| TABNAME | VARCHAR(128) | | |
| COLNAME | VARCHAR(128) | | Name of the column. |
| COLSEQ | SMALLINT | | Numeric position of the column in the key (initial position=1). |

## SYSCAT.NAMEMAPPINGS

Each row represents the mapping between logical objects and the corresponding implementation objects that implement the logical objects.

*Table 66. SYSCAT.NAMEMAPPINGS Catalog View*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| TYPE | CHAR(1) | | C = Column<br>I = Index<br>U = Typed table |
| LOGICAL_SCHEMA | VARCHAR(128) | | Qualified name of the logical object. |
| LOGICAL_NAME | VARCHAR(128) | | |
| LOGICAL_COLNAME | VARCHAR(128) | Yes | If TYPE = C, then the name of the logical column. Otherwise null. |
| IMPL_SCHEMA | VARCHAR(128) | | Qualified name of the implementation object that implements the logical object. |
| IMPL_NAME | VARCHAR(128) | | |
| IMPL_COLNAME | VARCHAR(128) | Yes | If TYPE = C, then the name of the implementation column. Otherwise null. |

## SYSCAT.NODEGROUPDEF

Contains a row for each partition that is contained in a nodegroup.

*Table 67. SYSCAT.NODEGROUPDEF Catalog View*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| NGNAME | VARCHAR(18) | | The name of the nodegroup that contains the partition (or node). |
| NODENUM | SMALLINT | | The partition (or node) number of a partition contained in the nodegroup. A valid partition number is between 0 and 999 inclusive. |
| IN_USE | CHAR(1) | | Status of the partition (or node).<br><br>A = The newly added partition is not in the partitioning map but the containers for the table spaces in the nodegroup are created. The partition is added to the partitioning map when a Redistribute Nodegroup operation is successfully completed.<br><br>D = The partition will be dropped when a Redistribute Nodegroup operation is completed.<br><br>T = The newly added partition is not in the partitioning map and it was added using the WITHOUT TABLESPACES clause. Containers must be specifically added to the table spaces for the nodegroup.<br><br>Y = The partition is in the partitioning map. |

## SYSCAT.NODEGROUPS

Contains a row for each nodegroup.

*Table 68. SYSCAT.NODEGROUPS Catalog View*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| NGNAME | VARCHAR(18) | | Name of the nodegroup. |
| DEFINER | VARCHAR(128) | | Authorization ID of the nodegroup definer. |
| PMAP_ID | SMALLINT | | Identifier of the partitioning map in SYSCAT.PARTITIONMAPS. |
| REBALANCE_PMAP_ID | SMALLINT | | Identifier of the partitioning map currently being used for redistribution. Value is -1 if redistribution is currently not in progress. |
| CREATE_TIME | TIMESTAMP | | Creation time of nodegroup. |
| REMARKS | VARCHAR(254) | Yes | User-provided comment. |

## SYSCAT.PACKAGEAUTH

Contains a row for every privilege held on a package.

Table 69. SYSCAT.PACKAGEAUTH Catalog View

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| GRANTOR | VARCHAR(128) | | Authorization ID of the user who granted the privileges. |
| GRANTEE | VARCHAR(128) | | Authorization ID of the user or group who holds the privileges. |
| GRANTEETYPE | CHAR(1) | | U = Grantee is an individual user. G = Grantee is a group. |
| PKGSCHEMA | VARCHAR(128) | | Name of the package on which the privileges are held. |
| PKGNAME | CHAR(8) | | |
| CONTROLAUTH | CHAR(1) | | Indicates whether grantee holds CONTROL privilege on the package: Y = Privilege is held. N = Privilege is not held. |
| BINDAUTH | CHAR(1) | | Indicates whether grantee holds BIND privilege on the package: Y = Privilege is held. N = Privilege is not held. |
| EXECUTEAUTH | CHAR(1) | | Indicates whether grantee holds EXECUTE privilege on the package: Y = Privilege is held. N = Privilege is not held. |

## SYSCAT.PACKAGEDEP

Contains a row for each dependency that packages have on indexes, tables, views, functions, aliases, types, and hierarchies.

*Table 70. SYSCAT.PACKAGEDEP Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| PKGSCHEMA | VARCHAR(128) | | Name of the package. |
| PKGNAME | CHAR(8) | | |
| BINDER | VARCHAR(128) | Yes | Binder of the package. |
| BTYPE | CHAR(1) | | Type of object BNAME:<br><br>A = Alias<br>D = Server definition<br>F = Function instance<br>I = Index<br>M = Function mapping<br>N = Nickname<br>O = Privilege dependency on all subtables or subviews in a table or view hierarchy<br>P = Page size<br>R = Structured type<br>S = Summary table<br>T = Table<br>U = Typed table<br>V = View<br>W = Typed view |
| BSCHEMA | VARCHAR(128) | | Qualified name of an object on which the package is dependent. |
| BNAME | VARCHAR(128) | | |
| TABAUTH | SMALLINT | Yes | If BTYPE is O, S, T, U, V or W then it encodes the privileges that are required by this package (Select, Insert, Delete, Update). |

**Note:**

1. When a depended-on function-instance is dropped, the package is placed into an "inoperative" state from which it must be explicitly rebound. When any other depended-on object is dropped, the package is placed into an "invalid" state from which the system will attempt to rebind it automatically when a package is first referenced.

## SYSCAT.PACKAGES

Contains a row for each package that has been created by binding an application program.

*Table 71. SYSCAT.PACKAGES Catalog View*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| PKGSCHEMA | VARCHAR(128) | | Name of the package. |
| PKGNAME | CHAR(8) | | |
| BOUNDBY | VARCHAR(128) | | Authorization ID (OWNER) of the binder of the package. |
| DEFINER | VARCHAR(128) | | Userid under which package was bound. |
| DEFAULT_SCHEMA | VARCHAR(128) | | Default schema (QUALIFIER) name used for unqualified names in static SQL statements. |
| VALID | CHAR(1) | | Y = Valid<br><br>N = Not valid<br><br>X = Package is inoperative because some function instance that it depends on has been dropped. Explicit rebind is needed. See Note 1 on "SYSCAT.PACKAGEDEP" on page 996 |
| UNIQUE_ID | CHAR(8) | | Internal date and time information indicating when the package was first created. |
| TOTAL_SECT | SMALLINT | | Total number of sections in the package. |
| FORMAT | CHAR(1) | | Date and time format associated with the package:<br><br>0 = Format associated with country code of the database<br><br>1 = USA date and time<br><br>2 = EUR date, EUR time<br><br>3 = ISO date, ISO time<br><br>4 = JIS date, JIS time<br><br>5 = LOCAL date, LOCAL time |
| ISOLATION | CHAR(2) | Yes | Isolation level:<br><br>RR = Repeatable read<br><br>RS = Read stability<br><br>CS = Cursor stability<br><br>UR = Uncommitted read |

## SYSCAT.PACKAGES

*Table 71. SYSCAT.PACKAGES Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| BLOCKING | CHAR(1) | Yes | Cursor blocking option:<br>N = No blocking<br>U = Block unambiguous cursors<br>B = Block all cursors |
| INSERT_BUF | CHAR(1) | | Insert option used during bind:<br>Y = Inserts are buffered<br>N = Inserts are not buffered |
| LANG_LEVEL | CHAR(1) | Yes | LANGLEVEL value used during BIND:<br>0 = SAA1<br>1 = SQL92E or MIA |
| FUNC_PATH | VARCHAR(254) | | The SQL path used by the last BIND command for this package. This is used as the default path for REBIND. SYSIBM for pre-Version 2 packages. |
| QUERYOPT | INTEGER | | Optimization class under which this package was bound. Used for rebind. The classes are: 0, 1, 3, 5 and 9. |
| EXPLAIN_LEVEL | CHAR(1) | | Indicates whether Explain was requested using the EXPLAIN or EXPLSNAP bind option.<br>P = Plan Selection level<br>Blank if 'No' Explain requested |
| EXPLAIN_MODE | CHAR(1) | | Value of EXPLAIN bind option:<br>Y = Yes (static)<br>N = No<br>A = All (static and dynamic) |
| EXPLAIN_SNAPSHOT | CHAR(1) | | Value of EXPLSNAP bind option:<br>Y = Yes (static)<br>N = No<br>A = All (static and dynamic) |
| SQLWARN | CHAR(1) | | Are positive SQLCODEs resulting from dynamic SQL statements returned to the application?<br>Y = Yes<br>N = No, they are suppressed. |

*Table 71. SYSCAT.PACKAGES Catalog View (continued)*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| SQLMATHWARN | CHAR(1) | | Value of database configuration parameter DFT_SQLMATHWARN at time of bind. Are arithmetic errors and retrieval conversion errors in static SQL statements handled as nulls with a warning?<br><br>Y = Yes<br><br>N = No, they are suppressed. |
| EXPLICIT_BIND_TIME | TIMESTAMP | | The time at which this package was last explicitly bound or rebound. When the package is implicitly rebound, no function instance will be selected that was created later than this time. |
| LAST_BIND_TIME | TIMESTAMP | | Time at which the package last explicitly or implicitly bound or rebound. |
| CODEPAGE | SMALLINT | | Application codepage at bind time (-1 if not known). |
| DEGREE | CHAR(5) | | Indicates the limit on intra-partition parallelism (as a bind option) when package was bound.<br><br>1 = No intra-partition parallelism.<br><br>2 - 32 767 = Degree of intra-partition parallelism.<br><br>ANY = Degree was determined by the database manager. |
| MULTINODE_PLANS | CHAR(1) | | Y = Package was bound in a multiple partition environment.<br><br>N =Package was bound in a single partition environment. |
| INTRA_PARALLEL | CHAR(1) | | Indicates the use of intra-partition parallelism by static SQL statements within the package.<br><br>Y = one or more static SQL statement in package uses intra-partition parallelism.<br><br>N = no static SQL statement in package uses intra-partition parallelism.<br><br>F = one or more static SQL statement in package can use intra-partition parallelism; this parallelism has been disabled for use on a system that is not configured for intra-partition parallelism. |
| VALIDATE | CHAR(1) | | B = All checking must be performed during BIND<br><br>R = Reserved |

## SYSCAT.PACKAGES

*Table 71. SYSCAT.PACKAGES Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| DYNAMICRULES | CHAR(1) | | B = Dynamic SQL statements are handled like static SQL statements at run time; binder's authid is used.<br><br>R = Dynamic SQL statements are handled like dynamic SQL statements at run time; executer's authid is used.<br><br>Initial value is R. |
| SQLERROR | CHAR(1) | | Indicates SQLERROR option on the most recent subcommand that bound or rebound the package.<br>  C = Reserved<br>  N = No package |
| REFRESHAGE | DECIMAL (20,6) | | Timestamp duration indicating the maximum length of time between when a REFRESH TABLE statement is run for a summary table and when the summary table is used in place of a base table. |
| REMARKS | VARCHAR(254) | Yes | User-supplied comment, or null. |

## SYSCAT.PARTITIONMAPS

Contains a row for each partitioning map that is used to distribute the rows of tables among the partitions in a nodegroup, based on hashing the tables partitioning key.

*Table 72. SYSCAT.PARTITIONMAPS Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| PMAP_ID | SMALLINT | | Identifier of the partitioning map. |
| PARTITIONMAP | LONG VARCHAR FOR BIT DATA | | The actual partitioning map, a vector of 4 096 two-byte integers for a multiple node nodegroup. For a single node nodegroup, there is one entry denoting the partition (or node) number of the single node. |

## SYSCAT.PASSTHRUAUTH

This catalog view contains information about authorizations to query data sources in pass-through sessions. A constraint on the base table requires that the values in SERVER correspond to the values in the SERVER column of SYSCAT.SERVERS. None of the fields in SYSCAT.PASSTHRUAUTH are nullable.

*Table 73. Columns in SYSCAT.PASSTHRUAUTH Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| GRANTOR | VARCHAR(128) | | Authorization ID of the user who granted the privilege. |
| GRANTEE | VARCHAR(128) | | Authorization ID of the user or group who holds the privilege. |
| GRANTEETYPE | CHAR(1) | | A letter that specifies the type of grantee: U = Grantee is an individual user. G = Grantee is a group. |
| SERVERNAME | VARCHAR(128) | | Name of the data source that the user or group is being granted authorization to. |

## SYSCAT.PROCEDURES

Contains a row for each stored procedure that is created.

*Table 74. SYSCAT.PROCEDURES Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| PROCSCHEMA | VARCHAR(128) | | Qualified procedure name. |
| PROCNAME | VARCHAR(128) | | |
| SPECIFICNAME | VARCHAR(18) | | The name of the procedure instance (may be system generated). |
| PROCEDURE_ID | INTEGER | | Internal ID of stored procedure. |
| DEFINER | VARCHAR(128) | | Authorization of the procedure definer. |
| PARM_COUNT | SMALLINT | | Number of procedure parameters. |
| PARM_SIGNATURE | VARCHAR(180) FOR BIT DATA | | Concatenation of up to 90 parameter types, in internal format. Zero length if procedure takes no parameters. |
| ORIGIN | CHAR(1) | | Always 'E' = User defined, external |
| CREATE_TIME | TIMESTAMP | | Timestamp of procedure registration. |
| DETERMINISTIC | CHAR(1) | | Y = Results are deterministic.<br>N = Results are not deterministic. |
| FENCED | CHAR(1) | | Y = Fenced<br>N = Not Fenced |
| NULLCALL | CHAR(1) | | Always Y = NULLCALL |
| LANGUAGE | CHAR(8) | | Implementation language of procedure body. Possible values are:<br>C<br>COBOL<br>JAVA |
| IMPLEMENTATION | VARCHAR(254) | Yes | Identifies the path/module/function (LANGUAGE = C or COBOL) or method (LANGUAGE = JAVA) that implements the procedure. |
| CLASS | VARCHAR(128) | Yes | If LANGUAGE = JAVA then it identifies the class that implements this procedure. Null otherwise. |
| JAR_ID | VARCHAR(128) | Yes | If LANGUAGE = JAVA then identifies the jar file that implements this procedure. Null otherwise. |

## SYSCAT.PROCEDURES

*Table 74. SYSCAT.PROCEDURES Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| PARM_STYLE | CHAR(8) | | DB2DARI = Language is C<br>DB2GENRL = Language is Java<br>DB2SQL = Language is C or COBOL<br>JAVA = Language is Java<br>GENERAL = Language is C or COBOL<br>GNLRNULL = Language is C or COBOL |
| CONTAINS_SQL | CHAR(1) | | Indicates whether a procedure contains SQL.<br>C = Contains simple SQL statements only with no subqueries<br>M = Contains SQL statements that modify data<br>N = Does not contain SQL statements<br>R = Contains read-only SQL statements |
| DBINFO | CHAR(1) | | Indicates whether a DBINFO parameter is passed to the procedure<br>N = DBINFO is not passed<br>Y = DBINFO is passed |
| PROGRAM_TYPE | CHAR(1) | | Indicates how procedure is invoked.<br>M = Main<br>S = Subroutine |
| RESULT_SETS | SMALLINT | | Estimated upper limit of returned result sets. |
| VALID | CHAR(1) | | Reserved for future use. |
| TEXT_BODY_OFFSET | INTEGER | | Reserved for future use. |
| TEXT | CLOB (1M) | Yes | Reserved for future use. |
| REMARKS | VARCHAR(254) | Yes | User supplied comment, or null. |

## SYSCAT.PROCOPTIONS

Each row contains procedure specific option values.

*Table 75. SYSCAT.PROCOPTIONS Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| PROCSCHEMA | VARCHAR(128) | | Qualifier for the stored procedure name or nickname. |
| PROCNAME | VARCHAR(128) | | Name or nickname of the stored procedure. |
| OPTION | VARCHAR(128) | | Name of the stored procedure option. |
| SETTING | VARCHAR(255) | | Value of the stored procedure option. |

## SYSCAT.PROCPARMOPTIONS

Each row contains procedure parameter specific option values.

*Table 76. SYSCAT.PROCPARMOPTIONS Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| PROCSCHEMA | VARCHAR(128) | | Qualified procedure name or nickname. |
| PROCNAME | VARCHAR(128) | | |
| ORDINAL | SMALLINT | | The parameter's numerical position within the procedure signature. |
| OPTION | VARCHAR(128) | | Name of the stored procedure option. |
| SETTING | VARCHAR(255) | | Value. |

## SYSCAT.PROCPARMS

Contains a row for each parameter of a stored procedure.

Table 77. SYSCAT.PROCPARMS Catalog View

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| PROCSCHEMA | VARCHAR(128) | | Qualified procedure name. |
| PROCNAME | VARCHAR(128) | | |
| SPECIFICNAME | VARCHAR(18) | | The name of the procedure instance (may be system generated). |
| SERVERNAME | VARCHAR(128) | Yes | Name of the data source on which the stored procedure resides. |
| ORDINAL | SMALLINT | | The parameter's numerical position within the procedure signature. |
| PARMNAME | VARCHAR(18) | | Parameter name. |
| TYPESCHEMA | VARCHAR(128) | | Qualified name of data type of the parameter. |
| TYPENAME | VARCHAR(18) | | |
| TYPEID | SMALLINT | Yes | Internal type ID. |
| SOURCETYPEID | SMALLINT | Yes | Internal type ID of source type. Null for built-in types. |
| NULLS | CHAR(1) | | federated database nullable rule:<br>  Y = Nullable<br>  N = Not nullable |
| LENGTH | INTEGER | | Length of the parameter. |
| SCALE | SMALLINT | | Scale of the parameter. |
| PARM_MODE | VARCHAR(5) | | IN = Input<br>OUT = Output<br>INOUT = Input/output |
| CODEPAGE | SMALLINT | | Code page of parameter. 0 denotes either not applicable or a parameter for character data declared with the FOR BIT DATA attribute. |
| DBCS_CODEPAGE | SMALLINT | Yes | DBCS codepage. Null for numeric fields. |
| AS_LOCATOR | CHAR(1) | | Always 'N' |
| TARGET_TYPESCHEMA | VARCHAR(128) | Yes | If type of parameter is reference then contains qualified name of target rowtype. Null otherwise. |
| TARGET_TYPENAME | VARCHAR(18) | | |

## SYSCAT.PROCPARMS

*Table 77. SYSCAT.PROCPARMS Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| SCOPE_TABSCHEMA | VARCHAR(128) | Yes | If type of parameter is reference then contains qualified name of scope (target table). Null otherwise. |
| SCOPE_TABNAME | VARCHAR(128) | | |

## SYSCAT.REFERENCES

Contains a row for each defined referential constraint.

Table 78. SYSCAT.REFERENCES Catalog View

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| CONSTNAME | VARCHAR(18) | | Name of constraint. |
| TABSCHEMA | VARCHAR(128) | | Qualified name of the constraint. |
| TABNAME | VARCHAR(128) | | |
| DEFINER | VARCHAR(128) | | User who created the constraint. |
| REFKEYNAME | VARCHAR(18) | | Name of parent key. |
| REFTABSCHEMA | VARCHAR(128) | | Name of the parent table. |
| REFTABNAME | VARCHAR(128) | | |
| COLCOUNT | SMALLINT | | Number of columns in the foreign key. |
| DELETERULE | CHAR(1) | | Delete rule: <br> A = NO ACTION <br> C = CASCADE <br> N = SET NULL <br> R = RESTRICT |
| UPDATERULE | CHAR(1) | | Update rule: <br> A = NO ACTION <br> R = RESTRICT |
| CREATE_TIME | TIMESTAMP | | The timestamp when the referential constraint was defined. |
| FK_COLNAMES | VARCHAR (640) | | List of foreign key column names. Warning: This column will be removed in the future. Use "SYSCAT.KEYCOLUSE" on page 991 for this information. |
| PK_COLNAMES | VARCHAR (640) | | List of parent key column names. Warning: This column will be removed in the future. Use "SYSCAT.KEYCOLUSE" on page 991 for this information. |

**Note:**

1. The SYSCAT.REFERENCES view is based on the SYSIBM.SYSRELS table from Version 1.

## SYSCAT.REVTYPEMAPPINGS

Each row contains reverse data type mappings (mappings from data types defined locally to data source data types). No data in this version. Defined for possible future use with data type mappings.

*Table 79. SYSCAT.REVTYPEMAPPINGS Catalog View*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| TYPE_MAPPING | VARCHAR(18) | | Name of the reverse type mapping (may be system-generated). |
| TYPESCHEMA | VARCHAR(128) | Yes | Schema name of the type. Null for system built-in types. |
| TYPENAME | VARCHAR(18) | | Name of the local type in a reverse type mapping. |
| TYPEID | SMALLINT | | Type identifier. |
| SOURCETYPEID | SMALLINT | | Source type identifier. |
| DEFINER | VARCHAR(128) | | Authorization ID under which this type mapping was created. |
| LOWER_LEN | INTEGER | Yes | Lower bound of the length/precision of the local type. |
| UPPER_LEN | INTEGER | Yes | Upper bound of the length/precision of the local type. If null then the system determines the best length/precision attribute. |
| LOWER_SCALE | SMALLINT | Yes | Lower bound of the scale for local decimal data types. |
| UPPER_SCALE | SMALLINT | Yes | Upper bound of the scale for local decimal data types. If null, then the system determines the best scale attribute. |
| S_OPR_P | CHAR(2) | Yes | Relationship between local scale and local precision. Basic comparison operators can be used. A null indicates that no specific relationship is required. |
| BIT_DATA | CHAR(1) | Yes | Y = Type is for bit data. N = Type is not for bit data. NULL = This is not a character data type or that the system determines the bit data attribute. |
| WRAPNAME | VARCHAR(128) | Yes | Mapping applies to this data access protocol. |
| SERVERNAME | VARCHAR(128) | Yes | Name of the data source. |
| SERVERTYPE | VARCHAR(30) | Yes | Mapping applies to this type of data source. |

*Table 79. SYSCAT.REVTYPEMAPPINGS Catalog View (continued)*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| SERVERVERSION | VARCHAR(18) | Yes | Mapping applies to this version of SERVERTYPE. |
| REMOTE_TYPESCHEMA | VARCHAR(128) | Yes | Schema name of the remote type. |
| REMOTE_TYPENAME | VARCHAR(128) | | Name of the data type as defined on the data source(s). |
| REMOTE_META_TYPE | CHAR(1) | Yes | S = Remote type is a system built-in type. T = Remote type is a distinct type. |
| REMOTE_LENGTH | INTEGER | Yes | Maximum number of digits for remote decimal type, and maximum number of characters for remote character type. Otherwise null. |
| REMOTE_SCALE | SMALLINT | Yes | Maximum number of digits allowed to the right of the decimal point (for remote decimal types). Otherwise null. |
| REMOTE_BIT_DATA | CHAR(1) | Yes | Y = Type is for bit data. N = Type is not for bit data. NULL = This is not a character data type or that the system determines the bit data attribute. |
| USER_DEFINED | CHAR(1) | | Defined by user. |
| CREATE_TIME | TIMESTAMP | | Time when this mapping was created. |
| REMARKS | VARCHAR(254) | Yes | User supplied comments, or null. |

## SYSCAT.SCHEMAAUTH

Contains one or more rows for each user or group who is granted a privilege on a particular schema in the database. All schema privileges for a single schema granted by a specific grantor to a specific grantee appear in a single row.

*Table 80. SYSCAT.SCHEMAAUTH Catalog View*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| GRANTOR | VARCHAR(128) | | Authorization ID of the user who granted the privileges or SYSIBM. |
| GRANTEE | VARCHAR(128) | | Authorization ID of the user or group who holds the privileges. |
| GRANTEETYPE | CHAR(1) | | U = Grantee is an individual user.<br>G = Grantee is a group. |
| SCHEMANAME | VARCHAR(128) | | Name of the schema. |
| ALTERINAUTH | CHAR(1) | | Indicates whether grantee holds ALTERIN privilege on the schema:<br>Y = Privilege is held.<br>G = Privilege is held and grantable.<br>N = Privilege is not held. |
| CREATEINAUTH | CHAR(1) | | Indicates whether grantee holds CREATEIN privilege on the schema:<br>Y = Privilege is held.<br>G = Privilege is held and grantable.<br>N = Privilege is not held. |
| DROPINAUTH | CHAR(1) | | Indicates whether grantee holds DROPIN privilege on the schema:<br>Y = Privilege is held.<br>G = Privilege is held and grantable.<br>N = Privilege is not held. |

## SYSCAT.SCHEMATA

Contains a row for each schema.

*Table 81. SYSCAT.SCHEMATA Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| SCHEMANAME | VARCHAR(128) | | Name of the schema. |
| OWNER | VARCHAR(128) | | Authorization id of the schema. The value for implicitly created schemas is SYSIBM. |
| DEFINER | VARCHAR(128) | | User who created the schema. |
| CREATE_TIME | TIMESTAMP | | Timstamp indicating when the object was created. |
| REMARKS | VARCHAR(254) | Yes | User-provided comment. |

## SYSCAT.SERVEROPTIONS

Each row contains configuration options at the server level.

*Table 82. Columns in SYSCAT.SERVEROPTIONS Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| WRAPNAME | VARCHAR(128) | Yes | Wrapper name. |
| SERVERNAME | VARCHAR(128) | Yes | Name of the server. |
| SERVERTYPE | VARCHAR(30) | Yes | Server type. |
| SERVERVERSION | VARCHAR(18) | Yes | Server version. |
| CREATE_TIME | TIMESTAMP | | Time when entry is 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) | Yes | User supplied comments, or null. |

## SYSCAT.SERVERS

Each row represents a data source. Catalog entries are not necessary for tables that are stored in the same instance that contains this catalog table.

*Table 83. Columns in SYSCAT.SERVERS Catalog View*

| Name | Data Type | Nullable | Description |
|---|---|---|---|
| WRAPNAME | VARCHAR(128) | | Wrapper name. |
| SERVERNAME | VARCHAR(128) | | Name of data source as it is known to the system. |
| SERVERTYPE | VARCHAR(30) | Yes | Type of data source (always uppercase). |
| SERVERVERSION | VARCHAR(18) | Yes | Version of data source. |
| REMARKS | VARCHAR(254) | Yes | User supplied comments, or null. |

## SYSCAT.STATEMENTS

Contains one or more rows for each SQL statement in each package in the database.

*Table 84. SYSCAT.STATEMENTS Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| PKGSCHEMA | VARCHAR(128) | | Name of the package. |
| PKGNAME | CHAR(8) | | |
| 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 | SMALLINT | | Always 1. |
| TEXT | CLOB (64K) | | Text of the SQL statement. |

## SYSCAT.TABAUTH

Contains one or more rows for each user or group who is granted a privilege on a particular table or view in the database. All the table privileges for a single table or view granted by a specific grantor to a specific grantee appear in a single row.

*Table 85. SYSCAT.TABAUTH Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| GRANTOR | VARCHAR(128) | | Authorization ID of the user who granted the privileges or SYSIBM. |
| GRANTEE | VARCHAR(128) | | Authorization ID of the user or group who holds the privileges. |
| GRANTEETYPE | CHAR(1) | | U = Grantee is an individual user. <br> G = Grantee is a group. |
| TABSCHEMA | VARCHAR(128) | | Qualified name of the table or view. |
| TABNAME | VARCHAR(128) | | |
| CONTROLAUTH | CHAR(1) | | Indicates whether grantee holds CONTROL privilege on the table or view: <br> Y = Privilege is held. <br> N = Privilege is not held. |
| ALTERAUTH | CHAR(1) | | Indicates whether grantee holds ALTER privilege on the table: <br> Y = Privilege is held. <br> N = Privilege is not held. <br> G = Privilege is held and grantable. |
| DELETEAUTH | CHAR(1) | | Indicates whether grantee holds DELETE privilege on the table or view: <br> Y = Privilege is held. <br> N = Privilege is not held. <br> G = Privilege is held and grantable. |
| INDEXAUTH | CHAR(1) | | Indicates whether grantee holds INDEX privilege on the table: <br> Y = Privilege is held. <br> N = Privilege is not held. <br> G = Privilege is held and grantable. |

## SYSCAT.TABAUTH

*Table 85. SYSCAT.TABAUTH Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| INSERTAUTH | CHAR(1) | | Indicates whether grantee holds INSERT privilege on the table or view: <br><br>Y = Privilege is held. <br><br>N = Privilege is not held. <br><br>G = Privilege is held and grantable. |
| SELECTAUTH | CHAR(1) | | Indicates whether grantee holds SELECT privilege on the table or view: <br><br>Y = Privilege is held. <br><br>N = Privilege is not held. <br><br>G = Privilege is held and grantable. |
| REFAUTH | CHAR(1) | | Indicates whether grantee holds REFERENCE privilege on the table or view: <br><br>Y = Privilege is held. <br><br>N = Privilege is not held. <br><br>G = Privilege is held and grantable. |
| UPDATEAUTH | CHAR(1) | | Indicates whether grantee holds UPDATE privilege on the table or view: <br><br>Y = Privilege is held. <br><br>N = Privilege is not held. <br><br>G = Privilege is held and grantable. |

## SYSCAT.TABCONST

Each row represents a table constraint of type CHECK, UNIQUE, PRIMARY KEY, or FOREIGN KEY.

*Table 86. SYSCAT.TABCONST Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| CONSTNAME | VARCHAR(18) | | Name of the constraint (unique within a table). |
| TABSCHEMA | VARCHAR(128) | | Qualified name of the table to which this constraint applies. |
| TABNAME | VARCHAR(128) | | |
| DEFINER | VARCHAR(128) | | Authorization ID under which the constraint was defined. |
| TYPE | CHAR(1) | | Indicates the constraint type: <br> F = FOREIGN KEY <br> K = CHECK <br> P = PRIMARY KEY <br> U = UNIQUE |
| REMARKS | VARCHAR(254) | Yes | User-supplied comment, or null. |

## SYSCAT.TABLES

Contains one row for each table, view, nickname or alias that is created. All of the catalog tables and views have entries in the SYSCAT.TABLES catalog view.

*Table 87. SYSCAT.TABLES Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| TABSCHEMA | VARCHAR(128) | | Qualified name of the table, view, nickname or alias. |
| TABNAME | VARCHAR(128) | | |
| DEFINER | VARCHAR(128) | | User who created the table, view, nickname or alias. |
| TYPE | CHAR(1) | | The type of object: |
| | | | A = Alias |
| | | | H = Hierarchy table |
| | | | N = Nickname |
| | | | S = Summary table |
| | | | T = Table |
| | | | U = Typed table |
| | | | V = View |
| | | | W = Typed view |
| STATUS | CHAR(1) | | The type of object: |
| | | | N = Normal table, view, alias or nickname |
| | | | C = Check pending on table or nickname |
| | | | X = Inoperative view or nickname |
| BASE_TABSCHEMA | VARCHAR(128) | Yes | If TYPE = A, these columns identify the table, view, alias or nickname that is referenced by this alias; otherwise they are null. |
| BASE_TABNAME | VARCHAR(128) | Yes | |
| ROWTYPESCHEMA | VARCHAR(128) | Yes | Contains the qualified name of the rowtype of this table, where applicable. Null otherwise. |
| ROWTYPENAME | VARCHAR(18) | | |
| CREATE_TIME | TIMESTAMP | | The timestamp indicating when the object was created. |
| STATS_TIME | TIMESTAMP | Yes | Last time when any change was made to recorded statistics for this table. Null if no statistics available. |
| COLCOUNT | SMALLINT | | Number of columns in table. |
| TABLEID | SMALLINT | | Internal table identifier. |
| TBSPACEID | SMALLINT | | Internal identifier of primary table space for this table. |

*Table 87. SYSCAT.TABLES Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| CARD | BIGINT | | Total number of rows in the table. For tables in a table hierarchy, its the number of rows at the given level of the hierarchy −1 if statistics are not gathered or the row describes a view or alias; −2 for hierarchy tables (H-tables) |
| NPAGES | INTEGER | | Total number of pages on which the rows of the table exist; −1 if statistics are not gathered or the row describes a view or alias; −2 for subtables or H-tables. |
| FPAGES | INTEGER | | Total number of pages; −1 if statistics are not gathered or the row describes a view or alias; −2 for subtables or H-tables. |
| OVERFLOW | INTEGER | | Total number of overflow records in the table; −1 if statistics are not gathered or the row describes a view or alias; −2 for subtables or H-tables. |
| TBSPACE | VARCHAR(18) | Yes | Name of primary table space for the table. If no other table space is specified, all parts of the table are stored in this table space. Null for aliases and views. |
| INDEX_TBSPACE | VARCHAR(18) | Yes | Name of table space that holds all indexes created on this table. Null for aliases and views, 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(18) | Yes | Name of table space that holds all long data (LONG or LOB column types) for this table. Null for aliases and views, 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 | Yes | Number of parent tables of this table (the number of referential constraints in which this table is a dependent). |
| CHILDREN | SMALLINT | Yes | Number of dependent tables of this table (the number of referential constraints in which this table is a parent). |
| SELFREFS | SMALLINT | Yes | Number of self-referencing referential constraints for this table (the number of referential constraints in which this table is both a parent and a dependent). |

*Table 87. SYSCAT.TABLES Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| KEYCOLUMNS | SMALLINT | Yes | Number of columns in the primary key of the table. |
| KEYINDEXID | SMALLINT | Yes | Index ID of the primary index. This field is null or 0 if there is no primary key. |
| KEYUNIQUE | SMALLINT | | Number of unique constraints (other than primary key) defined on this table. |
| CHECKCOUNT | SMALLINT | | Number of check constraints defined on this table. |
| DATACAPTURE | CHAR(1) | | Y = Table participates in data replication<br><br>N = Does not participate |
| CONST_CHECKED | CHAR(32) | | Byte 1 represents foreign key constraints. Byte 2 represents check constraints. Byte 5 represents summary table. Other bytes are reserved. Encodes constraint information on checking. Values:<br><br>Y = Checked by system<br><br>U = Checked by user<br><br>N = Not checked (pending)<br><br>W = Was in a 'U' state when the table was placed in check pending (pending) |
| PMAP_ID | SMALLINT | Yes | Identifier of the partitioning map used by this table. Null for aliases and views. |
| PARTITION_MODE | CHAR(1) | | Mode used for tables in a partitioned database.<br><br>H = Hash on the partitioning key<br><br>R = Table replicated across database partitions<br><br>Blank for aliases, views and tables in single partition nodegroups with no partitioning key defined. Also blank for nicknames. |
| LOG_ATTRIBUTE | CHAR(1) | | 0 = Default logging<br><br>1 = Table created not logged initially |
| PCTFREE | SMALLINT | | Percentage of each page to be reserved for future inserts. Can be changed by ALTER TABLE. |

*Table 87. SYSCAT.TABLES Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| APPEND_MODE | CHAR(1) | | Controls how rows are inserted on pages: <br><br> N = New rows are inserted into existing spaces if available <br><br> Y = New rows are appended at end of data <br><br> Initial value is N. |
| REFRESH | CHAR(1) | | Refresh mode <br><br> D = Deferred <br><br> I = Immeidate <br><br> O = Once <br><br> Blank if not a summary table |
| REFRESH_TIME | TIMESTAMP | Yes | For REFRESH = D or O, timestamp of the REFRESH TABLE statement that last refreshed the data. Otherwise null. |
| LOCKSIZE | CHAR(1) | | Indicates preferred lock granularity for tables when accessed by DML statements. Only applies to tables. Possible values are: <br><br> R = Row <br><br> T = Table <br><br> Blank if not applicable <br><br> Initial value is R. |
| VOLATILE | CHAR(1) | | C = Cardinality of the table is volatile <br><br> Blank if not applicable |
| REMARKS | VARCHAR(254) | Yes | User-provided comment. |

## SYSCAT.TABLESPACES

Contains a row for each table space.

*Table 88. SYSCAT.TABLESPACES Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| TBSPACE | VARCHAR(18) | | Name of table space. |
| DEFINER | VARCHAR(128) | | Authorization ID of table space definer. |
| CREATE_TIME | TIMESTAMP | | Creation time of table space. |
| TBSPACEID | INTEGER | | Internal table space identifier. |
| TBSPACETYPE | CHAR(1) | | The type of the table space:<br>S = System managed space<br>D = Database managed space |
| DATATYPE | CHAR(1) | | Type of data that can be stored:<br>A = All types of permanent data<br>L = Long data only<br>T = Temporary tables only |
| EXTENTSIZE | INTEGER | | Size of 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 prefetch is performed. |
| OVERHEAD | DOUBLE | | Controller overhead and disk seek and latency time in milliseconds. |
| TRANSFERRATE | DOUBLE | | Time to read one page of size PAGESIZE into the buffer. |
| PAGESIZE | INTEGER | | Size (in bytes) of pages in the table space. |
| NGNAME | VARCHAR(18) | | Name of the nodegroup for the table space. |
| BUFFERPOOLID | INTEGER | | ID of buffer pool used by this tablespace (1 indicates default buffer pool). |
| DROP_RECOVERY | CHAR(1) | | N = table is not recoverable after a DROP TABLE statement<br>Y = table is recoverable after a DROP TABLE statement |
| REMARKS | VARCHAR(254) | Yes | User-provided comment. |

## SYSCAT.TABOPTIONS

Each row contains option associated with a remote table.

*Table 89. SYSCAT.TABOPTIONS Catalog View*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| TABSCHEMA | VARCHAR(128) | | Qualified name of table, view, alias or nickname. |
| TABNAME | VARCHAR(128) | | |
| OPTION | VARCHAR(128) | | Name of the table, view, alias or nickname option. |
| SETTING | VARCHAR(255) | | Value. |

## SYSCAT.TBSPACEAUTH

Containes one row for each user or group whi is granted USE privilege on a particular table space.

*Table 90. SYSCAT.TBSPACEAUTH Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| GRANTOR | VARCHAR(128) | | Authorization ID of the user who granted USE privilege or SYSIBM. |
| GRANTEE | VARCHAR(128) | | Authorization ID of the user or group who hold the privileges. |
| GRANTEETYPE | CHAR(1) | | U = Grantee is an individual user. |
| | | | G = Grantee is a group. |
| TBSPACE | VARCHAR(18) | | Name of table space. |
| USEAUTH | CHAR(1) | | Indicates whether grantee holds USE privilege on the table space: |
| | | | G = Privilege is held and grantable. |
| | | | N = Privilege is not held. |
| | | | Y = Privilege is held. |

## SYSCAT.TRIGDEP

Contains a row for every dependency of a trigger on some other object.

*Table 91. SYSCAT.TRIGDEP Catalog View*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| TRIGSCHEMA | VARCHAR(128) | | Qualified name of the trigger. |
| TRIGNAME | VARCHAR(18) | | |
| BTYPE | CHAR(1) | | Type of object BNAME: |
| | | |     A = Alias |
| | | |     F = Function instance |
| | | |     N = Nickname |
| | | |     O = Privilege dependency on all subtables or subviews in a table or view hierarchy |
| | | |     R = Structured type |
| | | |     S = Summary table |
| | | |     T = Table |
| | | |     U = Typed table |
| | | |     V = View |
| | | |     W = Typed view |
| | | |     X = Index extension |
| BSCHEMA | VARCHAR(128) | | Qualified name of object depended on by a trigger. |
| BNAME | VARCHAR(128) | | |
| TABAUTH | SMALLINT | Yes | If BTYPE= O, S, T, U, V or W encodes the privileges on the table or view that are required by this trigger; otherwise null. |

## SYSCAT.TRIGGERS

Contains one row for each trigger. For table hierarchies, each trigger is recorded only at the level of the hierarchy where it was created.

*Table 92. SYSCAT.TRIGGERS Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| TRIGSCHEMA | VARCHAR(128) | | Qualified name of the trigger. |
| TRIGNAME | VARCHAR(18) | | |
| DEFINER | VARCHAR(128) | | Authorization ID under which the trigger was defined. |
| TABSCHEMA | VARCHAR(128) | | Qualified name of the table to which this trigger applies. |
| TABNAME | VARCHAR(128) | | |
| TRIGTIME | CHAR(1) | | Time when triggered actions are applied to the base table, relative to the event that fired the trigger:<br><br>A = Trigger applied after event<br>B = Trigger applied before event |
| TRIGEVENT | CHAR(1) | | Event that fires the trigger.<br><br>I = Insert<br>D = Delete<br>U = Update |
| GRANULARITY | CHAR(1) | | Trigger is executed once per:<br><br>S = Statement<br>R = Row |
| VALID | CHAR(1) | | Y = Trigger is valid<br>X = Trigger is inoperative; must be re-created. |
| CREATE_TIME | TIMESTAMP | | Time at which the trigger was defined. Used in resolving functions and types. |
| QUALIFIER | VARCHAR(128) | | Contains value of the default schema at the time of object definition. |
| FUNC_PATH | VARCHAR(254) | | Function path at the time the trigger was defined. Used in resolving functions and types. |
| TEXT | CLOB(64K) | | The full text of the CREATE TRIGGER statement, exactly as typed. |
| REMARKS | VARCHAR(254) | Yes | User-supplied comment, or null. |

## SYSCAT.TYPEMAPPINGS

Each row contains a user-defined mapping of a remote built-in data type to a local built-in data type.

*Table 93. SYSCAT.TYPEMAPPINGS Catalog View*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| TYPE_MAPPING | VARCHAR(18) | | Name of the type mapping (may be system-generated). |
| TYPESCHEMA | VARCHAR(128) | Yes | Schema name of the type. Null for system built-in types. |
| TYPENAME | VARCHAR(18) | | Name of the local type in a data type mapping. |
| TYPEID | SMALLINT | | Type identifier. |
| SOURCETYPEID | SMALLINT | | Source type identifier. |
| DEFINER | VARCHAR(128) | | Authorization ID under which this type mapping was created. |
| LENGTH | INTEGER | Yes | Maximum length or precision of the data type. If null, the system determines the best length/precision. |
| SCALE | SMALLINT | Yes | Scale for DECIMAL fields. If null, the system determines the best scale attribute. |
| BIT_DATA | CHAR(1) | Yes | Y = Type is for bit data. <br><br> N = Type is not for bit data. <br><br> NULL = This is not a character data type or that the system determines the bit data attribute. |
| WRAPNAME | VARCHAR(128) | Yes | Mapping applies to this data access protocol. |
| SERVERNAME | VARCHAR(128) | Yes | Name of the data source. |
| SERVERTYPE | VARCHAR(30) | Yes | Mapping applies to this type of data source. |
| SERVERVERSION | VARCHAR(18) | Yes | Mapping applies to this version of SERVERTYPE. |
| REMOTE_TYPESCHEMA | VARCHAR(128) | Yes | Schema name of the remote type. |
| REMOTE_TYPENAME | VARCHAR(128) | | Name of the data type as defined on the data source(s). |
| REMOTE_META_TYPE | CHAR(1) | Yes | S = Remote type is a system built-in type. <br><br> T = Remote type is a distinct type. |

## SYSCAT.TYPEMAPPINGS

*Table 93. SYSCAT.TYPEMAPPINGS Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| REMOTE_LOWER_LEN | INTEGER | Yes | Lower bound of the length/precision of the remote decimal type. For character data types, this field indicates the number of character. |
| REMOTE_UPPER_LEN | INTEGER | Yes | Upper bound of the length/precision of the remote decimal type. For character data types, this field indicates the number of character. |
| REMOTE_LOWER_SCALE | SMALLINT | Yes | Lower bound of the scale of the remote type. |
| REMOTE_UPPER_SCALE | SMALLINT | Yes | Upper bound of the scale of the remote type. |
| REMOTE_S_OPR_P | CHAR(2) | Yes | Relationship between remote scale and remote precision. Basic comparison operators can be used. A null indicated that no specific relationship is required. |
| REMOTE_BIT_DATA | CHAR(1) | Yes | Y = Type is for bit data.<br><br>N = Type is not for bit data.<br><br>NULL = This is not a character data type or that the system determines the bit data attribute. |
| USER_DEFINED | CHAR(1) | | Definition supplied by user. |
| CREATE_TIME | TIMESTAMP | | Time when this mapping was created. |
| REMARKS | VARCHAR(254) | Yes | User supplied comments, or null. |

## SYSCAT.USEROPTIONS

Each row contains server specific option values.

*Table 94. SYSCAT.USEROPTIONS Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| AUTHID | VARCHAR(128) | | Local authorization ID (always uppercase) |
| SERVERNAME | VARCHAR(128) | | Name of the server for which the user is defined. |
| OPTION | VARCHAR(128) | | Name of the user options. |
| SETTING | VARCHAR(255) | | Value. |

## SYSCAT.VIEWDEP

Contains a row for every dependency of a view or a summary table on some other object. Also encodes how privileges on this view depend on privileges on underlying tables and views.

*Table 95. SYSCAT.VIEWDEP Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| VIEWSCHEMA | VARCHAR(128) | | Name of the view or the name of a summary table having dependencies on a base table. |
| VIEWNAME | VARCHAR(128) | | |
| DTYPE | CHAR(1) | | S = Summary table |
| | | | V = View (untyped) |
| | | | W = Typed view |
| DEFINER | VARCHAR(128) | Yes | Authorization ID of the creator of the view. |
| BTYPE | CHAR(1) | | Type of object BNAME: |
| | | | A = Alias |
| | | | F = Function instance |
| | | | N = Nickname |
| | | | O = Privilege dependency on all subtables or subviews in a table or view hierarchy |
| | | | I = Index if recording dependency on a base table |
| | | | R = Structured type |
| | | | S = Summary table |
| | | | T = Table |
| | | | U = Typed table |
| | | | V = View |
| | | | W = Typed view |
| BSCHEMA | VARCHAR(128) | | Qualified name of object depended on by the view. |
| BNAME | VARCHAR(128) | | |
| TABAUTH | SMALLINT | Yes | If BTYPE= O, S, T, U, V, W then encodes the privileges on the underlying table or view that this view depends on. Otherwise null. |

## SYSCAT.VIEWS

Contains one or more rows for each view that is created.

*Table 96. SYSCAT.VIEWS Catalog View*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| VIEWSCHEMA | VARCHAR(128) | | Name of the view or the name of a table used to define a summary table. |
| VIEWNAME | VARCHAR(128) | | |
| DEFINER | VARCHAR(128) | | Authorization ID of the creator of the view. |
| SEQNO | SMALLINT | | Always 1. |
| VIEWCHECK | CHAR(1) | | States the type of view checking:<br><br>N = No check option<br><br>L = Local check option<br><br>C = Cascaded check option |
| READONLY | CHAR(1) | | Y = View is read-only because of its definition.<br><br>N = View is not read-only. |
| VALID | CHAR(1) | | Y = View or summary table definition is valid.<br><br>X = View or summary table definition is inoperative; must be re-created. |
| QUALIFIER | VARCHAR(128) | | Contains value of the default schema at the time of object definition. |
| FUNC_PATH | VARCHAR(254) | | The SQL path of the view creator at the time the view was defined. When the view is used in data manipulation statements, this path must be used to resolve function calls in the view. SYSIBM for views created before Version 2. |
| TEXT | CLOB(64k) | | Text of the CREATE VIEW statement. |

## SYSCAT.WRAPOPTIONS

Each row contains wrapper specific options.

*Table 97. SYSCAT.WRAPOPTIONS Catalog View*

| Column Name | Data Type | Nullable | Description |
| --- | --- | --- | --- |
| WRAPNAME | VARCHAR(128) | | Wrapper name. |
| OPTION | VARCHAR(128) | | Name of wrapper option. |
| SETTING | VARCHAR(255) | | Value. |

## SYSCAT.WRAPPERS

Each row contains information on the registered wrapper.

*Table 98. SYSCAT.WRAPPERS Catalog View*

| Column Name | Data Type | Nullable | Description |
|---|---|---|---|
| WRAPNAME | VARCHAR(128) | | Wrapper name. |
| WRAPTYPE | CHAR(1) | | N = Non-relational<br>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 associated with this wrapper. |
| REMARKS | VARCHAR(254) | Yes | User supplied comment, or null. |

## SYSSTAT.COLDIST

Each row describes the Nth-most-frequent value or Nth quantile value of some column. Statistics are not recorded for inherited columns of typed tables.

*Table 99. SYSSTAT.COLDIST Catalog View*

| Column Name | Data Type | Nullable | Description | Updatable |
|---|---|---|---|---|
| TABSCHEMA | VARCHAR(128) | | Qualified name of the table to which this entry applies. | |
| TABNAME | VARCHAR(128) | | | |
| COLNAME | VARCHAR(128) | | Name of the column to which this entry applies. | |
| TYPE | CHAR(1) | | Type of statistic collected:<br><br>F = Frequency (most frequent value)<br><br>Q = Quantile value | |
| SEQNO | SMALLINT | | If TYPE = F, then N in this column identifies the Nth most frequent value. If TYPE = Q, then N in this column identifies the Nth quantile value. | |
| COLVALUE | VARCHAR(254) | Yes | The data value, as a character literal or a null value.<br><br>This column can be updated with a valid representation of the value appropriate to the column that the statistic is associated with. If null is the required frequency value, the column should be set to NULL. | Yes |
| VALCOUNT | BIGINT | | If TYPE = F, then VALCOUNT is the number of occurrences of COLVALUE in the column. If TYPE = Q, then VALCOUNT is the number of rows whose value is less than or equal to COLVALUE.<br><br>This column can be only updated with the following values:<br><br>• >= 0 (zero) | Yes |
| DISTCOUNT | BIGINT | | If TYPE = q, this column records the number of distinct values that are less than or equal to COLVALUE (null iv unavailable.) the number of rows whose value is less than or equal to COLVALUE. | Yes |

## SYSSTAT.COLUMNS

Contains one row for each column for which statistics can be updated. Statistics are not recorded for inherited columns of typed tables.

*Table 100. SYSSTAT.COLUMNS Catalog View*

| Column Name | Data Type | Nullable | Description | Updatable |
|---|---|---|---|---|
| TABSCHEMA | VARCHAR(128) | | Qualified name of the table that contains the column. | |
| TABNAME | VARCHAR(128) | | | |
| COLNAME | VARCHAR(128) | | Column name. | |
| COLCARD | BIGINT | | Number of distinct values in the column; −1 if statistics are not gathered; −2 for inherited columns and columns of H-tables. For any column, COLCARD cannot have a value higher than the cardinality of the table containing that column. This column can only be updated with the following values:<br>• −1 or >= 0 (zero) | Yes |
| HIGH2KEY | VARCHAR(33) | Yes | Second highest value of the column. This field is empty if statistics are not gathered and for inherited columns and columns of H-tables. This column can be updated with a valid representation of the value appropriate to the column that the statistic is associated with. LOWKEY2 should not be greater than HIGH2KEY. | Yes |
| LOW2KEY | VARCHAR(33) | Yes | Second lowest value of the column. Empty if statistics not gathered and for inherited columns and columns of H-tables. This column can be updated with a valid representation of the value appropriate to the column that the statistic is associated with. | Yes |

## SYSSTAT.COLUMNS

*Table 100. SYSSTAT.COLUMNS Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description | Updatable |
|---|---|---|---|---|
| AVGCOLLEN | INTEGER | | Average column length. −1 if a long field or LOB, or statistics have not been collected; −2 for inherited columns and columns of H-tables.<br><br>This column can only be updated with the following values:<br>• −1 or >= 0 (zero) | Yes |

## SYSSTAT.FUNCTIONS

Contains a row for each user-defined function (scalar or aggregate). Does not include built-in functions. Statistics are not recorded for inherited columns of typed tables.

*Table 101. SYSSTAT.FUNCTIONS Catalog View*

| Column Name | Data Type | Nullable | Description | Updatable |
|---|---|---|---|---|
| FUNCSCHEMA | VARCHAR(128) | | Qualified function name. | |
| FUNCNAME | VARCHAR(18) | | | |
| SPECIFICNAME | VARCHAR(18) | | Function specific (instance) name. | |
| IOS_PER_INVOC | DOUBLE | | Estimated number of I/Os per invocation; −1 if not known (0 default).<br><br>This column can only be updated with the following values:<br>• −1 or >= 0 (zero) | Yes |
| INSTS_PER_INVOC | DOUBLE | | Estimated number of instructions per invocation; −1 if not known (450 default).<br><br>This column can only be updated with the following values:<br>• −1 or >= 0 (zero) | Yes |
| IOS_PER_ARGBYTE | DOUBLE | | Estimated number of I/O's per input argument byte; −1 if not known (0 default).<br><br>This column can only be updated with the following values:<br>• −1 or >= 0 (zero) | Yes |
| INSTS_PER_ARGBYTE | DOUBLE | | Estimated number of instructions per input argument byte; −1 if not known (0 default).<br><br>This column can only be updated with the following values:<br>• −1 or >= 0 (zero) | Yes |

## SYSSTAT.FUNCTIONS

*Table 101. SYSSTAT.FUNCTIONS Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description | Updatable |
|---|---|---|---|---|
| PERCENT_ARGBYTES | SMALLINT | | Estimated average percent of input argument bytes that the function will actually read; −1 if not known (100 default).<br><br>This column can only be updated with the following values:<br>• −1 or between 100 and 0 (zero) | Yes |
| INITIAL_IOS | DOUBLE | | Estimated number of I/O's performed the first/last time the function is invoked; −1 if not known (0 default).<br><br>This column can only be updated with the following values:<br>• −1 or >= 0 (zero) | Yes |
| INITIAL_INSTS | DOUBLE | | Estimated number of instructions executed the first/last time the function is invoked; −1 if not known (0 default).<br><br>This column can only be updated with the following values:<br>• −1 or >= 0 (zero) | Yes |
| CARDINALITY | BIGINT | | The predicted cardinality of a table function. −1 if not known, or if function is not a table function. | Yes |
| SELECTIVITY | DOUBLE | | Used for user defined predicates. Default = −1. | |

## SYSSTAT.INDEXES

Contains one row for each index that is defined for a table.

*Table 102. SYSSTAT.INDEXES Catalog View*

| Column Name | Data Type | Nullable | Description | Updatable |
|---|---|---|---|---|
| INDSCHEMA | VARCHAR(128) | | Qualified name of the index. | |
| INDNAME | VARCHAR(18) | | | |
| TABSCHEMA | VARCHAR(128) | | Qualifier of the table name. | |
| TABNAME | VARCHAR(128) | | Name of the table or nickname on which the index is defined. | |
| COLNAMES | CLOB(1M) | | List of column names with + or − prefixes. | |
| NLEAF | INTEGER | | Number of leaf pages; −1 if statistics are not gathered.<br><br>This column can only be updated with the following values:<br>• −1 or > 0 (zero) | Yes |
| NLEVELS | SMALLINT | | Number of index levels; −1 if statistics are not gathered.<br><br>This column can only be updated with the following values:<br>• −1 or > 0 (zero) | Yes |
| FIRSTKEYCARD | BIGINT | | Number of distinct first key values; −1 if statistics are not gathered.<br><br>This column can only be updated with the following values:<br>• −1 or >= 0 (zero) | Yes |
| FIRST2KEYCARD | BIGINT | | Number of distinct keys using the first two columns of the index (−1 if no statistics or inapplicable)<br><br>This column can only be updated with the following values:<br>• −1 or >= 0 (zero) | Yes |

## SYSSTAT.INDEXES

*Table 102. SYSSTAT.INDEXES Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description | Updatable |
|---|---|---|---|---|
| FIRST3KEYCARD | BIGINT | | Number of distinct keys using the first three columns of the index (−1 if no statistics or inapplicable)<br><br>This column can only be updated with the following values:<br>• −1 or >= 0 (zero) | Yes |
| FIRST4KEYCARD | BIGINT | | Number of distinct keys using the first four columns of the index (−1 if no statistics or inapplicable)<br><br>This column can only be updated with the following values:<br>• −1 or >= 0 (zero) | Yes |
| FULLKEYCARD | BIGINT | | Number of distinct full key values; −1 if statistics are not gathered.<br><br>This column can only be updated with the following values:<br>• −1 or >= 0 (zero) | Yes |
| CLUSTERRATIO | SMALLINT | | This is used by the optimizer. It indicates the degree of data clustering with the index; −1 if statistics are not gathered or if detailed index statistics have been gathered.<br><br>This column can only be updated with the following values:<br>• −1 or between 0 and 100 | Yes |
| CLUSTERFACTOR | DOUBLE | | This is used by the optimizer. It is a finer measurement of degree of clustering, or −1 if detailed index statistics have not been gathered.<br><br>This column can only be updated with the following values:<br>• −1 or between 0 and 1 | Yes |

*Table 102. SYSSTAT.INDEXES Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description | Updatable |
|---|---|---|---|---|
| SEQUENTIAL_PAGES | INTEGER | | 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.)<br><br>This column can only be updated with the following values:<br>• −1 or >= 0 (zero) | Yes |
| 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 no statistics are available.)<br><br>This column can only be updated with the following values:<br>• −1 or between 0 and 100 | Yes |
| PAGE_FETCH_PAIRS | VARCHAR(254) | | 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 index using that hypothetical buffer. (Zero-length string if no data available.)<br><br>This column can be updated with the following input values:<br>• The pair delimiter and pair separator characters are the only non-numeric characters accepted<br>• Blanks are the only characters recognized as a pair delimiter and pair separator<br>• Each number entry must have an accompanying partner number entry with the two being separated by the pair separator character<br>• Each pair must be separated from any other pairs by the pair delimiter character<br>• Each expected number entry must between 0-9 (only positive values) | Yes |

## SYSSTAT.TABLES

Contains one row for each *base* table. Views or aliases are, therefore, not included. For typed tables, only the root table of a table hierarchy is included in this view. Statistics are not recorded for inherited columns of typed tables. The CARD value applies to the root table only while the other statistics apply to the entire table hierarchy.

*Table 103. SYSSTAT.TABLES Catalog View*

| Column Name | Data Type | Nullable | Description | Updatable |
|---|---|---|---|---|
| TABSCHEMA | VARCHAR(128) | | Qualified name of the table. | |
| TABNAME | VARCHAR(128) | | | |
| CARD | BIGINT | | Total number of rows in the table; −1 if statistics are not gathered.<br><br>An update to CARD for a table should not attempt to assign it a value less than the COLCARD value of any of the columns in that table. This column can only be updated with the following values: [101].<br><br>• −1 or >= 0 (zero) | Yes |
| NPAGES | INTEGER | | Total number of pages on which the rows of the table exist; −1 if statistics are not gathered; −2 for subtables and H-tables.<br><br>This column can only be updated with the following values: [101]<br><br>• −1 or >= 0 (zero) | Yes |
| FPAGES | INTEGER | | Total number of pages in the file; −1 if statistics are not gathered; −2 for subtables and H-tables.<br><br>This column can only be updated with the following values: [101]<br><br>• −1 or >= 0 (zero) | Yes |

---

101. A value of −2 can not be changed and a column value can not be directly set to −2.

*Table 103. SYSSTAT.TABLES Catalog View  (continued)*

| Column Name | Data Type | Nullable | Description | Updatable |
|---|---|---|---|---|
| OVERFLOW | INTEGER | | Total number of overflow records in the table; −1 if statistics are not gathered; −2 for subtables and H-tables.<br><br>This column can only be updated with the following values: [101]<br><br>• −1 or >= 0 (zero) | Yes |

**SYSSTAT.TABLES**

# Appendix E. Federated Systems: Server Types; Options; Default Type Mappings; Pass-Through Guidelines

This appendix documents:

- The server types that can be defined in the SQL statements for establishing and using DB2 federated systems
- The options that can be defined in the SQL statements for establishing and using DB2 federated systems
- Default mappings between data types supported by the federated server and data types supported by data sources
- Factors to consider and restrictions to observe when using pass-through

## Server Types

Server types indicate what kind of data source the server will represent. Server types vary by vendor, purpose, and platform. Supported values will be available in future documentation.

## SQL Options for Federated Systems

This section documents:

- The column options that can be specified in the ALTER NICKNAME statement.
- The function mapping options that can be specified in the CREATE FUNCTION MAPPING statement
- The server options that can be specified in the CREATE SERVER, ALTER SERVER, and SET SERVER OPTION statements
- The user options that can be specified in the CREATE USER MAPPING and ALTER USER MAPPING statements

### Column Options

The primary purpose of column options is to provide information about nickname columns to the SQL compiler. Setting column options for one or more columns to 'Y' allows the compiler to consider additional push-down possibilities for predicates that perform evaluation operations. See *Administration Guide, Performance* for more information on push-down processing.

*Table 104. Column Options and Their Settings*

| Option | Valid Settings | | Default Setting |
|---|---|---|---|
| numeric_string | 'Y' | Yes, this column contains only strings of numeric data. IMPORTANT: If this column contains only numeric strings followed by trailing blanks, it is inadvisable to specify 'Y'. | 'N' |
| | 'N' | No, this column is not limited to strings of numeric data. | |
| | By setting numeric_string to 'Y' for a column, you are informing the optimizer that this column contains no blanks that could interfere with sorting of the column's data. This option is helpful when the collating sequence of a data source is different from DB2. Columns marked with this option will not be excluded from local (data source) evaluation because of a different collating sequence. | | |
| varchar_no_trailing_blanks | Indicates whether trailing blanks are absent from a specific VARCHAR column: | | 'N' |
| | 'Y' | Yes, trailing blanks are absent from this VARCHAR column. | |
| | 'N' | No, trailing blanks are not absent from this VARCHAR column. | |
| | If data source VARCHAR columns contain no padded blanks, then the optimizer's strategy for accessing them depends in part on whether they contain trailing blanks. By default, the optimizer "assumes" that they actually do contain trailing blanks. On this assumption, it develops an access strategy that involves modifying queries so that the values returned from these columns are the ones that the user expects. If, however, a VARCHAR column has no trailing blanks, and you let the optimizer know this, it can develop a more efficient access strategy. To tell the optimizer that a specific column has no trailing blanks, specify that column in the ALTER NICKNAME statement (for syntax, see the *SQL Reference*). | | |

## Function Mapping Options

The primary purpose of function mapping options is to provide information about the potential cost of executing a data source function at the data source. If pushdown analysis determines that either of two functions within a mapping can be called, the statistical information provided in the mapping definition helps the optimizer to compare the estimated cost of executing the data source function with the estimated cost of executing the DB2 function.

*Table 105. Function Mapping Options and Their Settings*

| Option | Valid Settings | Default Setting |
|---|---|---|
| disable | Disable a default function mapping. Valid values are 'Y' and 'N'. | 'N' |
| initial_insts | Estimated number of instructions processed the first and last time that the data source function is invoked. | '0' |
| initial_ios | Estimated number of I/Os performed the first and last time that the data source function is invoked. | '0' |
| ios_per_argbyte | Estimated number of I/Os expended for each byte of the argument set that's passed to the data source function. | '0' |
| ios_per_invoc | Estimated number of I/Os per invocation of a data source function. | '0' |
| insts_per_argbyte | Estimated number of instructions processed for each byte of the argument set that's passed to the data source function. | '0' |
| insts_per_invoc | Estimated number of instructions processed per invocation of the data source function. | '450' |
| percent_argbytes | Estimated average percent of input argument bytes that the data source function will actually read. | '100' |
| remote_name | Name of the data source function. | local name |

## Server Options

Server options are used to describe a server. In addition to location information (such as the data source machine name), options can specify security and performance attributes for a data source. The security options provide control over password communication (sent or not sent to data sources) and authentication information case (uppercase and/or lowercase IDs and passwords). The performance options help the optimizer determine if evaluation operations can be done at data sources and the best cost model for completing queries that retrieve data from data sources.

*Table 106. Server Options and Their Settings*

| Option | Valid Settings | Default Setting |
|---|---|---|
| collating_sequence | Specifies whether the data source uses the same default collating sequence as the federated database, based on the code set and the country information. If a data source has a collating sequence that differs from DB2's collating sequence, most operations depending on DB2's collating sequence cannot be remotely evaluated at a data source. An example is executing MAX column functions against a nickname character column at a data source with a different collating sequence. Because results might differ if the MAX function is evaluated at the remote data source, DB2 will perform the aggregate operation and the MAX function locally. | 'N' |
| | If your query contains an equal sign, it is possible to push-down that portion of the query even if the collating sequences are different (set to 'N'). For example, the predicate C1 = 'A' could be pushed-down to a data source. Of course, such queries cannot be pushed-down when the collating sequence at the data source is case-insensitive. When a data source is case-insensitive, the results from C1= 'A' and C1 = 'a' are the same, which is not acceptable in a case-sensitive environment (DB2). | |
| | Administrators can create federated databases with a particular collating sequence that matches the data source collating sequence. This approach may speed performance if all data sources use the same collating sequence or if most or all column functions are directed against data sources that use the same collating sequence. | |
| | 'Y'  Data source's collating sequence is the same as federated database's. | |
| | 'N'  Data source's collating sequence is not the same as federated database's. | |
| | 'I'  Data source's collating sequence is different from federated database's and is case-insensitive (for example, 'TOLLESON' and 'TolLESon' are considered equal). | |
| comm_rate | Specifies the communication rate between a federated server and its associated data sources. Expressed in megabytes per second. | '2.0' |

*Table 106. Server Options and Their Settings (continued)*

| Option | Valid Settings | Default Setting |
|---|---|---|
| connectstring | Specifies initialization properties needed to connect to an OLE DB provider. For the complete syntax and semantics of the connection string, see the "Data Link API of the OLE DB Core Components" in the *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK, Microsoft Press, 1998*. | None |
| cpu_ratio | Indicates how much faster or slower a data source's CPU runs than the federated server's CPU. | '1.0' |
| dbname | Name of the data source database that you want the federated server to access. Required for DB2 family data sources; does not apply to Oracle** data sources. | None. |
| fold_id (See notes 1 and 4 at the end of this table.) | Applies to user IDs that the federated server sends to data sources for authentication. Valid values are:<br><br>'U'    The federated server folds the user ID to uppercase before sending it to the data source. This is a logical choice for DB2 Family and Oracle** data sources (See note 2 at end of this table.)<br><br>'N'    The federated server does nothing to the user ID before sending it to the data source. (See note 2 at end of this table.)<br><br>'L'    The federated server folds the user ID to lowercase before sending it to the data source.<br><br>If none of these settings are used, the federated server tries to send the user ID to the data source in uppercase. If the user ID fails, the server tries sending it in lowercase. | None. |
| fold_pw (See notes 1, 3 and 4 at the end of this table.) | Applies to passwords that the federated server sends to data sources for authentication. Valid values are:<br><br>'U'    The federated server folds the password to uppercase before sending it to the data source. This is a logical choice for DB2 Family and Oracle** data sources.<br><br>'N'    The federated server does nothing to the password before sending it to the data source.<br><br>'L'    The federated server folds the password to lowercase before sending it to the data source.<br><br>If none of these settings are used, the federated server tries to send the password to the data source in uppercase. If the password fails, the server tries sending it in lowercase. | None. |
| io_ratio | Denotes how much faster or slower a data source's I/O system runs than the federated server's I/O system. | '1.0' |

*Table 106. Server Options and Their Settings  (continued)*

| Option | Valid Settings | Default Setting |
|---|---|---|
| node | Name by which a data source is defined as an instance to its RDBMS. Required for all data sources. For a DB2 family data source, this name is the node specified in the federated database's DB2 node directory. To view this directory, issue the **db2 list node directory** command. For an Oracle** data source, this name is the server name specified in the Oracle** tnsnames.ora file. To access this name on the Windows NT platform, specify the **View Configuration Information** option of the Oracle** SQL Net Easy Configuration tool. | None. |
| password | Specifies whether passwords are sent to a data source. 'Y'  Passwords are always sent to the data source and validated. This is the default value. 'N'  Passwords are not sent to the data source (regardless of any user mappings) and not validated. 'ENCRYPTION'  Passwords are always sent to the data source in encrypted form and validated. Valid only for DB2 Family data sources that support encrypted passwords. | 'Y' |
| plan_hints | Specifies whether **plan hints** are to be enabled. Plan hints are statement fragments that provide extra information for data source optimizers. This information can, for certain query types, improve query performance. The plan hints can help the data source optimizer decide whether to use an index, which index to use, or which table join sequence to use. 'Y'  Plan hints are to be enabled at the data source if the data source supports plan hints. 'N'  Plan hints are not to be enabled at the data source. | 'N' |
| pushdown | 'Y'  DB2 will consider letting the data source evaluate operations. 'N'  DB2 will retrieve only columns from the remote data source and will not let the data source evaluate other operations, such as joins. | 'Y' |

*Table 106. Server Options and Their Settings  (continued)*

| Option | Valid Settings | Default Setting |
|---|---|---|
| varchar_no_trailing_blanks | Specifies if this data source uses non-blank padded varchar comparison semantics. For varying-length character strings that contain no trailing blanks, some DBMS' s non-blank-padded comparison semantics return the same results as DB2's comparison semantics. If you are certain that all VARCHAR table/view columns at a data source contain no trailing blanks, consider setting this server option to 'Y' for a data source. This option is often used with Oracle** data sources. Ensure that you consider all objects that can potentially have nicknames (including views).<br><br>'Y'     This data source has non-blank-padded comparison semantics similar to DB2's.<br><br>'N'     This data source does not have the same non-blank-padded comparison semantics as DB2's. | 'N' |

Notes on Table 106 on page 1050:

1. This field is applied regardless of the value specified for authentication.

2. Because DB2 stores user IDs in uppercase, the values 'N' and 'U' are logically equivalent to each other.

3. The setting for fold_pw has no effect when the setting for password is 'N'. Because no password is sent, case cannot be a factor.

4. Avoid null settings for either of these options. A null setting may seem attractive because DB2 will make multiple attempts to resolve user IDs and passwords; however, performance might suffer (it is possible that DB2 will send a user ID and password four times before successfully passing data source authentication).

## User Options

User options provide authorization and accounting string information for user mappings. Use them to specify the ID and password used to represent a DB2 authentication ID when authenticating at a data source.

*Table 107. User Options and Their Settings*

| Option | Valid Settings | Default Setting |
|---|---|---|
| remote_authid | Indicates the authorization ID used at the data source. Valid settings include any string of length 255 or less. If this option is not specified, the ID used to connect to database is used. | None. |

*Table 107. User Options and Their Settings (continued)*

| Option | Valid Settings | Default Setting |
|---|---|---|
| remote_password | Indicates the authorization password used at the data source. Valid settings include any string of length 255 or less. If this option is not specified, the password used to connect to the database is used. | None. |
| accounting_string | Used to specify a DRDA accounting string. Valid settings include any string of length 255 or less. This option is required only if accounting information needs to be passed. See the *DB2 Connect User's Guide* | None. |

## Default Data Type Mappings

This section shows default mappings between DB2 data types supported by the federated server and data type supported by the following data sources:

- DB2 Universal Database for OS/390 and DB2 for MVS/ESA
- DB2 Universal Database for AS/400 and DB2 for OS/400
- Oracle
- DB2 for VM and VSE; SQL/DS

The mappings shown are between non-identical data types. Mappings between identical data types are not shown.

### Default Type Mappings between DB2 and DB2 Universal Database for OS/390 (and DB2 for MVS/ESA) Data Sources

*Table 108. Default Type Mappings between DB2 and DB2 Universal Database for OS/390 (and DB2 for MVS/ESA) Data Sources*

| DB2 for MVS, DB2 for OS/390 | DB2 |
|---|---|
| varchar(n), n <= 32672 | varchar(n) |
| vargraphic(n), n <= 16336 | vargraphic(n) |
| char(255) | varchar(255) |
| char(255) for bit data | varchar(255) for bit data |

### Default Type Mappings between DB2 and 2 Universal Database for AS/400 (and DB2 for OS/400) Data Sources

*Table 109. Default Type Mappings between DB2 and DB2 Universal Database for AS/400 (and DB2 for OS/400) Data Sources*

| DB2 for OS/400, DB2 for AS/400 | DB2 |
|---|---|
| char(n), n <= 254 | char(n) |

*Table 109. Default Type Mappings between DB2 and DB2 Universal Database for AS/400 (and DB2 for OS/400) Data Sources (continued)*

| DB2 for OS/400, DB2 for AS/400 | DB2 |
|---|---|
| char(n), n between 255 and 32672 | varchar(n) |
| varchar(n), n <= 32672 | varchar(n) |
| graphic(n), n <= 127 | graphic(n) |
| graphic(n), n between 127 and 16336 | vargraphic(n) |
| vargraphic(n), n <= 16336 | vargraphic(n) |

## Default Type Mappings between DB2 and Oracle Data Sources

*Table 110. Default Type Mappings between DB2 and Oracle Data Sources*

| Oracle | DB2 |
|---|---|
| rowid | char(18) |
| char(n), n <= 254 | char(n) |
| nchar(n), n <= 254 | char(n) |
| char(255) | varchar(255) |
| varchar2(n), n <= 32672 | varchar(n) |
| nvarchar2(n), n <= 32672 | varchar(n) |
| number(p,s), p <= 4 and s = 0 | smallint |
| number(p,s), 4 <= p <= 9 and s = 0 | integer |
| number(p,s), 10 <= p <= 18 and s = 0 | bigint |
| number(p,s), p <= 31 and 0 <= s <= p and previous two cases don't match | decimal |
| number(p,s), all cases other than the previous 4 | double |
| raw(n), n <= 254 | char(n) for bit data |
| raw(255) | varchar(255) for bit data |
| date (char(9)) | timestamp |

## Default Type Mappings between DB2 and DB2 for VM and VSE (and SQL/DS) Data Sources

*Table 111. Default Type Mappings between DB2 and DB2 for VM and VSE (and SQL/DS) Data Sources*

| DB2 for OS/390, SQL/DS | DB2 |
|---|---|
| varchar(n), n <= 32672 | varchar(n) |
| vargraphic(n), n <= 16336 | vargraphic(n) |

## Pass-Through Facility Processing

A facility called *pass-through* can be used to query a data source in the SQL that is native to that data source. This section:

- States what kind of SQL statements a federated server and its associated data sources process in pass-through sessions.
- Lists considerations and restrictions to be aware of when using pass-through.

### SQL Processing in Pass-Through Sessions

The following rules specify whether an SQL statement is processed by DB2 or by a data source:

- In a pass-through session, a statement is sent to the data source for processing if the statement is dynamically prepared and executed during the session through either (1) an EXECUTE IMMEDIATE statement or (2) a PREPARE statement and an EXECUTE statement.
- If a static statement is submitted in a pass-through session, it is sent to the federated server for processing.

### Considerations and Restrictions

There are a number of considerations and restrictions that apply to pass-through. Some of them are of a general nature; others concern Oracle data sources only.

#### Using Pass-Through with All Data Sources

The following information applies to all data sources:

- If a PREPARE statement is used to dynamically prepare a statement in a pass-through session, the latter statement cannot be processed unless it is referenced in an EXECUTE statement in the same session.
- If a COMMIT or ROLLBACK command is issued during a pass-through session, this command will complete the current unit of work (UOW).
- Users and applications can use pass-through to write to data sources; for example, to insert, update, and delete table rows. Note that a cursor cannot be opened directly against a data source object in a pass-through session (SQLSTATE 25000).
- An application can have several SET PASSTHRU statements in effect at the same time to different data sources. Although the application might have issued multiple SET PASSTHRU statements, the pass-through sessions are not truly nested. The federated server will not pass through one data source to access another. Rather, the server accesses each data source directly.

- If multiple pass-through sessions are open at the same time, each unit of work within each session must be concluded with a COMMIT or ROLLBACK statement. The sessions can then be ended in one operation with the SET PASSTHRU statement and its RESET option.
- It is not possible pass through to more than one data source at a time.
- Pass-through does not support stored procedure calls.

**Using Pass-Through with Oracle Data Sources**

The following information applies to Oracle data sources:

- The following restriction applies when a remote client issues a SELECT statement from a command line processor (CLP) in pass-through mode: If the client code is a DB2 SDK prior to DB2 Universal Database Version 5, the SELECT will elicit SQLSTATE 25000. To avoid this error, remote clients must use a DB2 SDK that is at Version 5 or greater.
- Any DDL statement issued against an Oracle server is performed at parse time and is not subject to transaction semantics. The operation, when complete, is automatically committed by Oracle. If a rollback occurs, the DDL is not rolled back.
- When a SELECT statement is issued from raw data types, the RAWTOHEX function should be invoked to receive the hexadecimal values. When an INSERT into raw data types is performed, the hexadecimal representation should be provided.

# Appendix F. Sample Tables

This appendix shows the information contained in the sample tables, and how to install and remove them. The sample tables are used in the examples that appear in this manual and other manuals in this library. In addition, the data contained in the sample files with BLOB and CLOB data types is shown.

The following sections are included in this appendix:.

**Sample Tables**

In the sample tables, a dash (-) indicates a null value.

## The Sample Database

The examples in this book use a sample database. To use these examples, you must install the SAMPLE database. To use it, the database manager must be installed.

### To Install the Sample Database

An executable file installs the sample database.[102] To install a database you must have SYSADM authority.

- **When Using UNIX-based platforms**

  If you are using the operating system command prompt, type:

  ```
  sqllib/misc/db2sampl <path>
  ```

  from the home directory of the database manager instance owner, where *path* is an optional parameter specifying the path where the sample database is to be created. Press Enter.[103] The schema for DB2SAMPL is the CURRENT SCHEMA special register value.

- **When using OS/2 or Windows platforms**

  If you are using the operating system command prompt, type:

  ```
  db2sampl e
  ```

  where *e* is an optional parameter specifying the drive where the database is to be created. Press Enter.[104]

  If you are not logged on to your workstation through User Profile Management, you will be prompted to do so.

### To Erase the Sample Database

If you do not need to access the sample database, you can erase it by using the DROP DATABASE command:

```
db2 drop database sample
```

---

102. For information related to this command, see the DB2SAMPL command in the *Command Reference*.

103. If the path parameter is not specified, the sample tables are installed in the default path specified by the DFTDBPATH parameter in the database manager configuration file.

104. If the drive parameter is not specified, the sample tables are installed on the same drive as DB2.

## CL_SCHED Table

| Name: | CLASS_CODE | DAY | STARTING | ENDING |
|---|---|---|---|---|
| Type: | char(7) | smallint | time | time |
| Desc: | Class Code (room:teacher) | Day # of 4 day schedule | Class Start Time | Class End Time |

## DEPARTMENT Table

| Name: | DEPTNO | DEPTNAME | MGRNO | ADMRDEPT | LOCATION |
|---|---|---|---|---|---|
| Type: | char(3) not null | varchar(29) not null | char(6) | char(3) not null | char(16) |
| Desc: | Department number | Name describing general activities of department | Employee number (EMPNO) of department manager | Department (DEPTNO) to which this department reports | Name of the remote location |
| Values: | A00 | SPIFFY COMPUTER SERVICE DIV. | 000010 | A00 | - |
| | B01 | PLANNING | 000020 | A00 | - |
| | C01 | INFORMATION CENTER | 000030 | A00 | - |
| | D01 | DEVELOPMENT CENTER | - | A00 | - |
| | D11 | MANUFACTURING SYSTEMS | 000060 | D01 | - |
| | D21 | ADMINISTRATION SYSTEMS | 000070 | D01 | - |
| | E01 | SUPPORT SERVICES | 000050 | A00 | - |
| | E11 | OPERATIONS | 000090 | E01 | - |
| | E21 | SOFTWARE SUPPORT | 000100 | E01 | - |

## EMPLOYEE Table

| Names: | EMPNO | FIRSTNME | MIDINIT | LASTNAME | WORKDEPT | PHONENO | HIREDATE |
|---|---|---|---|---|---|---|---|
| Type: | char(6) not null | varchar(12) not null | char(1) not null | varchar(15) not null | char(3) | char(4) | date |
| Desc: | Employee number | First name | Middle initial | Last name | Department (DEPTNO) in which the employee works | Phone number | Date of hire |

| JOB | EDLEVEL | SEX | BIRTHDATE | SALARY | BONUS | COMM |
|---|---|---|---|---|---|---|
| char(8) | smallint not null | char(1) | date | dec(9,2) | dec(9,2) | dec(9,2) |
| Job | Number of years of formal education | Sex (M male, F female) | Date of birth | Yearly salary | Yearly bonus | Yearly commission |

See the following page for the values in the EMPLOYEE table.

# Sample Tables

| EMPNO | FIRSTNME | MIDINIT | LASTNAME | WORKDEPT | PHONENO | HIREDATE | JOB | EDLEVEL | SEX | BIRTHDATE | SALARY | BONUS | COMM |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| char(6) not null | varchar(12) not null | char(1) not null | varchar(15) not null | char(3) | char(4) | date | char(8) | smallint not null | char(1) | date | dec(9,2) | dec(9,2) | dec(9,2) |
| 000010 | CHRISTINE | I | HAAS | A00 | 3978 | 1965-01-01 | PRES | 18 | F | 1933-08-24 | 52750 | 1000 | 4220 |
| 000020 | MICHAEL | L | THOMPSON | B01 | 3476 | 1973-10-10 | MANAGER | 18 | M | 1948-02-02 | 41250 | 800 | 3300 |
| 000030 | SALLY | A | KWAN | C01 | 4738 | 1975-04-05 | MANAGER | 20 | F | 1941-05-11 | 38250 | 800 | 3060 |
| 000050 | JOHN | B | GEYER | E01 | 6789 | 1949-08-17 | MANAGER | 16 | M | 1925-09-15 | 40175 | 800 | 3214 |
| 000060 | IRVING | F | STERN | D11 | 6423 | 1973-09-14 | MANAGER | 16 | M | 1945-07-07 | 32250 | 500 | 2580 |
| 000070 | EVA | D | PULASKI | D21 | 7831 | 1980-09-30 | MANAGER | 16 | F | 1953-05-26 | 36170 | 700 | 2893 |
| 000090 | EILEEN | W | HENDERSON | E11 | 5498 | 1970-08-15 | MANAGER | 16 | F | 1941-05-15 | 29750 | 600 | 2380 |
| 000100 | THEODORE | Q | SPENSER | E21 | 0972 | 1980-06-19 | MANAGER | 14 | M | 1956-12-18 | 26150 | 500 | 2092 |
| 000110 | VINCENZO | G | LUCCHESSI | A00 | 3490 | 1958-05-16 | SALESREP | 19 | M | 1929-11-05 | 46500 | 900 | 3720 |
| 000120 | SEAN | | O'CONNELL | A00 | 2167 | 1963-12-05 | CLERK | 14 | M | 1942-10-18 | 29250 | 600 | 2340 |
| 000130 | DOLORES | M | QUINTANA | C01 | 4578 | 1971-07-28 | ANALYST | 16 | F | 1925-09-15 | 23800 | 500 | 1904 |
| 000140 | HEATHER | A | NICHOLLS | C01 | 1793 | 1976-12-15 | ANALYST | 18 | F | 1946-01-19 | 28420 | 600 | 2274 |
| 000150 | BRUCE | | ADAMSON | D11 | 4510 | 1972-02-12 | DESIGNER | 16 | M | 1947-05-17 | 25280 | 500 | 2022 |
| 000160 | ELIZABETH | R | PIANKA | D11 | 3782 | 1977-10-11 | DESIGNER | 17 | F | 1955-04-12 | 22250 | 400 | 1780 |
| 000170 | MASATOSHI | J | YOSHIMURA | D11 | 2890 | 1978-09-15 | DESIGNER | 16 | M | 1951-01-05 | 24680 | 500 | 1974 |
| 000180 | MARILYN | S | SCOUTTEN | D11 | 1682 | 1973-07-07 | DESIGNER | 17 | F | 1949-02-21 | 21340 | 500 | 1707 |
| 000190 | JAMES | H | WALKER | D11 | 2986 | 1974-07-26 | DESIGNER | 16 | M | 1952-06-25 | 20450 | 400 | 1636 |
| 000200 | DAVID | | BROWN | D11 | 4501 | 1966-03-03 | DESIGNER | 16 | M | 1941-05-29 | 27740 | 600 | 2217 |
| 000210 | WILLIAM | T | JONES | D11 | 0942 | 1979-04-11 | DESIGNER | 17 | M | 1953-02-23 | 18270 | 400 | 1462 |
| 000220 | JENNIFER | K | LUTZ | D11 | 0672 | 1968-08-29 | DESIGNER | 18 | F | 1948-03-19 | 29840 | 600 | 2387 |
| 000230 | JAMES | J | JEFFERSON | D21 | 2094 | 1966-11-21 | CLERK | 14 | M | 1935-05-30 | 22180 | 400 | 1774 |
| 000240 | SALVATORE | M | MARINO | D21 | 3780 | 1979-12-05 | CLERK | 17 | M | 1954-03-31 | 28760 | 600 | 2301 |
| 000250 | DANIEL | S | SMITH | D21 | 0961 | 1969-10-30 | CLERK | 15 | M | 1939-11-12 | 19180 | 400 | 1534 |
| 000260 | SYBIL | P | JOHNSON | D21 | 8953 | 1975-09-11 | CLERK | 16 | F | 1936-10-05 | 17250 | 300 | 1380 |
| 000270 | MARIA | L | PEREZ | D21 | 9001 | 1980-09-30 | CLERK | 15 | F | 1953-05-26 | 27380 | 500 | 2190 |
| 000280 | ETHEL | R | SCHNEIDER | E11 | 8997 | 1967-03-24 | OPERATOR | 17 | F | 1936-03-28 | 26250 | 500 | 2100 |
| 000290 | JOHN | R | PARKER | E11 | 4502 | 1980-05-30 | OPERATOR | 12 | M | 1946-07-09 | 15340 | 300 | 1227 |
| 000300 | PHILIP | X | SMITH | E11 | 2095 | 1972-06-19 | OPERATOR | 14 | M | 1936-10-27 | 17750 | 400 | 1420 |
| 000310 | MAUDE | F | SETRIGHT | E11 | 3332 | 1964-09-12 | OPERATOR | 12 | F | 1931-04-21 | 15900 | 300 | 1272 |
| 000320 | RAMLAL | V | MEHTA | E21 | 9990 | 1965-07-07 | FIELDREP | 16 | M | 1932-08-11 | 19950 | 400 | 1596 |

| EMPNO | FIRSTNME | MID INIT | LASTNAME | WORK DEPT | PHONE NO | HIREDATE | JOB | ED LEVEL | SEX | BIRTHDATE | SALARY | BONUS | COMM |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 000330 | WING | | LEE | E21 | 2103 | 1976-02-23 | FIELDREP | 14 | M | 1941-07-18 | 25370 | 500 | 2030 |
| 000340 | JASON | R | GOUNOT | E21 | 5698 | 1947-05-05 | FIELDREP | 16 | M | 1926-05-17 | 23840 | 500 | 1907 |

## EMP_ACT Table

| Name: | EMPNO | PROJNO | ACTNO | EMPTIME | EMSTDATE | EMENDATE |
|---|---|---|---|---|---|---|
| Type: | char(6) not null | char(6) not null | smallint not null | dec(5,2) | date | date |
| Desc: | Employee number | Project number | Activity number | Proportion of employee's time spent on project | Date activity starts | Date activity ends |
| Values: | 000010 | AD3100 | 10 | .50 | 1982-01-01 | 1982-07-01 |
| | 000070 | AD3110 | 10 | 1.00 | 1982-01-01 | 1983-02-01 |
| | 000230 | AD3111 | 60 | 1.00 | 1982-01-01 | 1982-03-15 |
| | 000230 | AD3111 | 60 | .50 | 1982-03-15 | 1982-04-15 |
| | 000230 | AD3111 | 70 | .50 | 1982-03-15 | 1982-10-15 |
| | 000230 | AD3111 | 80 | .50 | 1982-04-15 | 1982-10-15 |
| | 000230 | AD3111 | 180 | 1.00 | 1982-10-15 | 1983-01-01 |
| | 000240 | AD3111 | 70 | 1.00 | 1982-02-15 | 1982-09-15 |
| | 000240 | AD3111 | 80 | 1.00 | 1982-09-15 | 1983-01-01 |
| | 000250 | AD3112 | 60 | 1.00 | 1982-01-01 | 1982-02-01 |
| | 000250 | AD3112 | 60 | .50 | 1982-02-01 | 1982-03-15 |
| | 000250 | AD3112 | 60 | .50 | 1982-12-01 | 1983-01-01 |
| | 000250 | AD3112 | 60 | 1.00 | 1983-01-01 | 1983-02-01 |
| | 000250 | AD3112 | 70 | .50 | 1982-02-01 | 1982-03-15 |
| | 000250 | AD3112 | 70 | 1.00 | 1982-03-15 | 1982-08-15 |
| | 000250 | AD3112 | 70 | .25 | 1982-08-15 | 1982-10-15 |
| | 000250 | AD3112 | 80 | .25 | 1982-08-15 | 1982-10-15 |
| | 000250 | AD3112 | 80 | .50 | 1982-10-15 | 1982-12-01 |

## Sample Tables

| Name: | EMPNO | PROJNO | ACTNO | EMPTIME | EMSTDATE | EMENDATE |
|---|---|---|---|---|---|---|
| | 000250 | AD3112 | 180 | .50 | 1982-08-15 | 1983-01-01 |
| | 000260 | AD3113 | 70 | .50 | 1982-06-15 | 1982-07-01 |
| | 000260 | AD3113 | 70 | 1.00 | 1982-07-01 | 1983-02-01 |
| | 000260 | AD3113 | 80 | 1.00 | 1982-01-01 | 1982-03-01 |
| | 000260 | AD3113 | 80 | .50 | 1982-03-01 | 1982-04-15 |
| | 000260 | AD3113 | 180 | .50 | 1982-03-01 | 1982-04-15 |
| | 000260 | AD3113 | 180 | 1.00 | 1982-04-15 | 1982-06-01 |
| | 000260 | AD3113 | 180 | .50 | 1982-06-01 | 1982-07-01 |
| | 000270 | AD3113 | 60 | .50 | 1982-03-01 | 1982-04-01 |
| | 000270 | AD3113 | 60 | 1.00 | 1982-04-01 | 1982-09-01 |
| | 000270 | AD3113 | 60 | .25 | 1982-09-01 | 1982-10-15 |
| | 000270 | AD3113 | 70 | .75 | 1982-09-01 | 1982-10-15 |
| | 000270 | AD3113 | 70 | 1.00 | 1982-10-15 | 1983-02-01 |
| | 000270 | AD3113 | 80 | 1.00 | 1982-01-01 | 1982-03-01 |
| | 000270 | AD3113 | 80 | .50 | 1982-03-01 | 1982-04-01 |
| | 000030 | IF1000 | 10 | .50 | 1982-06-01 | 1983-01-01 |
| | 000130 | IF1000 | 90 | 1.00 | 1982-01-01 | 1982-10-01 |
| | 000130 | IF1000 | 100 | .50 | 1982-10-01 | 1983-01-01 |
| | 000140 | IF1000 | 90 | .50 | 1982-10-01 | 1983-01-01 |
| | 000030 | IF2000 | 10 | .50 | 1982-01-01 | 1983-01-01 |
| | 000140 | IF2000 | 100 | 1.00 | 1982-01-01 | 1982-03-01 |
| | 000140 | IF2000 | 100 | .50 | 1982-03-01 | 1982-07-01 |
| | 000140 | IF2000 | 110 | .50 | 1982-03-01 | 1982-07-01 |
| | 000140 | IF2000 | 110 | .50 | 1982-10-01 | 1983-01-01 |
| | 000010 | MA2100 | 10 | .50 | 1982-01-01 | 1982-11-01 |
| | 000110 | MA2100 | 20 | 1.00 | 1982-01-01 | 1982-03-01 |
| | 000010 | MA2110 | 10 | 1.00 | 1982-01-01 | 1983-02-01 |
| | 000200 | MA2111 | 50 | 1.00 | 1982-01-01 | 1982-06-15 |
| | 000200 | MA2111 | 60 | 1.00 | 1982-06-15 | 1983-02-01 |
| | 000220 | MA2111 | 40 | 1.00 | 1982-01-01 | 1983-02-01 |
| | 000150 | MA2112 | 60 | 1.00 | 1982-01-01 | 1982-07-15 |
| | 000150 | MA2112 | 180 | 1.00 | 1982-07-15 | 1983-02-01 |
| | 000170 | MA2112 | 60 | 1.00 | 1982-01-01 | 1983-06-01 |
| | 000170 | MA2112 | 70 | 1.00 | 1982-06-01 | 1983-02-01 |
| | 000190 | MA2112 | 70 | 1.00 | 1982-02-01 | 1982-10-01 |
| | 000190 | MA2112 | 80 | 1.00 | 1982-10-01 | 1983-10-01 |
| | 000160 | MA2113 | 60 | 1.00 | 1982-07-15 | 1983-02-01 |

| Name: | EMPNO | PROJNO | ACTNO | EMPTIME | EMSTDATE | EMENDATE |
|---|---|---|---|---|---|---|
| | 000170 | MA2113 | 80 | 1.00 | 1982-01-01 | 1983-02-01 |
| | 000180 | MA2113 | 70 | 1.00 | 1982-04-01 | 1982-06-15 |
| | 000210 | MA2113 | 80 | .50 | 1982-10-01 | 1983-02-01 |
| | 000210 | MA2113 | 180 | .50 | 1982-10-01 | 1983-02-01 |
| | 000050 | OP1000 | 10 | .25 | 1982-01-01 | 1983-02-01 |
| | 000090 | OP1010 | 10 | 1.00 | 1982-01-01 | 1983-02-01 |
| | 000280 | OP1010 | 130 | 1.00 | 1982-01-01 | 1983-02-01 |
| | 000290 | OP1010 | 130 | 1.00 | 1982-01-01 | 1983-02-01 |
| | 000300 | OP1010 | 130 | 1.00 | 1982-01-01 | 1983-02-01 |
| | 000310 | OP1010 | 130 | 1.00 | 1982-01-01 | 1983-02-01 |
| | 000050 | OP2010 | 10 | .75 | 1982-01-01 | 1983-02-01 |
| | 000100 | OP2010 | 10 | 1.00 | 1982-01-01 | 1983-02-01 |
| | 000320 | OP2011 | 140 | .75 | 1982-01-01 | 1983-02-01 |
| | 000320 | OP2011 | 150 | .25 | 1982-01-01 | 1983-02-01 |
| | 000330 | OP2012 | 140 | .25 | 1982-01-01 | 1983-02-01 |
| | 000330 | OP2012 | 160 | .75 | 1982-01-01 | 1983-02-01 |
| | 000340 | OP2013 | 140 | .50 | 1982-01-01 | 1983-02-01 |
| | 000340 | OP2013 | 170 | .50 | 1982-01-01 | 1983-02-01 |
| | 000020 | PL2100 | 30 | 1.00 | 1982-01-01 | 1982-09-15 |

## EMP_PHOTO Table

| Name: | EMPNO | PHOTO_FORMAT | PICTURE |
|---|---|---|---|
| Type: | char(6) not null | varchar(10) not null | blob(100k) |
| Desc: | Employee number | Photo format | Photo of employee |
| Values: | 000130 | bitmap | db200130.bmp |
| | 000130 | gif | db200130.gif |
| | 000130 | xwd | db200130.xwd |
| | 000140 | bitmap | db200140.bmp |
| | 000140 | gif | db200140.gif |
| | 000140 | xwd | db200140.xwd |
| | 000150 | bitmap | db200150.bmp |
| | 000150 | gif | db200150.gif |
| | 000150 | xwd | db200150.xwd |
| | 000190 | bitmap | db200190.bmp |
| | 000190 | gif | db200190.gif |
| | 000190 | xwd | db200190.xwd |

## Sample Tables

- "Quintana Photo" on page 1071 shows the picture of the employee, Delores Quintana.
- "Nicholls Photo" on page 1073 shows the picture of the employee, Heather Nicholls.
- "Adamson Photo" on page 1074 shows the picture of the employee, Bruce Adamson.
- "Walker Photo" on page 1076 shows the picture of the employee, James Walker.

### EMP_RESUME Table

| Name: | EMPNO | RESUME_FORMAT | RESUME |
|---|---|---|---|
| Type: | char(6) not null | varchar(10) not null | clob(5k) |
| Desc: | Employee number | Resume Format | Resume of employee |
| Values: | 000130 | ascii | db200130.asc |
| | 000130 | script | db200130.scr |
| | 000140 | ascii | db200140.asc |
| | 000140 | script | db200140.scr |
| | 000150 | ascii | db200150.asc |
| | 000150 | script | db200150.scr |
| | 000190 | ascii | db200190.asc |
| | 000190 | script | db200190.scr |

- "Quintana Resume" on page 1072 shows the resume of the employee, Delores Quintana.
- "Nicholls Resume" on page 1073 shows the resume of the employee, Heather Nicholls.
- "Adamson Resume" on page 1075 shows the resume of the employee, Bruce Adamson.
- "Walker Resume" on page 1076 shows the resume of the employee, James Walker.

### IN_TRAY Table

| Name: | RECEIVED | SOURCE | SUBJECT | NOTE_TEXT |
|---|---|---|---|---|
| Type: | timestamp | char(8) | char(64) | varchar(3000) |
| Desc: | Date and Time received | User id of person sending note | Brief description | The note |

## ORG Table

| Name: | DEPTNUMB | DEPTNAME | MANAGER | DIVISION | LOCATION |
|---|---|---|---|---|---|
| Type: | smallint not null | varchar(14) | smallint | varchar(10) | varchar(13) |
| Desc: | Department number | Department name | Manager number | Division of corporation | City |
| Values: | 10 | Head Office | 160 | Corporate | New York |
| | 15 | New England | 50 | Eastern | Boston |
| | 20 | Mid Atlantic | 10 | Eastern | Washington |
| | 38 | South Atlantic | 30 | Eastern | Atlanta |
| | 42 | Great Lakes | 100 | Midwest | Chicago |
| | 51 | Plains | 140 | Midwest | Dallas |
| | 66 | Pacific | 270 | Western | San Francisco |
| | 84 | Mountain | 290 | Western | Denver |

## PROJECT Table

| Name: | PROJNO | PROJNAME | DEPTNO | RESPEMP | PRSTAFF | PRSTDATE | PRENDATE | MAJPROJ |
|---|---|---|---|---|---|---|---|---|
| Type: | char(6) not null | varchar(24) not null | char(3) not null | char(6) not null | dec(5,2) | date | date | char(6) |
| Desc: | Project number | Project name | Department responsible | Employee responsible | Estimated mean staffing | Estimated start date | Estimated end date | Major project, for a subproject |
| Values: | AD3100 | ADMIN SERVICES | D01 | 000010 | 6.5 | 1982-01-01 | 1983-02-01 | - |
| | AD3110 | GENERAL ADMIN SYSTEMS | D21 | 000070 | 6 | 1982-01-01 | 1983-02-01 | AD3100 |
| | AD3111 | PAYROLL PROGRAMMING | D21 | 000230 | 2 | 1982-01-01 | 1983-02-01 | AD3110 |
| | AD3112 | PERSONNEL PROGRAMMING | D21 | 000250 | 1 | 1982-01-01 | 1983-02-01 | AD3110 |
| | AD3113 | ACCOUNT PROGRAMMING | D21 | 000270 | 2 | 1982-01-01 | 1983-02-01 | AD3110 |
| | IF1000 | QUERY SERVICES | C01 | 000030 | 2 | 1982-01-01 | 1983-02-01 | - |
| | IF2000 | USER EDUCATION | C01 | 000030 | 1 | 1982-01-01 | 1983-02-01 | - |
| | MA2100 | WELD LINE AUTOMATION | D01 | 000010 | 12 | 1982-01-01 | 1983-02-01 | - |
| | MA2110 | W L PROGRAMMING | D11 | 000060 | 9 | 1982-01-01 | 1983-02-01 | MA2100 |
| | MA2111 | W L PROGRAM DESIGN | D11 | 000220 | 2 | 1982-01-01 | 1982-12-01 | MA2110 |
| | MA2112 | W L ROBOT DESIGN | D11 | 000150 | 3 | 1982-01-01 | 1982-12-01 | MA2110 |

## Sample Tables

| Name: | PROJNO | PROJNAME | DEPTNO | RESPEMP | PRSTAFF | PRSTDATE | PRENDATE | MAJPROJ |
|---|---|---|---|---|---|---|---|---|
| | MA2113 | W L PROD CONT PROGS | D11 | 000160 | 3 | 1982-02-15 | 1982-12-01 | MA2110 |
| | OP1000 | OPERATION SUPPORT | E01 | 000050 | 6 | 1982-01-01 | 1983-02-01 | - |
| | OP1010 | OPERATION | E11 | 000090 | 5 | 1982-01-01 | 1983-02-01 | OP1000 |
| | OP2000 | GEN SYSTEMS SERVICES | E01 | 000050 | 5 | 1982-01-01 | 1983-02-01 | - |
| | OP2010 | SYSTEMS SUPPORT | E21 | 000100 | 4 | 1982-01-01 | 1983-02-01 | OP2000 |
| | OP2011 | SCP SYSTEMS SUPPORT | E21 | 000320 | 1 | 1982-01-01 | 1983-02-01 | OP2010 |
| | OP2012 | APPLICATIONS SUPPORT | E21 | 000330 | 1 | 1982-01-01 | 1983-02-01 | OP2010 |
| | OP2013 | DB/DC SUPPORT | E21 | 000340 | 1 | 1982-01-01 | 1983-02-01 | OP2010 |
| | PL2100 | WELD LINE PLANNING | B01 | 000020 | 1 | 1982-01-01 | 1982-09-15 | MA2100 |

### SALES Table

| Name: | SALES_DATE | SALES_PERSON | REGION | SALES |
|---|---|---|---|---|
| Type: | date | varchar(15) | varchar(15) | int |
| Desc: | Date of sales | Employee's last name | Region of sales | Number of sales |
| Values: | 12/31/1995 | LUCCHESSI | Ontario-South | 1 |
| | 12/31/1995 | LEE | Ontario-South | 3 |
| | 12/31/1995 | LEE | Quebec | 1 |
| | 12/31/1995 | LEE | Manitoba | 2 |
| | 12/31/1995 | GOUNOT | Quebec | 1 |
| | 03/29/1996 | LUCCHESSI | Ontario-South | 3 |
| | 03/29/1996 | LUCCHESSI | Quebec | 1 |
| | 03/29/1996 | LEE | Ontario-South | 2 |
| | 03/29/1996 | LEE | Ontario-North | 2 |
| | 03/29/1996 | LEE | Quebec | 3 |
| | 03/29/1996 | LEE | Manitoba | 5 |
| | 03/29/1996 | GOUNOT | Ontario-South | 3 |
| | 03/29/1996 | GOUNOT | Quebec | 1 |
| | 03/29/1996 | GOUNOT | Manitoba | 7 |
| | 03/30/1996 | LUCCHESSI | Ontario-South | 1 |
| | 03/30/1996 | LUCCHESSI | Quebec | 2 |
| | 03/30/1996 | LUCCHESSI | Manitoba | 1 |
| | 03/30/1996 | LEE | Ontario-South | 7 |
| | 03/30/1996 | LEE | Ontario-North | 3 |
| | 03/30/1996 | LEE | Quebec | 7 |

| Name: | SALES_DATE | SALES_PERSON | REGION | SALES |
|---|---|---|---|---|
| | 03/30/1996 | LEE | Manitoba | 4 |
| | 03/30/1996 | GOUNOT | Ontario-South | 2 |
| | 03/30/1996 | GOUNOT | Quebec | 18 |
| | 03/30/1996 | GOUNOT | Manitoba | 1 |
| | 03/31/1996 | LUCCHESSI | Manitoba | 1 |
| | 03/31/1996 | LEE | Ontario-South | 14 |
| | 03/31/1996 | LEE | Ontario-North | 3 |
| | 03/31/1996 | LEE | Quebec | 7 |
| | 03/31/1996 | LEE | Manitoba | 3 |
| | 03/31/1996 | GOUNOT | Ontario-South | 2 |
| | 03/31/1996 | GOUNOT | Quebec | 1 |
| | 04/01/1996 | LUCCHESSI | Ontario-South | 3 |
| | 04/01/1996 | LUCCHESSI | Manitoba | 1 |
| | 04/01/1996 | LEE | Ontario-South | 8 |
| | 04/01/1996 | LEE | Ontario-North | - |
| | 04/01/1996 | LEE | Quebec | 8 |
| | 04/01/1996 | LEE | Manitoba | 9 |
| | 04/01/1996 | GOUNOT | Ontario-South | 3 |
| | 04/01/1996 | GOUNOT | Ontario-North | 1 |
| | 04/01/1996 | GOUNOT | Quebec | 3 |
| | 04/01/1996 | GOUNOT | Manitoba | 7 |

## STAFF Table

| Name: | ID | NAME | DEPT | JOB | YEARS | SALARY | COMM |
|---|---|---|---|---|---|---|---|
| Type: | smallint not null | varchar(9) | smallint | char(5) | smallint | dec(7,2) | dec(7,2) |
| Desc: | Employee number | Employee name | Department number | Job type | Years of service | Current salary | Commission |
| Values: | 10 | Sanders | 20 | Mgr | 7 | 18357.50 | - |
| | 20 | Pernal | 20 | Sales | 8 | 18171.25 | 612.45 |
| | 30 | Marenghi | 38 | Mgr | 5 | 17506.75 | - |
| | 40 | O'Brien | 38 | Sales | 6 | 18006.00 | 846.55 |
| | 50 | Hanes | 15 | Mgr | 10 | 20659.80 | - |
| | 60 | Quigley | 38 | Sales | - | 16808.30 | 650.25 |
| | 70 | Rothman | 15 | Sales | 7 | 16502.83 | 1152.00 |
| | 80 | James | 20 | Clerk | - | 13504.60 | 128.20 |
| | 90 | Koonitz | 42 | Sales | 6 | 18001.75 | 1386.70 |
| | 100 | Plotz | 42 | Mgr | 7 | 18352.80 | - |
| | 110 | Ngan | 15 | Clerk | 5 | 12508.20 | 206.60 |
| | 120 | Naughton | 38 | Clerk | - | 12954.75 | 180.00 |

## Sample Tables

| Name: | ID | NAME | DEPT | JOB | YEARS | SALARY | COMM |
|---|---|---|---|---|---|---|---|
| | 130 | Yamaguchi | 42 | Clerk | 6 | 10505.90 | 75.60 |
| | 140 | Fraye | 51 | Mgr | 6 | 21150.00 | - |
| | 150 | Williams | 51 | Sales | 6 | 19456.50 | 637.65 |
| | 160 | Molinare | 10 | Mgr | 7 | 22959.20 | - |
| | 170 | Kermisch | 15 | Clerk | 4 | 12258.50 | 110.10 |
| | 180 | Abrahams | 38 | Clerk | 3 | 12009.75 | 236.50 |
| | 190 | Sneider | 20 | Clerk | 8 | 14252.75 | 126.50 |
| | 200 | Scoutten | 42 | Clerk | - | 11508.60 | 84.20 |
| | 210 | Lu | 10 | Mgr | 10 | 20010.00 | - |
| | 220 | Smith | 51 | Sales | 7 | 17654.50 | 992.80 |
| | 230 | Lundquist | 51 | Clerk | 3 | 13369.80 | 189.65 |
| | 240 | Daniels | 10 | Mgr | 5 | 19260.25 | - |
| | 250 | Wheeler | 51 | Clerk | 6 | 14460.00 | 513.30 |
| | 260 | Jones | 10 | Mgr | 12 | 21234.00 | - |
| | 270 | Lea | 66 | Mgr | 9 | 18555.50 | - |
| | 280 | Wilson | 66 | Sales | 9 | 18674.50 | 811.50 |
| | 290 | Quill | 84 | Mgr | 10 | 19818.00 | - |
| | 300 | Davis | 84 | Sales | 5 | 15454.50 | 806.10 |
| | 310 | Graham | 66 | Sales | 13 | 21000.00 | 200.30 |
| | 320 | Gonzales | 66 | Sales | 4 | 16858.20 | 844.00 |
| | 330 | Burke | 66 | Clerk | 1 | 10988.00 | 55.50 |
| | 340 | Edwards | 84 | Sales | 7 | 17844.00 | 1285.00 |
| | 350 | Gafney | 84 | Clerk | 5 | 13030.50 | 188.00 |

### STAFFG Table

**Note:** STAFFG is only created for double-byte code pages.

| Name: | ID | NAME | DEPT | JOB | YEARS | SALARY | COMM |
|---|---|---|---|---|---|---|---|
| Type: | smallint not null | vargraphic(9) | smallint | graphic(5) | smallint | dec(9,0) | dec(9,0) |
| Desc: | Employee number | Employee name | Department number | Job type | Years of service | Current salary | Commission |
| Values: | 10 | Sanders | 20 | Mgr | 7 | 18357.50 | - |
| | 20 | Pernal | 20 | Sales | 8 | 18171.25 | 612.45 |
| | 30 | Marenghi | 38 | Mgr | 5 | 17506.75 | - |
| | 40 | O'Brien | 38 | Sales | 6 | 18006.00 | 846.55 |
| | 50 | Hanes | 15 | Mgr | 10 | 20659.80 | - |
| | 60 | Quigley | 38 | Sales | - | 16808.30 | 650.25 |

| Name: | ID | NAME | DEPT | JOB | YEARS | SALARY | COMM |
|---|---|---|---|---|---|---|---|
| | 70 | Rothman | 15 | Sales | 7 | 16502.83 | 1152.00 |
| | 80 | James | 20 | Clerk | - | 13504.60 | 128.20 |
| | 90 | Koonitz | 42 | Sales | 6 | 18001.75 | 1386.70 |
| | 100 | Plotz | 42 | Mgr | 7 | 18352.80 | - |
| | 110 | Ngan | 15 | Clerk | 5 | 12508.20 | 206.60 |
| | 120 | Naughton | 38 | Clerk | - | 12954.75 | 180.00 |
| | 130 | Yamaguchi | 42 | Clerk | 6 | 10505.90 | 75.60 |
| | 140 | Fraye | 51 | Mgr | 6 | 21150.00 | - |
| | 150 | Williams | 51 | Sales | 6 | 19456.50 | 637.65 |
| | 160 | Molinare | 10 | Mgr | 7 | 22959.20 | - |
| | 170 | Kermisch | 15 | Clerk | 4 | 12258.50 | 110.10 |
| | 180 | Abrahams | 38 | Clerk | 3 | 12009.75 | 236.50 |
| | 190 | Sneider | 20 | Clerk | 8 | 14252.75 | 126.50 |
| | 200 | Scoutten | 42 | Clerk | - | 11508.60 | 84.20 |
| | 210 | Lu | 10 | Mgr | 10 | 20010.00 | - |
| | 220 | Smith | 51 | Sales | 7 | 17654.50 | 992.80 |
| | 230 | Lundquist | 51 | Clerk | 3 | 13369.80 | 189.65 |
| | 240 | Daniels | 10 | Mgr | 5 | 19260.25 | - |
| | 250 | Wheeler | 51 | Clerk | 6 | 14460.00 | 513.30 |
| | 260 | Jones | 10 | Mgr | 12 | 21234.00 | - |
| | 270 | Lea | 66 | Mgr | 9 | 18555.50 | - |
| | 280 | Wilson | 66 | Sales | 9 | 18674.50 | 811.50 |
| | 290 | Quill | 84 | Mgr | 10 | 19818.00 | - |
| | 300 | Davis | 84 | Sales | 5 | 15454.50 | 806.10 |
| | 310 | Graham | 66 | Sales | 13 | 21000.00 | 200.30 |
| | 320 | Gonzales | 66 | Sales | 4 | 16858.20 | 844.00 |
| | 330 | Burke | 66 | Clerk | 1 | 10988.00 | 55.50 |
| | 340 | Edwards | 84 | Sales | 7 | 17844.00 | 1285.00 |
| | 350 | Gafney | 84 | Clerk | 5 | 13030.50 | 188.00 |

## Sample Files with BLOB and CLOB Data Type

This section shows the data found in the EMP_PHOTO files (pictures of employees) and EMP_RESUME files (resumes of employees).

### Quintana Photo

## Sample Tables



*Figure 14. Delores M. Quintana*

### Quintana Resume

The following text is found in the db200130.asc and db200130.scr files.

**Resume: Delores M. Quintana**

**Personal Information**
**Address:**              1150 Eglinton Ave Mellonville, Idaho 83725
**Phone:**                (208) 555-9933
**Birthdate:**            September 15, 1925
**Sex:**                  Female
**Marital Status:**       Married
**Height:**               5′2″
**Weight:**               120 lbs.

**Department Information**
**Employee Number:**      000130
**Dept Number:**          C01
**Manager:**              Sally Kwan
**Position:**             Analyst
**Phone:**                (208) 555-4578
**Hire Date:**            1971-07-28

**Education**

**1965**                  Math and English, B.A. Adelphi University

**1960**                  Dental Technician Florida Institute of
                          Technology

**Work History**

| **10/91 - present** | Advisory Systems Analyst Producing documentation tools for engineering department. |
| **12/85 - 9/91** | Technical Writer Writer, text programmer, and planner. |
| **1/79 - 11/85** | COBOL Payroll Programmer Writing payroll programs for a diesel fuel company. |

**Interests**
- Cooking
- Reading
- Sewing
- Remodeling

## Nicholls Photo



Figure 15. Heather A. Nicholls

## Nicholls Resume

The following text is found in the db200140.asc and db200140.scr files.

**Resume: Heather A. Nicholls**

**Personal Information**
| **Address:** | 844 Don Mills Ave Mellonville, Idaho 83734 |
| **Phone:** | (208) 555-2310 |
| **Birthdate:** | January 19, 1946 |
| **Sex:** | Female |
| **Marital Status:** | Single |
| **Height:** | 5'8" |

## Sample Tables

| | |
|---|---|
| **Weight:** | 130 lbs. |

**Department Information**

| | |
|---|---|
| **Employee Number:** | 000140 |
| **Dept Number:** | C01 |
| **Manager:** | Sally Kwan |
| **Position:** | Analyst |
| **Phone:** | (208) 555-1793 |
| **Hire Date:** | 1976-12-15 |

**Education**

| | |
|---|---|
| **1972** | Computer Engineering, Ph.D. University of Washington |
| **1969** | Music and Physics, M.A. Vassar College |

**Work History**

| | |
|---|---|
| **2/83 - present** | Architect, OCR Development Designing the architecture of OCR products. |
| **12/76 - 1/83** | Text Programmer Optical character recognition (OCR) programming in PL/I. |
| **9/72 - 11/76** | Punch Card Quality Analyst Checking punch cards met quality specifications. |

**Interests**
- Model railroading
- Interior decorating
- Embroidery
- Knitting

**Adamson Photo**
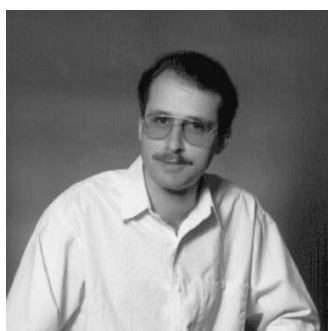
*Figure 16. Bruce Adamson*

## Adamson Resume

The following text is found in the db200150.asc and db200150.scr files.

**Resume: Bruce Adamson**

**Personal Information**
Address:                3600 Steeles Ave Mellonville, Idaho 83757
Phone:                  (208) 555-4489
Birthdate:              May 17, 1947
Sex:                    Male
Marital Status:         Married
Height:                 6'0"
Weight:                 175 lbs.

**Department Information**
Employee Number:        000150
Dept Number:            D11
Manager:                Irving Stern
Position:               Designer
Phone:                  (208) 555-4510
Hire Date:              1972-02-12

**Education**

1971                    Environmental Engineering, M.Sc. Johns
                        Hopkins University

1968                    American History, B.A. Northwestern
                        University

**Work History**

## Sample Tables

| | |
|---|---|
| **8/79 - present** | Neural Network Design Developing neural networks for machine intelligence products. |
| **2/72 - 7/79** | Robot Vision Development Developing rule-based systems to emulate sight. |
| **9/71 - 1/72** | Numerical Integration Specialist Helping bank systems communicate with each other. |

**Interests**
- Racing motorcycles
- Building loudspeakers
- Assembling personal computers
- Sketching

## Walker Photo
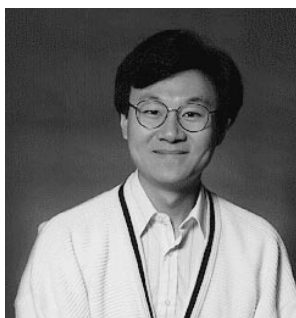


*Figure 17. James H. Walker*

## Walker Resume

The following text is found in the db200190.asc and db200190.scr files.

**Resume: James H. Walker**

**Personal Information**

| | |
|---|---|
| **Address:** | 3500 Steeles Ave Mellonville, Idaho 83757 |
| **Phone:** | (208) 555-7325 |
| **Birthdate:** | June 25, 1952 |
| **Sex:** | Male |
| **Marital Status:** | Single |
| **Height:** | 5'11" |
| **Weight:** | 166 lbs. |

**Sample Tables**

**Department Information**

| | |
|---|---|
| **Employee Number:** | 000190 |
| **Dept Number:** | D11 |
| **Manager:** | Irving Stern |
| **Position:** | Designer |
| **Phone:** | (208) 555-2986 |
| **Hire Date:** | 1974-07-26 |

**Education**

| | |
|---|---|
| **1974** | Computer Studies, B.Sc. University of Massachusetts |
| **1972** | Linguistic Anthropology, B.A. University of Toronto |

**Work History**

| | |
|---|---|
| **6/87 - present** | Microcode Design Optimizing algorithms for mathematical functions. |
| **4/77 - 5/87** | Printer Technical Support Installing and supporting laser printers. |
| **9/74 - 3/77** | Maintenance Programming Patching assembly language compiler for mainframes. |

**Interests**
- Wine tasting
- Skiing
- Swimming
- Dancing

**Sample Tables**

# Appendix G. Reserved Schema Names and Reserved Words

This appendix describes the restrictions of certain names used 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 though not prevented by the database manager.

## Reserved Schemas

The following schema names are reserved:
- SYSCAT
- SYSFUN
- SYSIBM
- SYSSTAT

In addition, it is strongly recommended that schema names never begin with the SYS prefix, as SYS is by convention used to indicate an area reserved by the system.

No user-defined functions, user-defined types, triggers, or aliases can be placed into a schema whose name starts with SYS (SQLSTATE 42939).

## Reserved Words

There are no words that are specifically reserved words in DB2 Version 6.

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.

IBM SQL and ISO/ANSI SQL92 include reserved words, listed in the following section. These reserved words are not enforced by DB2 Universal Database, however it is recommended that they not be used as ordinary identifiers, since this reduces portability.

## IBM SQL Reserved Words

The IBM SQL reserved words are as follows.

## Reserved Schema Names and Reserved Words

| | | | | |
|---|---|---|---|---|
| ACQUIRE | DATABASE | IDENTIFIED | OBID | SCHEDULE |
| ADD | DATE | IMMEDIATE | OF | SCHEMA |
| ALL | DAY | IN | ON | SECOND |
| ALLOCATE | DAYS | INDEX | ONLY | SECONDS |
| ALTER | DBA | INDICATOR | OPTIMIZE | SECQTY |
| AND | DBSPACE | INNER | OPTION | SELECT |
| ANY | DEFAULT | INOUT | OR | SET |
| AS | DELETE | INSERT | ORDER | SHARE |
| ASC | DESC | INTERSECT | OUT | SIMPLE |
| AUDIT | DESCRIPTOR | INTO | OUTER | SOME |
| AUTHORIZATION | DISTINCT | IS | | STATISTICS |
| AVG | DOUBLE | ISOLATION | PACKAGE | STOGROUP |
| | DROP | | PAGE | STORPOOL |
| BETWEEN | | JOIN | PAGES | SUBPAGES |
| BUFFERPOOL | EDITPROC | | PART | SUBSTRING |
| BY | END-EXEC | KEY | PCTFREE | SUM |
| | ELSE | | PCTINDEX | SYNONYM |
| CALL | ERASE | LABEL | PLAN | |
| CAPTURE | ESCAPE | LEFT | PRECISION | TABLE |
| CASE | EXCEPT | LIKE | PRIMARY | TABLESPACE |
| CAST | EXCEPTION | LOCK | PRIQTY | TO |
| CCSID | EXCLUSIVE | LOCKSIZE | PRIVATE | TRANSACTION |
| CHAR | EXECUTE | LONG | PRIVILEGES | TRIM |
| CHARACTER | EXISTS | | PROCEDURE | |
| CHECK | EXPLAIN | MAX | PROGRAM | UNION |
| CLUSTER | EXTERNAL | MICROSECOND | PUBLIC | UNIQUE |
| COLLECTION | | MICROSECONDS | | UPDATE |
| COLUMN | FETCH | MIN | REFERENCES | USER |
| COMMENT | FIELDPROC | MINUTE | RELEASE | USING |
| COMMIT | FOR | MINUTES | RESET | |
| CONCAT | FOREIGN | MODE | RESOURCE | VALIDPROC |
| CONNECT | FROM | MONTH | REVOKE | VALUES |
| CONNECTION | FULL | MONTHS | RIGHT | VARIABLE |
| CONSTRAINT | | | ROLLBACK | VCAT |
| COUNT | GO | NAMED | ROW | VIEW |
| CREATE | GOTO | NHEADER | ROWS | VOLUMES |
| CROSS | GRANT | NOT | RRN | |
| CURRENT | GRAPHIC | NULL | RUN | WHERE |
| CURRENT_DATE | GROUP | NUMPARTS | | WITH |
| CURRENT_SERVER | | | | WORK |
| CURRENT_TIME | HAVING | | | |
| CURRENT_TIMESTAMP | HOUR | | | YEAR |
| CURRENT_TIMEZONE | HOURS | | | YEARS |
| CURRENT_USER | | | | |
| CURSOR | | | | |

## ISO/ANS SQL92 Reserved Words

The ISO/ANS SQL92 reserved words that are not also in the IBM SQL list are as follows.

| | | | |
|---|---|---|---|
| ABSOLUTE | FALSE | OCTET_LENGTH | UNKNOWN |
| ACTION | FIRST | OPEN | UPPER |
| ARE | FLOAT | OUTER | USAGE |
| ASSERTION | FOUND | OUTPUT | |
| AT | FULL | OVERLAPS | VALUE |
| | | | VARCHAR |
| BEGIN | GET | PAD | VARYING |
| BIT_LENGTH | GLOBAL | PARTIAL | |
| BOTH | | POSITION | WHEN |
| | IDENTITY | PREPARE | WHENEVER |
| CASCADED | INITIALLY | PRESERVE | WRITE |
| CATALOG | INNER | PRIOR | |
| CHAR_LENGTH | INPUT | | ZONE |
| CHARACTER_LENGTH | INSENSITIVE | READ | |
| CLOSE | INTERVAL | REAL | |
| COALESCE | | RELATIVE | |
| COLLATE | JOIN | RIGHT | |
| COLLATION | | | |
| CONSTRAINTS | LANGUAGE | SCROLL | |
| CONTINUE | LAST | SECTION | |
| CONVERT | LEADING | SESSION | |
| CORRESPONDING | LEFT | SESSION_USER | |
| | LEVEL | SIZE | |
| DEALLOCATE | LOCAL | SMALLINT | |
| DEC | LOWER | SPACE | |
| DECIMAL | | SQL | |
| DECLARE | MATCH | SQLCODE | |
| DEFERRABLE | MODULE | SQLERROR | |
| DEFERRED | | SQLSTATE | |
| DESCRIBE | NAMES | SYSTEM_USER | |
| DIAGNOSTICS | NATIONAL | | |
| DISCONNECT | NATURAL | TEMPORARY | |
| DOMAIN | NCHAR | THEN | |
| | NEXT | TIMEZONE_HOUR | |
| END | NO | TIMEZONE_MINUTE | |
| EXTRACT | NULLIF | TRAILING | |
| | NUMERIC | TRANSLATION | |
| | | TRUE | |

**Reserved Schema Names and Reserved Words**

# Appendix H. Comparison of Isolation Levels

The following table summarizes information about isolation levels described in "Isolation Level" on page 25.

|  | UR | CS | RS | RR |
|---|---|---|---|---|
| Can the application see uncommitted changes made by other application processes? | Yes | No | No | No |
| Can the application update uncommitted changes made by other application processes? | No | No | No | No |
| Can the re-execution of a statement be affected by other application processes? *See phenomenon P3 (phantom) below*. | Yes | Yes | Yes | No |
| Can "updated" rows be updated by other application processes? | No | No | No | No |
| Can "updated" rows be read by other application processes that are running at an isolation level other than UR? | No | No | No | No |
| Can "updated" rows be read by other application processes that are running at the UR isolation level? | Yes | Yes | Yes | Yes |
| Can "accessed" rows be updated by other application processes? *See phenomenon P2 (nonrepeatable read) below*. | Yes | Yes | No | No |
| Can "accessed" rows be read by other application processes? | Yes | Yes | Yes | Yes |
| Can "current" row be updated or deleted by other application processes? *See phenomenon P1 (dirty-read) below*. | See Note below | See Note below | No | No |

**Note:**

1. If the cursor is not updatable, with CS the current row may be updated or deleted by other application processes in some cases.

## Isolation Levels

| | UR | CS | RS | RR |
|---|---|---|---|---|

**Examples of Phenomena:**

**P1**     *Dirty Read.* Unit of work UW1 modifies a row. Unit of work UW2 reads that row before UW1 performs a COMMIT. If UW1 then performs a ROLLBACK, UW2 has read a nonexistent row.

**P2**     *Nonrepeatable Read.* Unit of work UW1 reads a row. Unit of work UW2 modifies that row and performs a COMMIT. If UW1 then re-reads the row, it might receive a modified value.

**P3**     *Phantom.* Unit of work UW1 reads the set of $n$ rows that satisfies some search condition. Unit of work UW2 then INSERTs one or more rows that satisfies the search condition. If UW1 then repeats the initial read with the same search condition, it obtains the original rows plus the inserted rows.

# Appendix I. Interaction of Triggers and Constraints

This appendix describes the interaction of triggers with referential constraints and check constraints that may result from an update operation. Figure 18 and the associated description are representative of the processing that is performed for an SQL statement that updates data in the database.
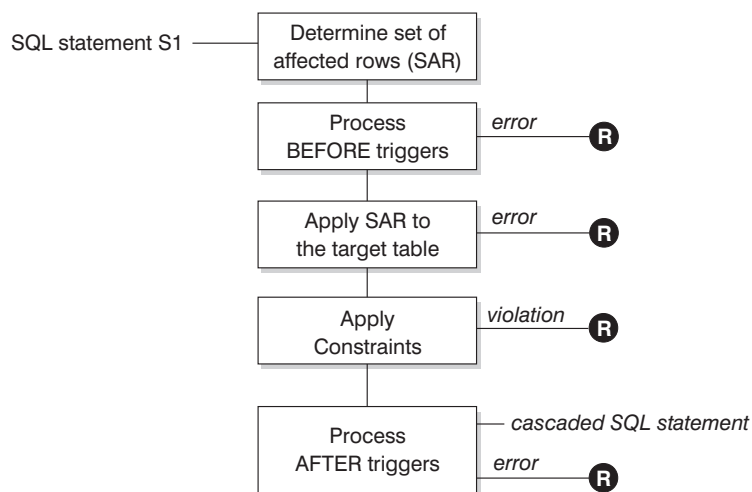


*Figure 18. Processing an SQL statement with associated triggers and constraints*

Figure 18 shows the general order of processing for an SQL statement that updates a table. It assumes a situation where the table includes before triggers, referential constraints, check constraints and after triggers that cascade. The following is a description of the boxes and other items found in Figure 18.

- SQL statement $S_1$

  This is the DELETE, INSERT, or UPDATE statement that begins the process. The SQL statement $S_1$ identifies a table (or an updatable view over some table) referred to as the *target table* throughout this description.

- Determine set of affected rows (*SAR*)

  This step is the starting point for a process that repeats for referential constraint delete rules of CASCADE and SET NULL and for cascaded SQL statements from after triggers.

  The purpose of this step is to determine the *set of affected rows* for the SQL statement. The set of rows included in *SAR* is based on the statement:

## Interaction of Triggers and Constraints

- for DELETE, all rows that satisfy the search condition of the statement (or the current row for a positioned DELETE)
- for INSERT, the rows identified by the VALUES clause or the fullselect
- for UPDATE, all rows that satisfy the search condition (or the current row for a positioned update).

If *SAR* is empty, there will be no BEFORE triggers, changes to apply to the target table, or constraints to process for the SQL statement.

- Process BEFORE triggers

  All BEFORE triggers are processed in ascending order of creation. Each BEFORE trigger will process the triggered action once for each row in *SAR*.

  An error may occur during the processing of a triggered action in which case all changes made as a result of the original SQL statement $S_1$ (so far) are rolled back.

  If there are no BEFORE triggers or the *SAR* is empty, this step is skipped.

- Apply *SAR* to the target table

  The actual delete, insert, or update is applied using *SAR* to the target table in the database.

  An error may occur when applying *SAR* (such as attempting to insert a row with a duplicate key where a unique index exists) in which case all changes made as a result of the original SQL statement $S_1$ (so far) are rolled back.

- Apply Constraints

  The constraints associated with the target table are applied if *SAR* is not empty. This includes unique constraints, unique indexes, referential constraints, check constraints and checks related to the WITH CHECK OPTION on views. Referential constraints with delete rules of cascade or set null may cause additional triggers to be activated.

  A violation of any constraint or WITH CHECK OPTION results in an error and all changes made as a result of $S_1$ (so far) are rolled back.

- Process AFTER triggers

  All AFTER triggers activated by $S_1$ are processed in ascending order of creation.

  FOR EACH STATEMENT triggers will process the triggered action exactly once, even if *SAR* is empty. FOR EACH ROW triggers will process the triggered action once for each row in *SAR*.

  An error may occur during the processing of a triggered action in which case all changes made as a result of the original $S_1$ (so far) are rolled back.

  The triggered action of a trigger may include triggered SQL statements that are DELETE, INSERT or UPDATE statements. For the purposes of this description, each such statement is considered a *cascaded SQL statement*.

  A cascaded SQL statement is a DELETE, INSERT, or UPDATE statement that is processed as part of the triggered action of an AFTER trigger. This

statement starts a cascaded level of trigger processing. This can be thought of as assigning the triggered SQL statement as a new $S_1$ and performing all of the steps described here recursively.

Once all triggered SQL statements from all AFTER triggers activated by each $S_1$ have been processed to completion, the processing of the original $S_1$ is completed.

- $\boxed{\text{R}}$ = roll back changes to before $S_1$

  Any error (including constraint violations) that occurs during processing results in a roll back of all the changes made directly or indirectly as a result of the original SQL statement $S_1$. The database is therefore back in the same state as immediately prior to the execution of the original SQL statement $S_1$

**Interaction of Triggers and Constraints**

# Appendix J. Incompatibilities Between Releases

This appendix identifies the incompatibilities that exist between DB2 Universal Database and previous releases of DB2.

An "incompatibility" is defined to be a part of DB2 Universal Database that works differently than it did in a previous release of DB2 in such a way that if it is used in an existing application, it will either produce a different result, necessitate a change to the application, or reduce performance. In this definition, "application" can apply to a broad range of things, such as:
- Application program code
- Third-party utilities
- Interactive SQL queries
- Command and/or API invocation.

This appendix does not describe incompatibilities where certain operations in the current release are less likely to generate an error condition than they did in the previous release, as those changes will only have a positive impact on existing applications.

This appendix lists incompatibilities in the following categories:
- System Catalog Views
- Application Programming
- SQL
- Database Security and Tuning
- Utilities and Tools
- Connectivity and Coexistence
- Configuration Parameters.

These categories are found in each of two major sections:
- DB2 Universal Database Version 6 Incompatibilities
- DB2 Universal Database Version 5 Incompatibilities

Each incompatibility includes a description of the change in that particular version that causes an incompatibility with previous releases, the symptom or effect this will have on your environment if no changes are made to it, and the possible resolutions that are available. There is also an indicator at the beginning of each incompatibility telling you what platforms are applicable, as follows:

**OS/2** OS/2

**UNIX**    Unix-based operating systems supported by DB2

**WIN**    Microsoft Windows platforms supported by DB2

**Extended**
    DB2 Extended Edition, and/or Enterprise – Extended Edition

**DB2 PE**
    DB2 Parallel Edition, Version 1.2 (shown only for previous version consistency)

## DB2 Universal Database Planned Incompatibilities

This section focuses on the DB2 Universal Database incompatibilities that we are reserving the right to do two versions in the future, following Version 6. Users of DB2 Universal Database are warned to code applications with these in mind, as well as to modify existing applications to make migration to future versions easier.

### Read-only Views in a Future Version of DB2 Universal Database

| UNIX | OS/2 | WIN | Extended |
|------|------|-----|----------|

**Change**

The system catalog views will be read-only views. The SYSSTAT views will continue to be updateable.

**Symptom**

UPDATE statements which used to work against columns in the SYSCAT views will fail.

**Explanation**

Tools or applications are coded to change values in the catalog by updating the column as defined in the SYSCAT view.

**Resolution**

Change the tool or application to change the catalog by updating the column as defined in the SYSSTAT view.

### PK_COLNAMES and FK_COLNAMES in a Future Version of DB2 Universal Database

| UNIX | OS/2 | WIN | Extended |
|------|------|-----|----------|

**Change**

The SYSCAT.REFERENCES columns PK_COLNAMES and FK_COLNAMES will no longer be available.

**Symptom**

Column does not exist and an error is returned.

**Explanation**

Tools or applications are coded to use the obsolete PK_COLNAMES and FK_COLNAMES columns.

**Resolution**

Change the tool or application to use the SYSCAT.KEYCOLUSE view instead.

## COLNAMES No Longer Available in a Future Version of DB2 Universal Database

| UNIX | OS/2 | WIN | Extended |
|------|------|-----|----------|

**Change**

The SYSCAT.INDEXES column COLNAMES will no longer be available.

**Symptom**

Column does not exist and an error is returned.

**Explanation**

Tools or applications are coded to use the obsolete COLNAMES column.

**Resolution**

Change the tool or application to use the SYSCAT.INDEXCOLUSE view instead.

## DB2 Universal Database Version 6 Incompatibilities

This section focuses on the DB2 Universal Database Version 6 incompatibilities.

## System Catalog Views

### System Catalog Views in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | |
|------|------|-----|--|

**Change:**

In the system catalog views, new codes have been introduced: "U" for typed tables, and "W" for typed views.

**Symptom:**

Queries that search for tables and views in the system catalogs, using the typecode "T" for tables and "V" for views, will no longer find typed tables and views.

**Explanation:**

Several system catalogs, including the system catalog views named TABLES, PACKAGEDEP, TRIGDEP, and VIEWDEP, have a column named TYPE or BTYPE containing a one-letter typecode. In Version 5.2, the typecode "T" was used for all tables, and "V" was used for all views. In Version 6, untyped tables will continue to have a typecode of "T" and typed tables will have a new typecode of "U". Similarly, untyped views will continue to have a typecode of "V" and typed views will have a new typecode of "W". Also, a new kind of table called a hierarchy table, not directly created by users but used by the system to implement table hierarchies, will appear in the system catalog tables with a typecode of "H".

**Resolution:**

Change the tool or application to recognize the codes for typed tables and views. If the tool or application needs a logical view of tables, then typecodes "T", "U", "V", and "W" should be used. If the tool or application needs a physical view of tables, including hierarchy tables, then typecodes "T" and "H" should be used.

### Primary and Foreign Key Column Names in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | |
|------|------|-----|--|

**Change:**

Data type change to two SYSCAT.REFERENCES columns, PK_COLNAMES and FK_COLNAMES, from VARCHAR(320) to VARCHAR(640).

**Symptom:**

Primary key and/or foreign key column names are truncated, are not correct, or are missing.

**Explanation:**

When column names greater than 18 bytes in length are used in a primary key or foreign key, the format under which the list of column names are stored in these two columns cannot remain the same. The 20-byte blank delimited column name(s) coming after the column whose length is greater than 18 will be shifted to the right the number of bytes that the column whose length is greater than 18 is over 18 bytes. As well, if the list of column names exceeds 640 bytes, the column will contain the empty string.

**Resolution:**

The view SYSCAT.KEYCOLUSE contains the list of columns that make up a primary, foreign, as well as a unique key and should be used instead of the columns in SYSCAT.REFERENCES. Alternatively, users can restrict the length of column names to 18 bytes or restrict the total length of the list of columns to 640 bytes.

### SYSCAT.VIEWS Column TEXT in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | |
|------|------|-----|--|

**Change:**

View text in the SYSCAT.VIEWS column TEXT will now not be split across multiple rows. The data type is changed from VARCHAR(3600) to CLOB(64K).

**Symptom:**

The complete view text is not given by the tool or in the application.

**Explanation:**

Tools or applications which were coded expecting no more than 3600 (or perhaps 3900) bytes to be returned from the TEXT column at one time are not handling the increased size of this field. The mechanism of retrieving multiple rows and reconstructing the view text using the SEQNO field is no longer necessary. The SEQNO value will now only ever be 1.

**Resolution:**

Change the tool or application to be able to handle values from the TEXT column which are greater than 3600 bytes. Alternatively, the view TEXT could be rewritten to fit within 3600 bytes.

## SYSCAT.STATEMENTS Column TEXT in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | |
|------|------|-----|---|

**Change:**

Statement text in the SYSCAT.STATEMENTS column TEXT will now not be split across multiple rows. The data type is changed from VARCHAR(3600) to CLOB(64K).

**Symptom:**

The complete statement text is not given by the tool or in the application.

**Explanation:**

Tools or applications which were coded expecting no more than 3600 (or perhaps 3900) bytes to be returned from the TEXT column at one time are not handling the increased size of this field. The mechanism of retrieving multiple rows and reconstructing the statement text using the SEQNO field is no longer necessary. The SEQNO value will now only ever be 1.

**Resolution:**

Change the tool or application to be able to handle values from the TEXT column which are greater than 3600 bytes. Alternatively, the statement TEXT could be rewritten to fit within 3600 bytes.

## SYSCAT.INDEXES Column COLNAMES in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | |
|------|------|-----|---|

**Change:**

The SYSCAT.INDEXES column COLNAMES data type is changed from VARCHAR(320) to VARCHAR(640).

**Symptom:**

Column names are missing in an index.

**Explanation:**

Tools or applications are coded to retrieve from a column with a data type of VARCHAR(320) and cannot handle the increased size of this field.

**Resolution:**

The view SYSCAT.INDEXCOLUSE contains the list of columns that make up an index and should be used instead of the column COLNAMES. Alternatively, remove a column from the index or reduce the size of the name of a column so that the list of column names (with the leading + or −) will fit in 320 bytes.

### SYSCAT.CHECKS Column TEXT in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | |
|------|------|-----|---|

**Change:**

CHECKS Column TEXT data type is changed from CLOB(32K) to CLOB(64K).

**Symptom:**

Check constraint clause is incomplete.

**Explanation:**

Tools or applications are coded to retrieve from a column with a data type of CLOB(32K) and cannot handle the increased size of this field.

**Resolution:**

Change the tool or application to be able to handle values from the TEXT column which are longer than 32K bytes. Alternatively, rewrite the check constraint clause to use fewer characters such that it will fit in 32K bytes.

### Column Data Type to BIGINT in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | |
|------|------|-----|---|

**Change:**

Several system catalog view columns have had their data type changed from INTEGER to BIGINT.

**Symptom:**

Values are much smaller (or larger) than expected, especially statistical information.

**Explanation:**

Tools or applications are coded to retrieve from a column with a data type of INTEGER and cannot handle the increased size of this field.

**Resolution:**

Change the tool or application to be able to handle values which are greater than the maximum or minimum value which can be stored in an INTEGER field. Alternatively, change the underlying structure or SQL code which causes the value to be greater than what can be represented in an INTEGER field.

### Column Mismatch in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | |
|------|------|-----|--|

**Change:**

New columns are not inserted at the end of views in the SYSCAT view definition.

**Symptom:**

Re-preprocessing fails with several column mismatches or column data type mismatches.

**Explanation:**

New columns are introduced to the system catalog views and placed in a position that is useful in an ad-hoc query environment, specifically, shorter columns are placed before very long columns and the REMARKS column is always the last one.

**Resolution:**

Explicitly name the columns in the select list instead of coding "SELECT *".

### SYSCAT.COLUMNS and SYSCAT.ATTRIBUTES in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | |
|------|------|-----|--|

**Change:**

SYSCAT.COLUMNS and SYSCAT.ATTRIBUTES now contain entries for inherited columns and attributes.

**Symptom:**

Queries of SYSCAT.COLUMNS to retrieve the columns of a typed table or view, and queries of SYSCAT.ATTRIBUTES to retrieve the attributes of a structured type, may return more rows in V6 than in V5.2, if the subject of the query is a subtable, subview, or subtype.

**Explanation:**

In Version 5.2, for a given table, view, or structured type, the COLUMNS and ATTRIBUTES catalogs contained entries only for columns and attributes that were introduced by that table, view, or type. Columns and attributes that were inherited from supertables or supertypes were not represented in the catalogs. However, in V6, the COLUMNS and ATTRIBUTES catalogs will contain entries for inherited columns and attributes.

**Resolution:**

Change the tool or application to recognize the new entries in the COLUMNS and ATTRIBUTES catalogs.

### OBJCAT Views No Longer Supported in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | Extended |
|------|------|-----|----------|

**Change:**

The recursive catalog views in the OBJCAT schema of Version 5.2 are no longer part of the shipped DB2 Universal Database product.

**Symptom:**

Queries written against the OBJCAT catalog views will no longer run successfully.

**Resolution:**

Most of the information formerly in the OBJCAT views has been incorporated into the regular SYSCAT catalog views. In most cases, you can obtain the information from the system catalog views. If you migrate from Version 5.2, and the OBJCAT catalog views exist, they should be dropped. This can be done by executing the CLP script called `objcatdp.db2` found under the `misc` subdirectory of the `sqllib` subdirectory.

If you wish, you could also create your own set of OBJCAT views that are equivalent to the catalog views supported in Version 5.2.

In version 5.2, the SQL Reference in Appendix E warned users that the OBJCAT catalog views were temporary and would not be supported in future releases.

## Dependency Codes Changed in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | Extended |
|------|------|-----|----------|

**Change:**

In the system catalog views, the hierarchic dependencies formerly denoted by code "H" are now denoted by code "O".

**Symptom:**

Queries that search for hierarchic dependencies by code "H" in the catalog views will no longer work correctly.

**Explanation:**

Several system catalogs, including the system catalog views named PACKAGEDEP, TRIGDEP, and VIEWDEP have a column named BTYPE. In Version 5.2, the OBJCAT views denoted hierarchic dependencies with the code "H". In Version 6, these dependencies are denoted with the code "O".

**Resolution:**

These queries will need to be revised to search for code "O".

## SYSIBM Base Catalog Tables in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | Extended |
|------|------|-----|----------|

**Change:**

The following are changes made to the SYSIBM base catalog tables, which customers may be using despite our encouragement for them to use the SYSCAT views. Customers who are not following our recommendation of coding to the SYSCAT views may experience incompatibilities due to the following changes:

1. Deleted fields (but still in the SYSCAT views):
   - SYSSTMT.SEQNO
   - SYSVIEWS.SEQNO

2. Renamed catalog table: SYSTRIGDEP named to SYSDEPENDENCIES. As well, the columns BCREATOR and DCREATOR were renamed to BSCHEMA and DSCHEMA respectively. The view SYSCAT.TRIGDEP did not change.
3. Deleted fields (were never in the SYSCAT views):
   - SYSATTRIBUTES.DEFAULT_VALUE
   - SYSATTRIBUTES.NULLS
   - SYSCOLUMNS.SERVERTYPE
   - SYSDATATYPES.REFREP_TYPENAME
   - SYSDATATYPES.REFREP_TYPESCHEMA
   - SYSDATATYPES.REFREP_LENGTH
   - SYSDATATYPES.REFREP_SCALE
   - SYSDATATYPES.REFREP_CODEPAGE
   - SYSINDEXES.TEXT
     (Was in the view, but reserved for future use only.)
   - SYSPLANDEP.PUBLICPRIV
   - SYSSECTION.SEQNO
   - SYSTABAUTH.UPDATE_BY_COLS
   - SYSTABAUTH.REF_BY_COLS
   - SYSTABLES.MINPDLENGTH
   - SYSTABLESPACES.READONLY
   - SYSTABLESPACES.REMOVABLEMEDIA
4. Data type changes:
   - SYSSECTION.SECTION from VARCHAR(3600) to CLOB(10M)
   - SYSPLANDEP.COLUSAGE from VARCHAR(3000) FOR BIT DATA to BLOB(5K)

## Application Programming

### VARCHAR Data Type in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | |
|------|------|-----|---|

**Change:**

Maximum possible size of VARCHAR (VARGRAPHIC) data type has increased from 4 000 characters (2 000 double bytes characters), to 32 672 characters (16 336 double bytes characters) in Version 6.

**Symptom:**

An application that uses fixed length buffers of 4 000 bytes for a VARCHAR (VARGRAPHIC) data type has the potential for buffer overwrite or truncation, if it fetches a VARCHAR field which is longer than 4 000 bytes into a buffer that is too small. The CLI function - SQLGetTypeInfo() now returns the size of VARCHAR as 32 672. CLI applications that use this value in table DDLs may get errors due to sufficient page size table spaces not being available.

**Resolution:**

The application should be coded, in the recommended manner, of first describing the columns of the result set by using the DESCRIBE statement, and then using buffers whose size is based on the data type's length as returned from the DESCRIBE of the column.

### Java Programming Positioned UPDATE and DELETE in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | |
|------|------|-----|--|

**Change:**

When programming using Java in Version 6, positioned UPDATE and DELETE statements use the default authorization identifier of the person that bound the cursor package. This is different from Version 5.2 where the authorization identifier of the person running the package was used.

**Symptom:**

The package containing the positioned UPDATE and DELETE statements may not run because the authorization identifier of the person who bound the package does not have sufficient authority.

**Resolution:**

The authorization identifier of the person who binds the package must be granted sufficient authority to run the positioned UPDATE and DELETE statements found in the package. Grant the correct privileges and then re-bind the package.

### Syntax Change in FOR UPDATE Clause in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | |
|------|------|-----|--|

**Change:**

In Version 5.2, in an SQLJ program, the FOR UPDATE clause can be used in a SELECT statement to identify the columns that can be updated in subsequent positioned UPDATE statements. The syntax is changed for Version 6.

**Symptom:**

You will receive the error message SQJ0204E if a SELECT statement contains a FOR UPDATE clause.

**Resolution:**

Remove the FOR UPDATE clause from the SELECT statement. Specify an updatable iterator through the iterator declaration clause. For example:

```
#sql public iterator DelByName implements sqlj.runtime.ForUpdate(String EmpNo)
    with updateColumns = (salary);
```

If you want to explicitly identify what columns are updateable, specify them through the `updateColumns` keyword, used in conjunction with the WITH clause.

Refer to *Application Development Guide* for more information on positioned iterator declarations.

### Character Name Sizes in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | Extended |
|------|------|-----|----------|

**Change:**

DB2 Universal Database Version 6 supports 128 byte table/view/alias names and 30 byte column names. The former support was for 18 byte names for each of these entities.

USER and CURRENT SCHEMA special registers were CHAR(8) and are now VARCHAR(128). The CURRENT EXPLAIN MODE special register was CHAR(8) and is now VARCHAR(254). The output for TYPE_SCHEMA and TABLE_SCHEMA built-in functions were CHAR(8) and are now VARCHAR(128).

**Symptom:**

If applications that were developed before Version 6 are run against a Version 6 database that does not utilize the longer limits, then the application behavior should not change at all. However, running these applications against a Version 6 database that does utilize longer names, could result in certain side effects, depending on how these applications were coded.

Here are some examples:

- Consider an existing application that FETCHes a table or column name (typically from a catalog view) into a host variable that was defined to be 18 bytes long. Since 18 bytes was the limit on the size of the table or column name until Version 6, this application may not bother to check the `sqlwarn1` bit of the SQLCA, assuming that there should never be truncation. This application may proceed with subsequent logic, assuming that no truncation has occurred whereas truncation actually did occur.

- Consider an application that FETCHes a table or column name (typically from a catalog view) into an SQLDA where the size of the sqldata field was allocated based on the `sqllen` field from a DESCRIBE of the SELECT. This will result in the correct (untruncated) result being returned to the application even though the size of the table/column names may have increased. If other application logic relies on the assumption that column names or lengths are limited to 18 bytes, then the longer names that have been returned may introduce unintended behavior in the application. As a simple example, the display of a longer column name may be truncated at 18 bytes.

- Since the SQLCA token field (`sqlerrmc`) is limited to 70 bytes, this may affect existing applications that attempt to insert a row into a table. In the event of the error message SQL0204N, this application determines the name of the table from the SQLCA `sqlerrmc` field and then performs some operations based on that object name. Whereas with earlier versions of DB2, the table/schema identifier limit guaranteed that the table name was included in the SQLCA in its entirety (as in the case with the error message SQL0204N), this is not the case with Version 6.

- An application using a downlevel API (such as LIST HISTORY, or GET SNAPSHOT) will only get the first 18 bytes of a table name.

- Existing CLI and ODBC applications that use the schema functions (such as SQLTables(), or SQLColumns(), and others) will be affected if connecting to a server with support for greater than 18 byte names. Although there will be truncation warnings, the application may not check for this warning and may proceed with a truncated name.

**Resolution:**

The best way to resolve problems of this type is to re-code the application to handle longer table and column names. Otherwise, ensure that these applications are not run against Version 6 databases that use > 18 byte names.

**PC/IXF Format Changes in DB2 Universal Database Version 6**

| UNIX | OS/2 | WIN | Extended |
|------|------|-----|----------|

**Change:**

DB2 Universal Database Version 6 supports 128 byte table/view/alias names and 30 byte column names. The former support was for 18 byte names for each of these entities.

**Symptom:**

A DB2 Universal Database Version 5 client is not able to import a PC/IXF that was exported by a DB2 Universal Database Version 6 export client. The error message SQL3059N will result.

Also, a PC/IXF file (an export from a DB2 Universal Database Version 6 client) cannot be loaded into a DB2 Universal Database Version 5 database. The error message SQL3059N will result.

**Resolution:**

Always be aware of the version level of the PC/IXF file when running.

### SQLNAME in a Non-doubled SQLVAR in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | Extended |
|------|------|-----|----------|

**Change:**

DB2 Universal Database Version 6 supports 30 byte column names. The former support was for 18 byte names. In Version 5, the documented behavior was that "0xFF" is placed in the 30th byte of an SQLNAME field for a non-doubled SQLVAR. Also in Version 5, for system-generated names and for user-specified column names specified in an "AS" clause, "0x00" is placed in the 30th byte.

In Version 6, the new behavior only returns "0xFF" in the 30th column if the name is system-generated.

**Symptom:**

Any applications that rely on the 30th-byte of the SQLNAME to determine whether it is a user-specified column name or a system-generated name may receive unexpected logic checks if the user-specifed column name is 30 characters long. This should be a rare occurrence.

**Resolution:**

These applications should be modified to only check for "0xFF" in the 30th byte of the SQLNAME field if the length of the SQLNAME is less than 30. In this case, the name is user-generated.

### Obsolete DB2 CLI/ODBC Configuration Keywords in DB2 Universal Database Version 6

|  |  | WIN |
|---|---|---|

**Change:**

When moving from version to version, you can change the behavior of the DB2 CLI/ODBC driver by specifying a set of optional keywords in the *db2cli.ini* file.

In Version 6, the TRANSLATEDLL and the TRANSLATEOPTION keywords became obsolete.

**Symptom:**

These keywords will be ignored if they still exist. You may notice behavioral changes based on the removal of these settings.

**Resolution:**

You will need to review the new list of valid parameters to decide what the appropriate keywords and settings are for your environment. See the *CLI Guide and Reference* for information on these keywords.

### Event Monitor Output Stream Format in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | Extended |
|---|---|---|---|

**Change:**

Event monitor output streams have no version control. As a result, adding support for the greater than 18 byte table names requires the move to an output stream format.

**Symptom:**

Applications that parse the event monitor output streams will no longer function as in previous releases.

**Resolution:**

There are two options:
1. Update the application to use the new data stream.
2. Set the registry variable DB2OLDEVMON=evmonname1,evmonname2,... where "evmonname" is the name of the event monitor you wish to have

write in the old data format. Note that any new fields in the event monitor will not be accessible in the old format.

## SQL

### Datalink Columns in DB2 Universal Database Version 6

| UNIX | | |
|------|---|---|

**Change:**

Datalink values inserted in DB2 Universal Database Version 6 will require an extra four bytes of space in the column value descriptor.

**Symptom:**

When datalink columns created in Version 5.2 are updated, an additional four bytes are required on the data page to store the new column value. As a result, there may not be enough space in the data page to complete the update and it may have to be moved to a new page. This could cause the update to run out of space.

**Resolution:**

You will need to add more space on your system to allow for updates.

### SYSFUN String Function Signatures in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | Extended |
|------|------|-----|----------|

**Change:**

A number of string functions in the SYSFUN schema now have improved versions defined in the SYSIBM schema (built-in functions). The function names are LCASE, LTRIM, RTRIM, and UCASE.

**Symptom:**

When preparing statements or creating views, the returned data type from any of these functions may be different than running the same statement on versions previous to Version 6. This occurs because the built-in functions (under the SYSIBM schema) are usually resolved before functions in the SYSFUN schema.

**Resolution:**

No action is required. Usually, the behavior of the new built-in function is desired over the function as found in the SYSFUN schema. The previous

version behavior can be restored by switching the SQL path so that SYSFUN precedes SYSIBM, but performance of function resolution is degraded. Alternatively, the previous version function can be invoked by qualifying the function name with the schema name SYSFUN.

Migrated packages, views, summary tables, triggers, and constraints that reference these functions continue to use the version from the SYSFUN schema unless explicit action is taken such as explicitly binding the package or re-creating the view, summary table, trigger, or constraint.

### SYSTABLE Column Change With New Integrity State in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | Extended |
|------|------|-----|----------|

**Change:**

The "U" states in the CONST_CHECKED column of SYSCAT.TABLES changes differently when a SET INTEGRITY ... OFF statement is run.

**Symptom:**

Prior to Version 6, any "U" state in CONST_CHECKED column changes to an "N" when a SET INTEGRITY ... OFF statement is run. There is now another state, "W", to which the "U" state is changed.

**Resolution:**

No action is required.

The new "W" state in a CONST_CHECKED byte is used to indicate that the type of constraint was previously checked by the user and some data in the table may need to be checked for integrity.

Without this new state, the "U" state would be changed to the "N" state on a SET INTEGRITY ... OFF as it did in previous versions. From the "N" state alone, the database manager would not be sure if there exists any old data that has not yet been verified by the database manager. On a subsequent a SET INTEGRITY ... IMMEDIATE CHECKED INCREMENTAL statement, the database manager has to return an error. The error is returned because the database manager will not be able to guarantee data integrity if only the new change (if any) were checked.

With the new "W" state, on a subsequent a SET INTEGRITY ... IMMEDIATE CHECKED INCREMENTAL statement, the "W" state can be changed back to the "U" state if the INCREMENTAL option is specified to indicate that the user is still responsible for the data integrity of the table. If the

INCREMENTAL option is not specified, the database manager will pick full processing, change the "W" state to a "Y" state, and re-assume the responsibility of maintaining data integrity.

## Database Security and Tuning

### Creating Databases Using Clients in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | Extended |
|------|------|-----|----------|

**Change:**

The method used by clients to create a database.

**Symptom:**

Using a down-level client (from a previous release than that of the server) to create a database will result in errors.

**Resolution:**

When using a client to create a database, only use clients at the same level as that of the server.

### SELECT Privilege Required on Hierarchy in DB2 Universal Database Version 6

| UNIX | OS/2 | Windows 32-bit | Extended |
|------|------|----------------|----------|

**Change:**

Specification of the ONLY keyword with a table now requires that the user have SELECT privilege on all subtables of the specified typed table. Similarly, specification of the ONLY keyword with a view now requires that the user have SELECT privilege on all subviews of a specified typed table. Previous versions only required SELECT privilege on the specified table or view.

**Symptom:**

There are two possible symptoms:
1. An authorization error (SQLCODE -551, SQLSTATE 42501) occurs when rebinding a package containing an SQL statement that specified the ONLY keyword in a FROM clause, if the authorization ID under which the package was bound lacks SELECT privilege on the subtables of the named typed table (or view).
2. If the definition of a view or trigger contains the keyword ONLY used in a FROM clause, the view or trigger will continue to work normally.

However, the definition of the view or trigger can no longer be used to create a new view or trigger unless the creator holds SELECT privilege on all the subtables of the named table (or view).

**Resolution:**

The authorization ID that needs to rebind a package or to crate a new view or trigger should be granted SELECT privilege on all subtables (and subviews) of the table (or view) specified following the ONLY keyword.

### Obsolete Profile Registry and Environment Variables in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | DB2 PE |
|------|------|-----|--------|

**Change:**

The following profile registry values or environment variables are obsolete:
- DB2_VECTOR

**Resolution:**

There is no longer a need for these profile registry or environment variables.

## Utilities and Tools

### Current Explain Mode in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN |
|------|------|-----|

**Change:**

The type of the "CURRENT EXPLAIN MODE" special register has changed from CHAR(8) to VARCHAR(254).

**Symptom:** If the application assumes that the type is still CHAR(8), then the value may be truncated from 254 to 8 bytes.

**Resolution:**

Redefine the type of all host variables which read the special register from CHAR(8) to VARCHAR(254).

This change is required to accommodate two new values for the "CURRENT EXPLAIN MODE" special register. These new values are "EVALUATE INDEXES" and "RECOMMEND INDEXES".

### The USING and SORT BUFFER Parameters in DB2 Universal Database Version 6

| UNIX | OS/2 | WIN | DB2 PE |
|------|------|-----|--------|

**Change:**

As of Version 6, the USING and SORT BUFFER parameters of the LOAD command are no longer supported (the parameters are ignored).

**Symptom:**

If specified on the LOAD command, the use receives a warning message saying that USING and SORT BUFFER parameters are no longer supported and are ignored by the LOAD utility.

**Resolution:**

Ignore the warning message. See *Data Movement Utilities Guide and Reference* for additional information.

## Connectivity and Coexistence

### Replace RUMBA with PCOMM in DB2 Universal Database Version 6

| | | WIN |
|---|---|-----|

**Change:**

In Version 6, RUMBA is replaced by PCOMM on Windows NT, Windows 95, and Windows 98 only. RUMBA on Windows 3.1 will not be replaced.

**Symptom:**

None.

**Resolution:**

Windows 3.1 users will continue using RUMBA.

## Configuration Parameters

### Obsolete Database Configuration Parameters

| UNIX | OS/2 | WIN |
|------|------|-----|

**Change:**

The following database configuration parameters are obsolete with this version:
* DL_NUM_BACKUP (replaced by NUM_DB_BACKUP database configuration parameter)

**Resolution:**

Applications should be updated to not reference these parameters.

## DB2 Universal Database Version 5 Incompatibilities

This section focuses on the DB2 Universal Database Version 5 incompatibilities.

### System Catalog Views

#### System Catalog Views Column Changes

| UNIX | OS/2 | WIN |
|------|------|-----|

**Change:**

A variety of changes have been made to the system catalog views. This section will discuss the subset which could cause incompatibilities. To see a description of all changes (for example, new columns, new values in a column, and so on) refer to the *SQL Reference*.

**SYSCOLUMNS**
> **COLTYPE:**
>> Changed values: "FLOAT" to "DOUBLE"
>
> **NULLS:**
>> Changed values: "D" to "N". (Default flag now found in SYSCAT.COLUMNS.DEFAULT)
>
> **HIGH2KEY:**
>> Changed type: VARCHAR(16) to VARCHAR(33). Changed values: Values now stored in printable format rather than binary format
>
> **LOW2KEY:**
>> Changed type: VARCHAR(16) to VARCHAR(33). Changed values: Values are now stored in printable format rather than binary format for all datatypes.

**SYSINDEXES**
> **CLUSTERRATIO:**
>> Changed value: Value will always be -1 if the columns CLUSTERFACTOR and PAGE_FETCH_PAIRS are populated.

**SECT_INFO:**
> Changed type: LONG VARCHAR to BLOB(1M).

**HOST_VARS:**
> Changed type: LONG VARCHAR to BLOB(1M).

**ISOLATION:**
> Changed type: CHAR(1) to CHAR(2). Changed values: "R" to "RR", "S" to "RS", "C" to "CS", "U" to "UR".

**SYSRELS**

**RELNAME:**
> Changed type: CHAR(8) to VARCHAR(18).

**SYSSECTION**

**SECTION:**
> Changed type: VARCHAR(3900) to VARCHAR(3600)

**SYSSTMT**

**TEXT:** Changed type: VARCHAR(3900) to VARCHAR(3600)

**SYSTABLES**

**PACKED_DESC:**
> Changed type: LONG VARCHAR to BLOB(10M)

**VIEW_DESC:**
> Changed type: LONG VARCHAR to BLOB(32K)

**REL_DESC**
> Changed type: LONG VARCHAR to BLOB(32K)

**FID** Will no longer uniquely identify a table on its own. Must be used with TID to uniquely identify a table.

**SYSVIEWS**

**CHECK:**
> Changed values: "Y" to "L".

**TEXT:** Changed type: VARCHAR(3900) to VARCHAR(3600). Contains the full text of the create view statement (including the CREATE VIEW). In Version 1, only the select portion was shown.

**Symptom:**

A variety of symptoms could occur.

If you have an application which has a qualified search on a field that has had a value change (for example, ISOLATION in SYSIBM.SYSPLAN) this will cause your application to react differently than you would want.

If you have an application which accesses some field where the field type or size has changed (such as SECTION in SYSIBM.SYSSECTION), you could

retrieve an incomplete set of data, too much data, or have the wrong type defined in your application to represent the data type of the table column.

**Resolution:**

If you use the SYSIBM tables for application processing or anything else, you must review the changes listed above to decide whether or not you are affected and what the appropriate action to correct the situation is. You may need to refer to the *SQL Reference* to understand what the new columns, new values for columns and other changes that were made to these tables.

If you need a rough approximation of the degree of clustering, select both CLUSTERRATIO and CLUSTERFACTOR and choose the "greater" one.

## Application Programming

### External Table Functions in DB2 Universal Database Version 5

| UNIX | OS/2 | WIN | Extended |
|------|------|-----|----------|

**Change:**

In Version 5.2, the user has been given explicit control over scratchpad duration.

For User Defined Functions (UDFs) which are table functions:
- FINAL CALL is now optional for the CREATE FUNCTION statement for table functions, as it is for scalar functions. NO FINAL CALL (the default) may thus be specified for table functions.
- When writing a table function which has a scratchpad, you can elect to have the scratchpad be initialized for each OPEN call to the function, by specifying NO FINAL CALL, in which case the table function is expected to release resources on each CLOSE call.

  Or, you can elect to have the scratchpad content be preserved across OPEN calls, by specifying FINAL CALL. If you do this, you need not release resources during the CLOSE call processing, because a FINAL call will be made to the table function at end-of-statement. The FINAL CALL is new with this change, as is the balancing FIRST call which takes place before the first OPEN call. These two new call types occur only on table functions which specify FINAL CALL.
- For Java table functions, there is a change to the execution model based on the new rules declared above. Refer to the *Application Development Guide* for details on the changes to table function processing.

**Symptom:**

The introduction of new call types may generate unexpected calls and may result in bad results being returned from the function depending on how the UDF is written to examine the call-type arguments.

**Resolution:**

Customers who have existing table functions should examine them knowing the changes introduced by the new call types and the new execution model. The following changes should be considered:

- If you depend on the new scratchpad rules implied by FINAL CALL then you may have to change your UDF to add the new call types, because with FINAL CALL specified, the new calls will be made to the UDF. Also, any acquired resources should be freed when the FINAL call is handled by the table function.

- If you wish to use the rules implied by NO FINAL CALL, you are advised to change and rerun your CREATE FUNCTION statement with NO FINAL CALL instead of FINAL CALL, so that your UDF will not get the new calls.

- If you have a Java table function, then you will have to change your function to conform to the new execution models. For either execution model (depending on whether you choose FINAL CALL or NO FINAL CALL), you will note that additional calls are made to the UDF method. This gives additional power to the UDF writer, but means that the writer must distinguish the calls. Use of the new `getCallType` method in the parent class included by DB2 provides this capability.

Refer to the *Application Development Guide* for detailed information on managing these changes.

### NS, NW and NX Locks

| UNIX | OS/2 | WIN | DB2 PE |
|------|------|-----|--------|

**Change:**

Due to the addition of NS and NX lock modes in DB2 Version 5, there is a difference in the behavior of index scans with isolation level Cursor Stability (CS) or Read Stability (RS).

**Symptom:**

In DB2 Version 5, an index scan with isolation level, CS or RS, will not see an uncommitted delete of a row that is within the scanned range. In DB2 Version 2, the scanner would not see an uncommitted delete of a row that was at the end of the scanned range. However, if the deleted row was within the range, the scanner would remain in a lock wait until the delete was committed or rolled back.

For example in DB2 Version 5, the following can occur with an index on Column A:

| Sequence | Application 1 | Application 2 |
|---|---|---|
| 1 | delete from t1 where a=3 | |
| 2 | | select a from t1 where a>1 and a<5 |

```
  A
------------
         2
         4
         5
```

| Sequence | Application 1 | Application 2 |
|---|---|---|
| 3 | rollback | |
| 4 | | select a from t1 where a>1 and a<5 |

```
  A
------------
         2
         3
         4
         5
```

The same scenario in previous versions of DB2 would result in application 2 being in lock wait until Application 1 committed or rolled back.

**Resolution:**

There is no resolution as this is an enhancement to isolation level Cursor Stability or Read Stability.

**Symptom:** The previous example showed what occurs with an uncommitted deletion. A similar situation could also arise when inserting new values.

For example, you could have a scenario where you are scanning a table using an index on a column and looking for a value greater than or equal to two, but less than or equal to six, while using an isolation level of RS. The existing values that qualify in this example are two, four, and six. Then another user inserts five. An NS lock is obtained on columns returning two, four, and six; and the NW lock attempt on the column containing six succeeds, so the insertion of five is not blocked by the scan.

In Version 2, an S lock would be obtained on columns with the values two, four, and six; and the attempt to get an X lock on the column returning six would wait. The insert of five would wait for the S lock on six to be released.

**Resolution:** In general, since more concurrency is supported in Version 5, applications built with a previous version of DB2 that were created with dependencies on some lock waiting may require modification.

### DB2 Call Level Interface (DB2 CLI) Defaults

| UNIX | OS/2 | WIN |
|------|------|-----|

**Change:**

When moving from Version 2 to Version 5, the default values for
**AUTOCOMMIT** and **CURSORHOLD** have changed. Both AUTOCOMMIT
and CURSORHOLD will now default to ON.

**Symptom:**

If an application was written assuming that AUTOCOMMIT was OFF or that
WITH HOLD semantics was NOT used for cursors, then these default
changes could cause the application to fail.

**Resolution:**

Add one or both of the following two lines to your DB2CLI.INI file.
- AUTOCOMMIT = 0
- CURSORHOLD = 0

### DB2 CLI SQLSTATEs

| UNIX | OS/2 | WIN |
|------|------|-----|

**Change:**

When moving from Version 2 to Version 5, a more explicit set of SQLSTATEs
(in the S1090 to S1110 range) has replaced the generic SQLSTATE S1009.

**Symptom:**

SQLSTATE values returned to the application calling DB2 CLI APIs have
changed.

**Resolution:**

Update your application to check for the new SQLSTATEs. Refer to the
*Message Reference* for a complete list of these SQLSTATEs.

### DB2 CLI/ODBC Configuration Keyword Defaults

| UNIX | OS/2 | WIN |
|------|------|-----|

**Change:**

When moving from Version 2 to Version 5, the default value for the CLI/ODBC configuration keyword DEFERREDPREPARE has changed. In DB2 CLI Version 5 deferred prepare is now on by default.

**Symptom:**

Applications that rely on the prepare to be executed as soon as it is issued will not function as expected. In particular, the row and cost estimates normally returned in the SQLERRD(3) and SQLERRD(4) of the SQLCA of a prepare statement may become zeros. The application will not be able to use this information to decide whether or not to continue the execution of the SQL statement.

**Resolution:**

Add the following line to your *db2cli.ini* file:

```
DEFERREDPREPARE = 0
```

### Obsolete DB2 CLI/ODBC Configuration Keywords in DB2 Universal Database Version 5

| UNIX | OS/2 | WIN |
|------|------|-----|

**Change:**

When moving from version to version, you can change the behavior of the DB2 CLI/ODBC driver by specifying a set of optional keywords in the *db2cli.ini* file.

In Version 5, the AUTOCOMMIT keyword became obsolete.

**Symptom:**

These keywords will be ignored if they still exist. You may notice behavioral changes based on the removal of these settings.

**Resolution:**

You will need to review the new list of valid parameters to decide what the appropriate keywords and settings are for your environment. See the *CLI Guide and Reference* for information on these keywords.

### DB2 CLI SQLSTATEs

| UNIX | OS/2 | WIN |
|------|------|-----|

**Change:**

When moving from Version 2 to Version 5, the category of SQLSTATEs that started with S1 in DB2 CLI Version 2 have been renamed to begin with HY in Version 5. For example, the SQLSTATE S1010 is now HY010.

**Symptom:**

SQLSTATE values returned to the application calling DB2 CLI APIs have changed.

**Resolution:**

Applications should be updated to expect the new HY class of SQLSTATEs. Alternatively, the environment attribute SQL_ATTR_ODBC_VERSION can be set to SQL_OV_ODBC2 using the DB2 CLI function `SQLSetEnvAttr()`. The DB2 CLI/ODBC driver will then return the S1 class of SQLSTATEs.

### Stored Procedure Catalog Table

| UNIX | OS/2 | WIN |
|------|------|-----|
|      |      |     |

**Change:**

When moving from Version 2 to Version 5, Version 5 now has 2 system catalog views used to store information about all the stored procedures on the server (SYSCAT.PROCEDURES and SYSCAT.PROCPARMS). These replace the Version 2 pseudo catalog table for stored procedures

**Symptom:**

By default the server will look in the new system catalog views for information about stored procedures, not the older pseudo catalog table. DB2 CLI functions such as `SQLProcedureColumns()` and `SQLProcedures()` will therefore not return the appropriate information.

**Resolution:**

Register the stored procedures using the CREATE PROCEDURE SQL command. See the *SQL Reference* for more information. Alternatively, the DB2 CLI/ODBC configuration keyword PATCH1 can be set to 262144 to force the DB2 CLI/ODBC driver to use the pseudo catalog table as it did in Version 2.

### Change to SMALLINT Constants

| UNIX | OS/2 | WIN |
|------|------|-----|
|      |      |     |

**Change:**

Integer constants in the range -32,768 to 32,767 are now treated as INTEGER types, rather than SMALLINT. This resolves an incompatibility with IBM SQL, as well as simplifying the rules for determining literal types.

It is also worth mentioning that the smallest INTEGER constant in Version 1 (-2 147 483 648) is a DECIMAL constant with a precision of 10 and a scale of 0 in Version 5.

Further, 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 large INTEGER values). The INTEGER constant -2 147 483 648 is a BIGINT, not a DECIMAL (as it was before Version 5.2).

In general, if an integer constant is outside the range of a large integer and within the range of a BIGINT, then it is a BIGINT. If it is too big for a BIGINT, then it is a DECIMAL.

**Symptom:**

This affects the result precision and scale of decimal operations. (Which impacts, for example, the print width of decimal fields.)

**Resolution:**

There is no resolution. This change in handling integers results in an increase in precision.

### Down-level Client and Distinct Types Sourced on BIGINT

| UNIX | OS/2 | WIN | DB2 PE |
|------|------|-----|--------|
|      |      |     |        |

**Change:**

A distinct type based on BIGINT in a Version 5.2 server is reported in a DESCRIBE to a down-level client as a DECIMAL(19,0) instead of as a BIGINT which is not supported by the client. This data type cannot be implicitly cast on assignment to the distinct type on which it is based. This is different than other situations where the client perceives a distinct type as a built-in data type and is able to assign host variables of the built-in type to columns of the associated distinct type.

**Symptom:**

An SQLCODE of -408 (SQLSTATE 42821) is returned when using a data type of DECIMAL(19,0) for the host variable (or parameter marker) assigned to the distinct type value that was described to the down-level client as DECIMAL(19,0).

**Resolution:**

The database should include a function that will cast a DECIMAL(19,0) to the distinct type. This can be defined as a sourced function based on the function that casts a BIGINT to the distinct type. The application (at the client) must then explicitly apply this function to the DECIMAL(19,0) host variable (or parameter marker) in the INSERT or UPDATE statement.

For example, if the distinct type sourced on BIGINT is called DT1, then updating the column C1 of type DT1 would require the following sourced function to be defined:

```
CREATE FUNCTION DT1(DECIMAL(19,0)) RETURNS DT1 SOURCE DT1(BININT);
```

And then the update statement in the application would be:

```
UPDATE table SET c1=DT1(:dechv1);
```

## Maximum Number of Sections in a Package

| UNIX | OS/2 | WIN |
|------|------|-----|
|      |      |     |

**Change:**

The limit for the maximum number of sections in a package has changed from 400 to whatever the storage allows. This limit used to be hard-coded at 400, but now depends on the type of SQL statements in the program. As a result of this change, the constant for the maximum number of SQL statements has been removed from the common include files *sql.h*, *sql.cbl*, and *sql.f*.

**Symptom:** If an application program references the following constants, it will not compile successfully in Version 5:
- SQL_MAXSTMTS (in *sql.h*)
- SQL-MAXSTMTS (in *sql.cbl*)
- SQL_MAXSTMTS (in *sql.f*)

**Resolution:**

Remove references to these constants or define them directly within your application.

## Bind Options

| UNIX | OS/2 | WIN |
|------|------|-----|
|      |      |     |

**Change:**

The new SQLWARN bind option has a default value of 'YES'.

**Symptom:**

By default, positive SQLCODEs may now be returned on DESCRIBE, PREPARE, and EXECUTE IMMEDIATE requests which were previously not returned. (For instance, a SQLCODE of +236 may be returned).

**Resolution:**

Rebind with SQLWARN NO if your application cannot tolerate positive SQLCODEs or treats them as errors.

### Supported Level of JDBC

| UNIX | OS/2 | WIN |
|------|------|-----|

**Change:**

The supported level of the JDBC (Java Support) API has changed. DB2 Version 5 provides a driver for JDBC 1.1 instead of JDBC 1.0, which came with DB2 Version 2.1.2.

**Symptom:**

Compiled JDBC 1.0 clients fail when executed directly as a DB2 Version 5 client. Old Java classes are not found.

**Resolution:**

To continue using JDBC 1.0 clients, run them on a DB2 Version 2.1.2 client, with a remote DB2 Version 5 server. Modify the client source code to upgrade to the JDBC 1.1 API. Run the JDBC 1.1 clients on a Java Development Kit Version 1.1-compatible virtual machine.

### Java Runtime Environment

| UNIX | OS/2 | WIN |
|------|------|-----|

**Change:**

The level of the Java runtime environment required for Java stored procedures, user-defined functions, and JDBC clients has changed in DB2 Version 5.

**Symptom:**

The JDBC DLL will not load when JDBC 1.1 clients are run. Java stored procedures and UDFs will fail.

**Resolution:**

Install a compatible Java 1.1.2 or Java 1.2 runtime environment at the client and server. At the server, set the `jdk11_path` configuration parameter.

## SQL

### Updating Partitioning Key Columns

| UNIX | OS/2 | WIN | DB2 PE |
| --- | --- | --- | --- |

**Change:**

In DB2 PE Version 1.2, partitioning key columns could be updated if the table was in a single-node nodegroup. In DB2 Version 5, partitioning key columns can be updated if the table is in a table space in a single-node nodegroup, and there is no partitioning key defined.

**Symptom:**

An update statement fails with SQL270N (SQLCODE -270, SQLSTATE 42997) with reason code 2. The same error is returned if the table is in a table space in a single or multiple node nodegroup.

**Resolution:**

If the table is in a table space in a single node nodegroup, then use the ALTER TABLE statement to DROP the partitioning key. As with DB2 PE Version 1.2, if the table is in a table space in a multiple node nodegroup, the nodegroup must be changed to a single-node nodegroup and REDISTRIBUTE NODEGROUP must be issued before attempting to update partitioning key columns.

### Column NGNAME

| UNIX | OS/2 | WIN | DB2 PE |
| --- | --- | --- | --- |

**Change:**

In DB2 PE Version 1.2, a table was directly associated with a nodegroup. In DB2 Version 5, a table is in a table space, which is within a nodegroup. Since there is no longer a direct relationship with a nodegroup, there is no need for a column, named NGNAME in the SYSIBM.SYSTABLES catalog table.

**Symptom:**

An SQL statement that refers to the NGNAME column from
SYSIBM.SYSTABLES catalog table will return an SQLCODE of −206
(SQLSTATE 42703).

**Resolution:**

Remove the column NGNAME from the SQL statement. To determine the
nodegroup name for the table, refer to NGNAME in the row of
SYSCAT.TABLESPACES catalog view, that relates to the table space in which
the table is stored.

### Node Number Temporary Space Usage

| UNIX | OS/2 | WIN | DB2 PE |
|------|------|-----|--------|
|      |      |     |        |

**Change:**

When moving from Parallel Edition Version 1. 2 to DB2 Universal Database
Version 5, and using a temporary table that requires row identifiers, the
amount of space needed is increased to include the node number. The space
limit for temporary tables is 4 005 bytes. If temporary tables are close to the
4 005 byte limit, any further increase can exceed this limit.

**Symptom:**

There are two possible symptoms of this change.
- An SQL statement may fail to compile and return an SQLCODE of
  SQL0670N (SQLSTATE 54010).
- The temporary table is not used, which may affect the performance of the
  query.

**Resolution:**

You should review and use the directions in the Actions section of the
message details for SQL0670N to fix the error.

### Authorities for Create and Drop Nodegroups

| UNIX | OS/2 | WIN | DB2 PE |
|------|------|-----|--------|
|      |      |     |        |

**Change:**

In Version 5, the authorization required for creating or dropping a nodegroup
changed from SYSADM or DBADM to SYSADM or SYSCTRL. A user ID with
DBADM authority cannot create, alter, or drop nodegroups.

**Symptom:**

A user ID, with DBADM authority, issuing a CREATE NODEGROUP or
DROP NODEGROUP statement will receive an SQL00551N (SQLSTATE
42501).

**Resolution:**

Issue the statement using a user ID that has SYSADM or SYSCTRL authority.
For your convenience, you may wish to include the user ID in the SYSCTRL
group. Refer to the *Administration Guide* for further information.

### Target Map in REDISTRIBUTE NODEGROUP

| UNIX | OS/2 | WIN | DB2 PE |
|------|------|-----|--------|

**Change:**

Beginning in Version 5, the specification of a target map in the
REDISTRIBUTE NODEGROUP command or API no longer causes database
partitions to be implicitly added or dropped from the node group. This means
that the target map cannot include nodes that are not defined to the node
group. An undefined node that is included in the target map file will cause an
error to be returned. A database partition, which has been defined to the node
group, can be excluded from the target map file and will not appear in the
partition map.

**Symptom:**

If a node is included in the target map file and was not defined to the node
group, the REDISTRIBUTE NODEGROUP command will return an
SQLCODE–6053 with a reason code 6.

**Resolution:**

Before issuing the REDISTRIBUTE NODEGROUP command, add the database
partition to the node group, using the ALTER NODEGROUP statement. You
can also drop the node from the node group using the ALTER NODEGROUP
statement, either before or after issuing the REDISTRIBUTE NODEGROUP
command. Refer to the *SQL Reference* for further information on the ALTER
NODEGROUP statement.

### Node Group for Create Table

| UNIX | OS/2 | WIN | DB2 PE |
|------|------|-----|--------|

**Change:**

In DB2 PE Version 1, a table was directly associated with a node group. In DB2 Version 5, a table is in a table space within a node group. When a user issues a CREATE TABLE statement, the name following the IN keyword is a table space name, not a node group name. The default table space selected may not be defined in the IBMDEFAULTGROUP node group, which was the default node group in DB2 PE Version 1.

**Symptom:**

If you use existing CREATE TABLE statements from DB2 PE Version 1, they may fail with an SQLCODE of SQL0204N (SQLSTATE 42704), with the name specified following the IN keyword in the message. This will occur if a table space with the same name as the node group has not been automatically created during database migration.

If you are using CREATE TABLE statements that do not specify the IN keyword, the table space selected, by default, may not be using the node group, IBMDEFAULTGROUP, and will not include data on all the database nodes. You can check the partition map for the table to confirm this.

**Resolution:**

Ensure that any name specified following the IN keyword on the CREATE TABLE statement is the name of a defined table space. For existing statements, you could set up a table space for each node group with the same name.

To ensure that tables default to the IBMDEFAULTGROUP for all users, define a table space called IBMDEFAULTGROUP, defined in the node group, IBMDEFAULTGROUP. This ensures that tables created by any users will default to use this table space.

**Note:** This is done automatically during database migration from DB2 PE Version 1 to DB2 Version 5.

### Revoking CONTROL on Tables or Views

| UNIX | OS/2 | WIN | DB2 PE |
|------|------|-----|--------|

**Change:**

A user can grant privileges on a table or view using the CONTROL privilege. In DB2 Version 5, the WITH GRANT OPTION provides a mechanism to determine a user's authorization to grant privileges on tables and views to other users. This mechanism is used in place of CONTROL to determine whether a user may grant privileges to others. When CONTROL is revoked, users will continue to be able to grant privileges to others.

**Symptom:**

A user can still grant privileges on tables or views, following the revocation of CONTROL privilege.

**Resolution:**

If a user should no longer be authorized to grant privileges on tables or views to others, revoke all privileges on the table or view and grant only those required.

### High Level Qualifiers for Objects in DB2 Version 5

| UNIX | OS/2 | WIN | DB2 PE |
|------|------|-----|--------|

**Change:**

In DB2 PE Version 1, users would create a table, view, index or package with any schema name or qualifier with the exception of SYSIBM. This differs from other IBM database products and is not compliant with SQL92. In DB2 Version 5, there are limits of the schema names that you can use.

- The schema names for tables, views, indexes, and packages cannot be SYSIBM, SYSCAT, SYSSTAT, OR SYSFUN.

  **Note:** The schema names for all other objects must not start with SYS.

- Each schema is an object defined in the database catalog.

  Users require IMPLICIT_SCHEMA authority to implicitly create a schema. Once a schema is created, specific privileges allow users to create objects (CREATEIN privilege), drop any object in the schema (DROPIN privilege), or alter (comment on) any object in the schema (ALTERIN privilege). The change to supporting schemas, as objects with privileges, has resulted in changes to privileges associated with various statements.
  - For creating objects in an existing schemas, you must have CREATEIN privilege.
  - For creating objects in a schema that does not exist, you must have IMPLICIT_SCHEMA authority.
  - For dropping objects in a schema, you must be the definer of the object, have CONTROL privilege, or have DROPIN privilege on the schema.
  - For altering, including commenting on, objects in a schema you must be the definer of the object, have CONTROL privilege, or have ALTERIN privilege on the schema.

  **Note:** For altering or commenting on a table, the ALTER privilege on the table is also valid.

**Symptom:**

If you create an object with an invalid schema name, the CREATE statement returns an SQLCODE of SQL0553N. This message indicates that the object cannot be created with the schema name.

If a CREATE, ALTER, COMMENT ON or DROP statement returns an SQLCODE of SQL0551N, you did not have the necessary privilege. This may be the result of schema-related privileges and could indicate that:
- The object cannot be created because the schema does exist and you do not have the IMPLICIT_SCHEMA authority.
- The object cannot be created because the schema does not exist and you do not have the CREATEIN privilege.
- The object cannot be dropped because another user created the object and you do not have the DROPIN privilege.
- The object cannot be altered (commented on) because another user created the object and you do not have ALTERIN privilege.

**Resolution:**

Depending on the symptom:
- Do not create schema names with SYS.
- If a user can create a table, view, index or package, grant the necessary authority or privilege using the GRANT (Database Authorities) statement for IMPLICIT_SCHEMA authority, or the GRANT (Schema Privileges) statement for CREATEIN, DROPIN or ALTERIN privilege on the schema. A user with DBADM authority must first create the schema.

## Database Security and Tuning

### Creating Databases Using Clients in DB2 Universal Database Version 5

| UNIX | OS/2 | WIN | Extended |
|------|------|-----|----------|

**Change:**

The method used by clients to create a database.

**Symptom:**

Using a down-level client (from a previous release than that of the server) to create a database will result in errors.

**Resolution:**

When using a client to create a database, only use clients at the same level as that of the server.

## Utilities and Tools

### LOAD TERMINATE

|  |  |  | DB2 PE |
|---|---|---|---|

**Change:**

The LOAD TERMINATE command has a different function in DB2 UDB than it did in DB2 Parallel Edition Version 1.x. In DB2 Parallel Edition, you could use LOAD TERMINATE if an error occurred during the load operation to ensure that the table data was consistent. In DB2 Universal Database Version 5 however, if you use LOAD TERMINATE, the table space is moved into the recovery pending state. (When the table space is in the recovery pending state, you must either restore the table space or drop it.)

**Symptom:**

Instead of being placed in a consistent state, the table space is placed in a recovery pending state.

**Resolution:**

Instead of using LOAD TERMINATE to clean up after a failed load operation, you should use LOAD RESTART or LOAD REPLACE. You also have the option of dropping and re-creating the table space.

### REORG - Alternate Path Option

| UNIX | OS/2 |  |
|---|---|---|

**Change:**

When moving from Version 2 to Version 5, the REORG command and API no longer support an "alternate path" as a work area, but rather support the name of a table space to be used as a work area. APIs and commands will not fail, however, this option will be ignored.

**Symptom:**  REORG invocations from downlevel clients will ignore the alternate work path and arbitrarily choose a temporary table space to use as a work area.

Another symptom is you may run out of disk space.

**Resolution:**

Your applications will continue to function, but you should consider
upgrading to the DB2 Version 5 calls which contain valid options.

## Connectivity and Coexistence

There are no incompatibilities for this area.

## Configuration Parameters

### LOGFILSIZ

| OS/2 | WIN | UNIX | DB2 PE |
|------|-----|------|--------|

**Change:**

The data type of this database configuration parameter has changed from
being an unsigned 2-byte integer to an unsigned 4-byte integer. A new token
has been added for the configuration APIs indicating a 4-byte integer.
  For DB2 Version 5, the token is SQLF_DBTN_LOGFIL_SIZ
  For DB2 Version 2, the token is SQLF_DBTN_LOGFILSIZ

The configuration API will still recognize the Version 2 token, but the full
range of values of this parameter is greater than what is supported by a
2-byte integer.

**Symptom:**

Existing applications will continue to work using the configuration API or via
REXX, but the results might be unpredictable because of the larger range in
DB2 Version 5.

**Resolution:**

Re-code the application or REXX script to use the new token. For users of the
Command Line Processor or the Control Center, this change in the token
would not affect your applications.

### BUFFPAGE and Multiple Buffer Pools

| UNIX | OS/2 | WIN | DB2 PE |
|------|------|-----|--------|

**Change:**

Before Version 5, each database had one buffer pool, which was created when
the database was created. You could change the size of the buffer pool using
the *buffpage* parameter. In DB2 Version 5 and later, each database can have
multiple buffer pools. You can create additional buffer pools of possibly
different page sizes or change the size of a buffer pool through the CREATE

BUFFERPOOL or ALTER BUFFERPOOL statements or through the Control Center using the appropriate command.

If the buffer pool size is specified to be −1, then the value of the database configuration parameter is used as the size of the buffer pool.

**Note:** When the BUFFPAGE database configuration parameter is updated, you will receive an SQLCODE SQL1482W warning.

**Symptom:**

In DB2 Version 5, a new or migrated database has a default buffer pool. For a new database created in DB2 Version 5, the size of the default buffer pool is determined by the operating system. For a migrated database, the size of the buffer pool is set to −1, which then refers to the *buffpage* configuration parameter.

**Resolution:**

To resolve this problem, you will need to do the following:
1. For a new database created in DB2 Version 5, you may change the size of the buffer pool using the ALTER BUFFERPOOL statement.
2. Following the creation or migration of a database, you can then create additional buffer pools for the database using the CREATE BUFFERPOOL statement.

### NEWLOGPATH

| OS/2 | WIN | UNIX | DB2 PE |
|------|-----|------|--------|

**Change:**

In DB2 Version 5, in a partitioned database, the node number is appended to the path in the form `path_name \NODExxxx` (`path_name /NODExxxx` on UNIX-based systems), where xxxx is the 4 digit node number. This maintains the uniqueness of the path across the database partitions.

**Symptom:**

When updating the NEWLOGPATH configuration parameter, the node number is automatically appended to the path name. This may result in path names that are too long (greater than 242 characters), and the configuration parameter update may fail.

**Resolution:**

Be aware that the log files will reside in the path that includes the node numbering designation. If the configuration parameter update failed, ensure that the path length, including the node number designation, is less than or equal to 242 characters.

## MULTIPAGE_ALLOC

|  |  |  | DB2 PE |
|---|---|---|---|

**Change:**

In DB2 PE Version 1.2, this database configuration parameter was known as MULTIPGAL and the data type of this database configuration parameter was an unsigned 1-byte integer. In DB2 Version 5, the data type of this parameter is an unsigned 2-byte integer, using a new token.

For DB2 Version 5, the token is SQLF_DBTN_MULTIPAGE_ALLOC
For DB2 PE Version 1, the token is SQLF_DBTN_MULTIPGAL

**Symptom:**

Existing applications will continue to work using either the SQLF_DBTN_MULTIPGAL or the SQLF_DBNR_MULTIPAGE_ALLOC tokens.

**Resolution:**

While the configuration APIs support both tokens, applications should be updated to use the new tokens.

## LOCKLIST

| UNIX | OS/2 |  |
|---|---|---|

**Change:**

In DB2 Version 5, the size of a lock request block has been changed to 36 bytes. As a result, fewer lock request blocks will fit in the configured amount of space allocated for the lock list.

**Symptom:**

This may result in more frequent lock escalations.

**Resolution:**

You should increase the setting of the LOCKLIST configuration parameter accordingly.

# Appendix K. Explain Tables and Definitions

The Explain tables capture access plans when the Explain facility is activated. The following Explain tables and definitions are described in this section:

The Explain tables must be created before Explain can be invoked. To create them, use the sample command line processor input script provided in the EXPLAIN.DDL file located in the 'misc' subdirectory of the 'sqllib' directory. Connect to the database where the Explain tables are required. Then issue the command: `db2 -tf EXPLAIN.DDL` and the tables will be created. See "Table Definitions for Explain Tables" on page 1153 for more information.

The population of the Explain tables by the Explain facility will neither activate any triggers nor activate any 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.

For more details on the Explain facility, see the *Administration Guide*.

**Legend for the Explain Tables:**

| Heading | Explanation |
|---|---|
| Column name | Name of the column |
| Data Type | Data type of the column |
| Nullable? | Yes: Nulls are permitted |
| | No: Nulls are not permitted |
| Key? | PK: Column is part of a primary key |
| | FK: Column is part of a foreign key |
| Description | Description of the column |

## EXPLAIN_ARGUMENT Table

The EXPLAIN_ARGUMENT table represents the unique characteristics for each individual operator, if there are any.

For the definition of this table, see "EXPLAIN_ARGUMENT Table Definition" on page 1154.

*Table 112. EXPLAIN_ARGUMENT Table*

| 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. |
| 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(1M) | 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 113. ARGUMENT_TYPE and ARGUMENT_VALUE Column Values*

| ARGUMENT_TYPE Value | Possible ARGUMENT_VALUE Values | Description |
|---|---|---|
| AGGMODE | COMPLETE PARTIAL INTERMEDIATE FINAL | Partial aggregation indicators. |
| BITFLTR | TRUE FALSE | Hash Join will use a bit filter to enhance performance. |
| CSETEMP | TRUE FALSE | Temporary Table over Common Subexpression Flag. |
| DIRECT | TRUE | Direct fetch indicator. |

*Table 113. ARGUMENT_TYPE and ARGUMENT_VALUE Column Values (continued)*

| ARGUMENT_TYPE Value | Possible ARGUMENT_VALUE Values | Description |
|---|---|---|
| DUPLWARN | TRUE<br>FALSE | Duplicates Warning flag. |
| EARLYOUT | TRUE<br>FALSE | Early out indicator. |
| ENVVAR | Each row of this type will contain:<br>• Environment variable name<br>• Environment variable value | Environment variable affecting the optimizer |
| FETCHMAX | IGNORE<br>INTEGER | Override value for MAXPAGES argument on FETCH operator. |
| GROUPBYC | TRUE<br>FALSE | Whether Group By columns were provided. |
| GROUPBYN | Integer | Number of comparison columns. |
| GROUPBYR | Each row of this type will contain:<br>• Ordinal value of column in group by clause (followed by a colon and a space)<br>• Name of Column | Group By requirement. |
| INNERCOL | Each row of this type will contain:<br>• Ordinal value of column in order (followed by a colon and a space)<br>• Name of Column<br>• Order Value<br><br>   **(A)**     Ascending<br><br>   **(D)**     Descending | Inner order columns. |
| ISCANMAX | IGNORE<br>INTEGER | Override value for MAXPAGES argument on ISCAN operator. |
| JN_INPUT | INNER<br>OUTER | Indicates if operator is the operator feeding the inner or outer of a join. |
| LISTENER | TRUE<br>FALSE | Listener Table Queue indicator. |
| MAXPAGES | ALL<br>NONE<br>INTEGER | Maximum pages expected for Prefetch. |
| MAXRIDS | NONE<br>INTEGER | Maximum Row Identifiers to be included in each list prefetch request. |
| NUMROWS | INTEGER | Number of rows expected to be sorted. |
| ONEFETCH | TRUE<br>FALSE | One Fetch indicator. |

## Explain Tables

*Table 113. ARGUMENT_TYPE and ARGUMENT_VALUE Column Values (continued)*

| ARGUMENT_TYPE Value | Possible ARGUMENT_VALUE Values | Description |
|---|---|---|
| OUTERCOL | Each row of this type will contain:<br>• Ordinal value of column in order (followed by a colon and a space)<br>• Name of Column<br>• Order Value<br><br>**(A)**     Ascending<br><br>**(D)**     Descending | Outer order columns. |
| OUTERJN | LEFT<br>RIGHT | Outer join indicator. |
| PARTCOLS | Name of Column | Partitioning columns for operator. |
| PREFETCH | LIST<br>NONE<br>SEQUENTIAL | Type of Prefetch Eligible. |
| RMTQTEXT | Query text | Remote Query Text |
| ROWLOCK | EXCLUSIVE<br>NONE<br>REUSE<br>SHARE<br>SHORT (INSTANT) SHARE<br>UPDATE | Row Lock Intent. |
| ROWWIDTH | INTEGER | Width of row to be sorted. *** |
| SCANDIR | FORWARD<br>REVERSE | Scan Direction. |
| SCANGRAN | INTEGER | Intra-partition parallelism, granularity of the intra-partition parallel scan, expressed in SCANUNITs. |
| SCANTYPE | LOCAL PARALLEL | intra-partition parallelism, Index or Table scan. |
| SCANUNIT | ROW<br>PAGE | Intra-partition parallelism, scan granularity unit. |
| SERVER | Remote server | Remote server |
| SHARED | TRUE | Intra-partition parallelism, shared TEMP indicator. |
| SLOWMAT | TRUE<br>FALSE | Slow Materialization flag. |
| SNGLPROD | TRUE<br>FALSE | Intra-partition parallelism sort or temp produced by a single agent. |

*Table 113. ARGUMENT_TYPE and ARGUMENT_VALUE Column Values  (continued)*

| ARGUMENT_TYPE Value | Possible ARGUMENT_VALUE Values | Description |
|---|---|---|
| SORTKEY | Each row of this type will contain:<br>• Ordinal value of column in key (followed by a colon and a space)<br>• Name of Column<br>• Order Value<br><br>  **(A)**     Ascending<br><br>  **(D)**     Descending | Sort key columns. |
| SORTTYPE | PARTITIONED<br>SHARED<br>ROUND  ROBIN<br>REPLICATED | Intra-partition parallelism, sort type. |
| TABLOCK | EXCLUSIVE<br>INTENT  EXCLUSIVE<br>INTENT  NONE<br>INTENT  SHARE<br>REUSE<br>SHARE<br>SHARE  INTENT  EXCLUSIVE<br>SUPER  EXCLUSIVE<br>UPDATE | Table Lock Intent. |
| TQDEGREE | INTEGER | intra-partition parallelism, number of subagents accessing Table Queue. |
| TQMERGE | TRUE<br>FALSE | Merging (sorted) Table Queue indicator. |
| TQREAD | READ  AHEAD<br>STEPPING<br>SUBQUERY  STEPPING | Table Queue reading property. |
| TQSEND | BROADCAST<br>DIRECTED<br>SCATTER<br>SUBQUERY  DIRECTED | Table Queue send property. |
| TQTYPE | LOCAL | Intra-partition parallelism, Table Queue. |
| TRUNCSRT | TRUE | Truncated sort (limits number of rows produced). |
| UNIQUE | TRUE<br>FALSE | Uniqueness indicator. |
| UNIQKEY | Each row of this type will contain:<br>• Ordinal value of column in key (followed by a colon and a space)<br>• Name of Column | Unique key columns. |
| VOLATILE | TRUE | Volatile table |

## 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.

For the definition of this table, see "EXPLAIN_INSTANCE Table Definition" on page 1155.

*Table 114. EXPLAIN_INSTANCE Table*

| 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. |
| EXPLAIN_OPTION | CHAR(1) | No | No | Indicates what Explain Information was requested for this request.<br><br>Possible values are:<br>**P**     PLAN SELECTION |
| SNAPSHOT_TAKEN | CHAR(1) | No | No | Indicates whether an Explain Snapshot was taken for this request.<br><br>Possible values are:<br>**Y**     Yes, an Explain Snapshot(s) was taken and stored in the EXPLAIN_STATEMENT table. Regular Explain information was also captured.<br>**N**     No Explain Snapshot was taken. Regular Explain information was captured.<br>**O**     Only an Explain Snapshot was taken. Regular Explain information was not captured. |
| DB2_VERSION | CHAR(7) | No | No | Product release number for DB2 Universal Database which processed this explain request. Format is vv.rr.m, where:<br>**vv**     Version Number<br>**rr**     Release Number<br>**m**     Maintenance Release Number |

*Table 114. EXPLAIN_INSTANCE Table  (continued)*

| Column Name | Data Type | Nullable? | Key? | Description |
|---|---|---|---|---|
| SQL_TYPE | CHAR(1) | No | No | Indicates whether the Explain Instance was for static or dynamic SQL.<br><br>Possible values are:<br>**S**       Static SQL<br>**D**       Dynamic SQL |
| 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. For more information, see the BLOCK column in SYSCAT.PACKAGES.<br><br>Possible values are:<br>**N**       No Blocking<br>**U**       Block Unambiguous Cursors<br>**B**       Block All Cursors |
| ISOLATION | CHAR(2) | No | No | Indicates what type of isolation was used when compiling the SQL statements. For more information, see the ISOLATION column in SYSCAT.PACKAGES.<br><br>Possible values are:<br>**RR**       Repeatable Read<br>**RS**       Read Stability<br>**CS**       Cursor Stability<br>**UR**       Uncommitted Read |
| BUFFPAGE | INTEGER | No | No | Contains the value of the BUFFPAGE database configuration setting at the time of the Explain invocation. |
| 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 setting at the time of the Explain invocation. |
| LOCKLIST | INTEGER | No | No | Contains the value of the LOCKLIST database configuration setting at the time of the Explain invocation. |
| MAXLOCKS | SMALLINT | No | No | Contains the value of the MAXLOCKS database configuration setting at the time of the Explain invocation. |

## Explain Tables

*Table 114. EXPLAIN_INSTANCE Table (continued)*

| Column Name | Data Type | Nullable? | Key? | Description |
|---|---|---|---|---|
| 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 setting 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 setting at the time of Explain invocation. |
| COMM_SPEED | DOUBLE | No | No | Contains the value of the COMM_BANDWIDTH database configuration setting at the time of Explain invocation. |
| PARALLELISM | CHAR(2) | No | No | Possible values are:<br>• N = No parallelism<br>• P = Intra-partition parallelism<br>• IP = Inter-partition parallelism<br>• BP = Intra-partition parallelism and inter-partition parallelism |
| DATAJOINER | CHAR(1) | No | No | Possible values are:<br>• N = Non-federated systems plan<br>• Y = Federated systems plan |

## EXPLAIN_OBJECT Table

The EXPLAIN_OBJECT table identifies those data objects required by the access plan generated to satisfy the SQL statement.

For the definition of this table, see "EXPLAIN_OBJECT Table Definition" on page 1156.

*Table 115. EXPLAIN_OBJECT Table*

| 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. |
| EXPLAIN_LEVEL | CHAR(1) | No | FK | Level of Explain information for which this row is relevant. |

*Table 115. EXPLAIN_OBJECT Table (continued)*

| Column Name | Data Type | Nullable? | Key? | Description |
|---|---|---|---|---|
| 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 | INTEGER | 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 if the rows in the object are distinct (i.e. no duplicates)<br><br>Possible values are:<br><br>**Y**      Yes<br><br>**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. |
| 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. |
| 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. |
| PREFETCHSIZE | INTEGER | No | No | Number of data pages to be read when prefetch is performed. Set to -1 for a table function. |
| 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. |

*Table 115. EXPLAIN_OBJECT Table  (continued)*

| Column Name | Data Type | Nullable? | Key? | Description |
|---|---|---|---|---|
| NLEAF | INTEGER | 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. |
| 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 | INTEGER | 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 $\{2,3,4\}$ columns of the index. Set to $-1$ for a table, table function or if this statistic is not available. |
| FIRST3KEYCARD | BIGINT | No | No | |
| FIRST4KEYCARD | BIGINT | No | No | |
| SEQUENTIAL_PAGES | INTEGER | 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. |

*Table 116. Possible OBJECT_TYPE Values*

| Value | Description |
|---|---|
| IX | Index |
| TA | Table |
| TF | Table Function |

## EXPLAIN_OPERATOR Table

The EXPLAIN_OPERATOR table contains all the operators needed to satisfy the SQL statement by the SQL compiler.

For the definition of this table, see "EXPLAIN_OPERATOR Table Definition" on page 1157.

*Table 117. EXPLAIN_OPERATOR Table*

| 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. |
| 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. |
| 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. |
| 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 timerons) 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 executing the chosen access plan up to and including this operator. |
| FIRST_COMM_COST | DOUBLE | No | No | Estimated cumulative communications cost (in TCP/IP frames) of fetching the first row for the chosen access plan up to and including this operator. This value includes any initial overhead required. |

*Table 117. EXPLAIN_OPERATOR Table (continued)*

| Column Name | Data Type | Nullable? | Key? | Description |
|---|---|---|---|---|
| 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. |

*Table 118. OPERATOR_TYPE Values*

| Value | Description |
|---|---|
| DELETE | Delete |
| FETCH | Fetch |
| FILTER | Filter rows |
| GENROW | Generate Row |
| GRPBY | Group By |
| HSJOIN | Hash Join |
| INSERT | Insert |
| IXAND | Dynamic Bitmap Index ANDing |
| IXSCAN | Index Scan |
| MSJOIN | Merge Scan Join |
| NLJOIN | Nested loop Join |
| RETURN | Result |
| RIDSCN | Row Identifier (RID) Scan |
| RQUERY | Remote Query |
| SORT | Sort |
| TBSCAN | Table Scan |
| TEMP | Temporary Table Construction |
| TQ | Table Queue |
| UNION | Union |
| UNIQUE | Duplicate Elimination |
| UPDATE | Update |

## EXPLAIN_PREDICATE Table

The EXPLAIN_PREDICATE table identifies which predicates are applied by a specific operator.

For the definition of this table, see "EXPLAIN_PREDICATE Table Definition" on page 1158.

*Table 119. EXPLAIN_PREDICATE Table*

| 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. |
| 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. |
| HOW_APPLIED | CHAR(5) | 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. |

## Explain Tables

*Table 119. EXPLAIN_PREDICATE Table (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. |
| PREDICATE_TEXT | CLOB(1M) | Yes | No | The text of the predicate as recreated from the internal representation of the SQL statement. Null if not available. |

*Table 120. Possible HOW_APPLIED Values*

| Value | Description |
|---|---|
| JOIN | Used to join tables |
| RESID | Evaluated as a residual predicate |
| SARG | Evaluated as a sargable predicate for index or data page |
| START | Used as a start condition |
| STOP | Used as a stop condition |

*Table 121. Possible RELOP_TYPE Values*

| Value | Description |
|---|---|
| blanks | Not Applicable |
| EQ | Equals |
| GE | Greater Than or Equal |
| GT | Greater Than |
| IN | In list |
| LE | Less Than or Equal |
| LK | Like |
| LT | Less Than |
| NE | Not Equal |
| NL | Is Null |
| NN | Is Not Null |

## 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 and/or enhanced with additional predicates as determined by the SQL Compiler.

For the definition of this table, see "EXPLAIN_STATEMENT Table Definition" on page 1159.

*Table 122. EXPLAIN_STATEMENT Table*

| 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. |
| EXPLAIN_LEVEL | CHAR(1) | No | PK | Level of Explain information for which this row is relevant.<br><br>Valid values are:<br>**O**     Original Text (as entered by user)<br>**P**     PLAN SELECTION |
| 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. |

## Explain Tables

*Table 122. EXPLAIN_STATEMENT Table (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.<br><br>Possible values are:<br>**S** Select<br>**D** Delete<br>**DC** Delete where current of cursor<br>**I** Insert<br>**U** Update<br>**UC** Update where current of cursor |
| 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.<br><br>Possible values are:<br>**' '** Not applicable (blank)<br>**N** No<br>**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.<br><br>Possible values are:<br>**' '** Not applicable (blank)<br>**N** No<br>**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 *O* (original text) since no access plan has been chosen at this time. |

*Table 122. EXPLAIN_STATEMENT Table (continued)*

| Column Name | Data Type | Nullable? | Key? | Description |
|---|---|---|---|---|
| STATEMENT_TEXT | CLOB(1M) | No | No | Text or portion of the text of the SQL statement being explained. The text shown for the Plan Selection level 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.<br><br>This column is intended for use with DB2 Visual Explain. Column is set to null if EXPLAIN_LEVEL is *0* (original statement) since no access plan has been chosen at the time that this specific version of the statement is captured. |
| QUERY_DEGREE | INTEGER | No | No | Indicates the degree of intra-partition parallelism at the time of Explain invocation. For the original statement, this contains the directed degree of intra-partition parallelism. For the PLAN SELECTION, this contains the degree of intra-partition 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.

For the definition of this table, see "EXPLAIN_STREAM Table Definition" on page 1160.

*Table 123. EXPLAIN_STREAM Table*

| 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. |
| 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. |

## Explain Tables

*Table 123. EXPLAIN_STREAM Table (continued)*

| Column Name | Data Type | Nullable? | Key? | Description |
|---|---|---|---|---|
| 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:<br><br>**O**      Operator<br><br>**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'. |
| TARGET_TYPE | CHAR(1) | No | No | Indicates the target of this data stream:<br><br>**O**      Operator<br><br>**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(1M) | Yes | No | This column contains the names and ordering information of the columns involved in this stream.<br><br>These names will be in the format of:<br><br>`NAME1(A)+NAME2(D)+NAME3+NAME4`<br><br>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. |
| PMID | SMALLINT | No | No | Partitioning map ID. |

*Table 123. EXPLAIN_STREAM Table  (continued)*

| Column Name | Data Type | Nullable? | Key? | Description |
|---|---|---|---|---|
| SINGLE_NODE | CHAR(5) | Yes | No | Indicates if this data stream is on a single or multiple partitions:<br><br>**MULT**  On multiple partitions<br><br>**COOR**  On coordinator node<br><br>**HASH**  Directed using hashing<br><br>**RID**  Directed using the row ID<br><br>**FUNC**  Directed using a function (PARTITION() or NODENUMBER())<br><br>**CORR**  Directed using a correlation value |
| PARTITION_COLUMNS | CLOB(64K) | Yes | No | List of columns this data stream is partitioned on. |

## ADVISE_INDEX Table

The ADVISE_INDEX table represents the recommended indexes.

For the definition of this table, see "ADVISE_INDEX Table Definition" on page 1161.

*Table 124. ADVISE_INDEX Table*

| 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 static SQL was explained. |
| SOURCE_SCHEMA | VARCHAR(128) | No | No | Schema, or qualifier, of source of 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. |

## Explain Tables

*Table 124. ADVISE_INDEX Table (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. |
| 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(64K) | No | No | List of column names. |
| UNIQUERULE | CHAR(1) | No | No | Unique rule:<br>D = Duplicates allowed<br>P = Primary index<br>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. |
| NLEAF | INTEGER | 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. |
| FULLKEYCARD | BIGINT | No | No | Number of distinct full key values; −1 if statistics are not gathered. |
| FIRSTKEYCARD | BIGINT | No | No | Number of distinct first 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). |
| 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 124. ADVISE_INDEX Table (continued)*

| Column Name | Data Type | Nullable? | Key? | Description |
|---|---|---|---|---|
| SYSTEM_REQUIRED | SMALLINT | No | No | 1 if this index is required for primary key or unique key constraint, OR if this is the index on the object identifier (OID) column of a typed table.<br><br>2 if this index is required for primary key or unique key constraint, AND this is the index on the object identifier (OID) column of a typed table.<br><br>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(254) | 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 | INTEGER | 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) |
| 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) |

## Explain Tables

*Table 124. ADVISE_INDEX Table  (continued)*

| Column Name | Data Type | Nullable? | Key? | Description |
|---|---|---|---|---|
| MINPCTUSED | SMALLINT | No | No | If not zero, then on-line index reorganization 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<br>N = Index does not support reverse scans |
| USE_INDEX | CHAR(1) | Yes | No | Y = index recommended or evaluated<br>N = index not to be recommended |
| CREATION_TEXT | CLOB(1M) | No | No | The SQL statement used to create the index. |
| PACKED_DESC | BLOB(20M) | Yes | No | Internal description of the table. |

## ADVISE_WORKLOAD Table

The ADVISE_WORKLOAD table represents the statement that makes up the workload. For more details on workload refer to *Administration Guide, Performance*.

For the definition of this table, see "ADVISE_WORKLOAD Table Definition" on page 1163.

*Table 125. ADVISE_WORKLOAD Table*

| Column Name | Data Type | Nullable? | Key? | Description |
|---|---|---|---|---|
| WORKLOAD_NAME | CHAR(128) | No | No | Name of the collection of SQL statements (workload) that this statments belongs to. |
| 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. |
| COST_BEFORE | DOUBLE | Yes | No | The cost (in timerons) of the query if the recommended indexes are not created. |
| COST_AFTER | DOUBLE | Yes | No | The cost (in timerons) of the query if the recommended indexes are created. |

## Table Definitions for Explain Tables

The Explain tables must be created before Explain can be invoked. The following definitions specify how to create the necessary Explain tables:

- "EXPLAIN_ARGUMENT Table Definition" on page 1154

- "EXPLAIN_INSTANCE Table Definition" on page 1155

- "EXPLAIN_OBJECT Table Definition" on page 1156

- "EXPLAIN_OPERATOR Table Definition" on page 1157

- "EXPLAIN_PREDICATE Table Definition" on page 1158

- "EXPLAIN_STATEMENT Table Definition" on page 1159

- "EXPLAIN_STREAM Table Definition" on page 1160

- "ADVISE_INDEX Table Definition" on page 1161

- "ADVISE_WORKLOAD Table Definition" on page 1163

Alternately, create them by using the sample command line processor input script provided in the EXPLAIN.DDL file located in the 'misc' subdirectory of the 'sqllib' directory. Connect to the database where the Explain tables are required. Then issue the command: `db2 -tf EXPLAIN.DDL` and the tables will be created.

**Explain Tables**

### EXPLAIN_ARGUMENT Table Definition

```
CREATE TABLE EXPLAIN_ARGUMENT ( EXPLAIN_REQUESTER   VARCHAR(128)  NOT NULL,
                                EXPLAIN_TIME        TIMESTAMP     NOT NULL,
                                SOURCE_NAME         VARCHAR(128)  NOT NULL,
                                SOURCE_SCHEMA       VARCHAR(128)  NOT NULL,
                                EXPLAIN_LEVEL       CHAR(1)       NOT NULL,
                                STMTNO              INTEGER       NOT NULL,
                                SECTNO              INTEGER       NOT NULL,
                                OPERATOR_ID         INTEGER       NOT NULL,
                                ARGUMENT_TYPE       CHAR(8)       NOT NULL,
                                ARGUMENT_VALUE      VARCHAR(1024) NOT NULL,
                                LONG_ARGUMENT_VALUE CLOB(1M)      NOT LOGGED,
                                   FOREIGN KEY (EXPLAIN_REQUESTER,
                                                EXPLAIN_TIME,
                                                SOURCE_NAME,
                                                SOURCE_SCHEMA,
                                                EXPLAIN_LEVEL,
                                                STMTNO,
                                                SECTNO)
                                   REFERENCES EXPLAIN_STATEMENT
                                   ON DELETE CASCADE )
```

## EXPLAIN_INSTANCE Table Definition

```
CREATE TABLE EXPLAIN_INSTANCE ( EXPLAIN_REQUESTER VARCHAR(128) NOT NULL,
                                EXPLAIN_TIME      TIMESTAMP   NOT NULL,
                                SOURCE_NAME       VARCHAR(128) NOT NULL,
                                SOURCE_SCHEMA     VARCHAR(128) NOT NULL,
                                EXPLAIN_OPTION    CHAR(1)     NOT NULL,
                                SNAPSHOT_TAKEN    CHAR(1)     NOT NULL,
                                DB2_VERSION       CHAR(7)     NOT NULL,
                                SQL_TYPE          CHAR(1)     NOT NULL,
                                QUERYOPT          INTEGER     NOT NULL,
                                BLOCK             CHAR(1)     NOT NULL,
                                ISOLATION         CHAR(2)     NOT NULL,
                                BUFFPAGE          INTEGER     NOT NULL,
                                AVG_APPLS         INTEGER     NOT NULL,
                                SORTHEAP          INTEGER     NOT NULL,
                                LOCKLIST          INTEGER     NOT NULL,
                                MAXLOCKS          SMALLINT    NOT NULL,
                                LOCKS_AVAIL       INTEGER     NOT NULL,
                                CPU_SPEED         DOUBLE      NOT NULL,
                                REMARKS           VARCHAR(254),
                                DBHEAP            INTEGER     NOT NULL,
                                COMM_SPEED        DOUBLE      NOT NULL,
                                PARALLELISM       CHAR(2)     NOT NULL,
                                DATAJOINER        CHAR(1)     NOT NULL,
                                   PRIMARY KEY (EXPLAIN_REQUESTER,
                                                EXPLAIN_TIME,
                                                SOURCE_NAME,
                                                SOURCE_SCHEMA))
```

**Explain Tables**

### EXPLAIN_OBJECT Table Definition

```
CREATE TABLE EXPLAIN_OBJECT ( EXPLAIN_REQUESTER VARCHAR(128) NOT NULL,
                              EXPLAIN_TIME      TIMESTAMP   NOT NULL,
                              SOURCE_NAME       VARCHAR(128) NOT NULL,
                              SOURCE_SCHEMA     VARCHAR(128) NOT NULL,
                              EXPLAIN_LEVEL     CHAR(1)     NOT NULL,
                              STMTNO            INTEGER     NOT NULL,
                              SECTNO            INTEGER     NOT NULL,
                              OBJECT_SCHEMA     VARCHAR(128) NOT NULL,
                              OBJECT_NAME       VARCHAR(128) NOT NULL,
                              OBJECT_TYPE       CHAR(2)     NOT NULL,
                              CREATE_TIME       TIMESTAMP,
                              STATISTICS_TIME   TIMESTAMP,
                              COLUMN_COUNT      SMALLINT    NOT NULL,
                              ROW_COUNT         INTEGER     NOT NULL,
                              WIDTH             INTEGER     NOT NULL,
                              PAGES             INTEGER     NOT NULL,
                              DISTINCT          CHAR(1)     NOT NULL,
                              TABLESPACE_NAME   VARCHAR(128),
                              OVERHEAD          DOUBLE      NOT NULL,
                              TRANSFER_RATE     DOUBLE      NOT NULL,
                              PREFETCHSIZE      INTEGER     NOT NULL,
                              EXTENTSIZE        INTEGER     NOT NULL,
                              CLUSTER           DOUBLE      NOT NULL,
                              NLEAF             INTEGER     NOT NULL,
                              NLEVELS           INTEGER     NOT NULL,
                              FULLKEYCARD       BIGINT      NOT NULL,
                              OVERFLOW          INTEGER     NOT NULL,
                              FIRSTKEYCARD      BIGINT      NOT NULL,
                              FIRST2KEYCARD     BIGINT      NOT NULL,
                              FIRST3KEYCARD     BIGINT      NOT NULL,
                              FIRST4KEYCARD     BIGINT      NOT NULL,
                              SEQUENTIAL_PAGES  INTEGER     NOT NULL,
                              DENSITY           INTEGER     NOT NULL,
                                      FOREIGN KEY (EXPLAIN_REQUESTER,
                                                   EXPLAIN_TIME,
                                                   SOURCE_NAME,
                                                   SOURCE_SCHEMA,
                                                   EXPLAIN_LEVEL,
                                                   STMTNO,
                                                   SECTNO)
                                  REFERENCES EXPLAIN_STATEMENT
                                  ON DELETE CASCADE )
```

## EXPLAIN_OPERATOR Table Definition

```
CREATE TABLE EXPLAIN_OPERATOR ( EXPLAIN_REQUESTER VARCHAR(128) NOT NULL,
                                EXPLAIN_TIME      TIMESTAMP   NOT NULL,
                                SOURCE_NAME       VARCHAR(128) NOT NULL,
                                SOURCE_SCHEMA     VARCHAR(128) NOT NULL,
                                EXPLAIN_LEVEL     CHAR(1)     NOT NULL,
                                STMTNO            INTEGER     NOT NULL,
                                SECTNO            INTEGER     NOT NULL,
                                OPERATOR_ID       INTEGER     NOT NULL,
                                OPERATOR_TYPE     CHAR(6)     NOT NULL,
                                TOTAL_COST        DOUBLE      NOT NULL,
                                IO_COST           DOUBLE      NOT NULL,
                                CPU_COST          DOUBLE      NOT NULL,
                                FIRST_ROW_COST    DOUBLE      NOT NULL,
                                RE_TOTAL_COST     DOUBLE      NOT NULL,
                                RE_IO_COST        DOUBLE      NOT NULL,
                                RE_CPU_COST       DOUBLE      NOT NULL,
                                COMM_COST         DOUBLE      NOT NULL,
                                FIRST_COMM_COST   DOUBLE      NOT NULL,
                                REMOTE_TOTAL_COST DOUBLE      NOT NULL,
                                REMOTE_COMM_COST  DOUBLE      NOT NULL,
                                    FOREIGN KEY (EXPLAIN_REQUESTER,
                                                 EXPLAIN_TIME,
                                                 SOURCE_NAME,
                                                 SOURCE_SCHEMA,
                                                 EXPLAIN_LEVEL,
                                                 STMTNO,
                                                 SECTNO)
                                    REFERENCES EXPLAIN_STATEMENT
                                    ON DELETE CASCADE )
```

**Explain Tables**

## EXPLAIN_PREDICATE Table Definition

```
CREATE TABLE EXPLAIN_PREDICATE ( EXPLAIN_REQUESTER VARCHAR(128) NOT NULL,
                                 EXPLAIN_TIME      TIMESTAMP    NOT NULL,
                                 SOURCE_NAME       VARCHAR(128) NOT NULL,
                                 SOURCE_SCHEMA     VARCHAR(128) NOT NULL,
                                 EXPLAIN_LEVEL     CHAR(1)      NOT NULL,
                                 STMTNO            INTEGER      NOT NULL,
                                 SECTNO            INTEGER      NOT NULL,
                                 OPERATOR_ID       INTEGER      NOT NULL,
                                 PREDICATE_ID      INTEGER      NOT NULL,
                                 HOW_APPLIED       CHAR(5)      NOT NULL,
                                 WHEN_EVALUATED    CHAR(3)      NOT NULL,
                                 RELOP_TYPE        CHAR(2)      NOT NULL,
                                 SUBQUERY          CHAR(1)      NOT NULL,
                                 FILTER_FACTOR     DOUBLE       NOT NULL,
                                 PREDICATE_TEXT    CLOB(1M)     NOT LOGGED,
                                     FOREIGN KEY (EXPLAIN_REQUESTER,
                                                  EXPLAIN_TIME,
                                                  SOURCE_NAME,
                                                  SOURCE_SCHEMA,
                                                  EXPLAIN_LEVEL,
                                                  STMTNO,
                                                  SECTNO)
                                     REFERENCES EXPLAIN_STATEMENT
                                     ON DELETE CASCADE )
```

## EXPLAIN_STATEMENT Table Definition

```
CREATE TABLE EXPLAIN_STATEMENT ( EXPLAIN_REQUESTER VARCHAR(128) NOT NULL,
                                 EXPLAIN_TIME      TIMESTAMP    NOT NULL,
                                 SOURCE_NAME       VARCHAR(128) NOT NULL,
                                 SOURCE_SCHEMA     VARCHAR(128) NOT NULL,
                                 EXPLAIN_LEVEL     CHAR(1)      NOT NULL,
                                 STMTNO            INTEGER      NOT NULL,
                                 SECTNO            INTEGER      NOT NULL,
                                 QUERYNO           INTEGER      NOT NULL,
                                 QUERYTAG          CHAR(20)     NOT NULL,
                                 STATEMENT_TYPE    CHAR(2)      NOT NULL,
                                 UPDATABLE         CHAR(1)      NOT NULL,
                                 DELETABLE         CHAR(1)      NOT NULL
                                 TOTAL_COST        DOUBLE       NOT NULL,
                                 STATEMENT_TEXT    CLOB(1M)     NOT NULL
                                                                NOT LOGGED,
                                 SNAPSHOT          BLOB(10M)    NOT LOGGED,
                                 QUERY_DEGREE      INTEGER      NOT NULL,
                                     PRIMARY KEY (EXPLAIN_REQUESTER,
                                                  EXPLAIN_TIME,
                                                  SOURCE_NAME,
                                                  SOURCE_SCHEMA,
                                                  EXPLAIN_LEVEL,
                                                  STMTNO,
                                                  SECTNO),
                                     FOREIGN KEY (EXPLAIN_REQUESTER,
                                                  EXPLAIN_TIME,
                                                  SOURCE_NAME,
                                                  SOURCE_SCHEMA)
                                     REFERENCES EXPLAIN_INSTANCE
                                     ON DELETE CASCADE )
```

**Explain Tables**

### EXPLAIN_STREAM Table Definition

```
CREATE TABLE EXPLAIN_STREAM ( EXPLAIN_REQUESTER VARCHAR(128) NOT NULL,
                              EXPLAIN_TIME      TIMESTAMP    NOT NULL,
                              SOURCE_NAME       VARCHAR(128) NOT NULL,
                              SOURCE_SCHEMA     VARCHAR(128) NOT NULL,
                              EXPLAIN_LEVEL     CHAR(1)      NOT NULL,
                              STMTNO            INTEGER      NOT NULL,
                              SECTNO            INTEGER      NOT NULL,
                              STREAM_ID         INTEGER      NOT NULL,
                              SOURCE_TYPE       CHAR(1)      NOT NULL,
                              SOURCE_ID         SMALLINT     NOT NULL,
                              TARGET_TYPE       CHAR(1)      NOT NULL,
                              TARGET_ID         SMALLINT     NOT NULL,
                              OBJECT_SCHEMA     VARCHAR(128),
                              OBJECT_NAME       VARCHAR(128),
                              STREAM_COUNT      DOUBLE       NOT NULL,
                              COLUMN_COUNT      SMALLINT     NOT NULL,
                              PREDICATE_ID      INTEGER      NOT NULL,
                              COLUMN_NAMES      CLOB(1M)     NOT LOGGED,
                              PMID              SMALLINT     NOT NULL,
                              SINGLE_NODE       CHAR(5),
                              PARTITION_COLUMNS CLOB(64K)    NOT LOGGED,
                                 FOREIGN KEY (EXPLAIN_REQUESTER,
                                              EXPLAIN_TIME,
                                              SOURCE_NAME,
                                              SOURCE_SCHEMA,
                                              EXPLAIN_LEVEL,
                                              STMTNO,
                                              SECTNO)
                                 REFERENCES EXPLAIN_STATEMENT
                                 ON DELETE CASCADE )
```

## ADVISE_INDEX Table Definition

```
CREATE TABLE ADVISE_INDEX (EXPLAIN_REQUESTER VARCHAR(128) NOT NULL
                                             WITH DEFAULT '',
                          EXPLAIN_TIME      TIMESTAMP   NOT NULL
                                             WITH DEFAULT CURRENT TIMESTAMP,
                          SOURCE_NAME       VARCHAR(128) NOT NULL
                                             WITH DEFAULT '',
                          SOURCE_SCHEMA     VARCHAR(128) NOT NULL
                                             WITH DEFAULT '',
                          EXPLAIN_LEVEL     CHAR(1)     NOT NULL
                                             WITH DEFAULT '',
                          STMTNO            INTEGER     NOT NULL
                                             WITH DEFAULT 0,
                          SECTNO            INTEGER     NOT NULL
                                             WITH DEFAULT 0,
                          QUERYNO           INTEGER     NOT NULL
                                             WITH DEFAULT 0,
                          QUERYTAG          CHAR(20)    NOT NULL
                                             WITH DEFAULT '',
                          NAME              VARCHAR(128) NOT NULL,
                          CREATOR           VARCHAR(128) NOT NULL
                                             WITH DEFAULT '',
                          TBNAME            VARCHAR(128) NOT NULL,
                          TBCREATOR         VARCHAR(128) NOT NULL
                                             WITH DEFAULT '',
                          COLNAMES          CLOB(64K)   NOT NULL,
                          UNIQUERULE        CHAR(1)     NOT NULL
                                             WITH DEFAULT '',
                          COLCOUNT          SMALLINT    NOT NULL
                                             WITH DEFAULT 0,
                          IID               SMALLINT    NOT NULL
                                             WITH DEFAULT 0,
                          NLEAF             INTEGER     NOT NULL
                                             WITH DEFAULT 0,
                          NLEVELS           SMALLINT    NOT NULL
                                             WITH DEFAULT 0,
                          FIRSTKEYCARD      BIGINT      NOT NULL
                                             WITH DEFAULT 0,
                          FULLKEYCARD       BIGINT      NOT NULL
                                             WITH DEFAULT 0,
                          CLUSTERRATIO      SMALLINT    NOT NULL
                                             WITH DEFAULT 0,
                          CLUSTERFACTOR     DOUBLE      NOT NULL
                                             WITH DEFAULT 0,
                          USERDEFINED       SMALLINT    NOT NULL
                                             WITH DEFAULT 0,
                          SYSTEM_REQUIRED   SMALLINT    NOT NULL
                                             WITH DEFAULT 0,
                          CREATE_TIME       TIMESTAMP   NOT NULL
                                             WITH DEFAULT CURRENT TIMESTAMP,
                          STATS_TIME        TIMESTAMP
                                             WITH DEFAULT CURRENT TIMESTAMP,
                          PAGE_FETCH_PAIRS  VARCHAR(254) NOT NULL
                                             WITH DEFAULT '',
                          REMARKS           VARCHAR(254)
```

## Explain Tables

```
                         WITH DEFAULT '',
DEFINER           VARCHAR(128) NOT NULL
                         WITH DEFAULT '',
CONVERTED          CHAR(1) NOT NULL
                         WITH DEFAULT '',
SEQUENTIAL_PAGES INTEGER     NOT NULL
                         WITH DEFAULT 0,
DENSITY           INTEGER     NOT NULL
                         WITH DEFAULT 0,
FIRST2KEYCARD     BIGINT      NOT NULL
                         WITH DEFAULT 0,
FIRST3KEYCARD     BIGINT      NOT NULL
                         WITH DEFAULT 0,
FIRST4KEYCARD     BIGINT      NOT NULL
                         WITH DEFAULT 0,
PCTFREE           SMALLINT    NOT NULL
                         WITH DEFAULT -1,
UNIQUE_COLCOUNT   SMALLINT    NOT NULL
                         WITH DEFAULT -1,
MINPCTUSED        SMALLINT    NOT NULL
                         WITH DEFAULT 0,
REVERSE_SCANS     CHAR(1)     NOT NULL
                         WITH DEFAULT 'N',
USE_INDEX         CHAR(1),
CREATION_TEXT     CLOB(1M)    NOT NULL
                         NOT LOGGED WITH DEFAULT '',
PACKED_DESC       BLOB(1M)    NOT LOGGED)
```

## ADVISE_WORKLOAD Table Definition

```
CREATE TABLE ADVISE_WORKLOAD (WORKLOAD_NAME  CHAR(128)    NOT NULL
                                             WITH DEFAULT 'WK0',
                             STATEMENT_NO    INTEGER      NOT NULL
                                             WITH DEFAULT 1,
                             STATEMENT_TEXT  CLOB(1M)     NOT NULL NOT LOGGED,
                             STATEMENT_TAG   VARCHAR(256) NOT NULL
                                             WITH DEFAULT '',
                             FREQUENCY       INTEGER      NOT NULL
                                             WITH DEFAULT 1,
                             IMPORTANCE      DOUBLE       NOT NULL
                                             WITH DEFAULT 1,
                             COST_BEFORE     DOUBLE,
                             COST_AFTER      DOUBLE)
```

**Explain Tables**

# Appendix L. Explain Register Values

This appendix describes the interaction of the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values with each other and with the PREP and BIND commands.

The CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values interact in the following way for dynamic SQL.

*Table 126. Interaction of Explain Special Register Values for Dynamic SQL*

| EXPLAIN SNAPSHOT values | EXPLAIN MODE values | | | | |
|---|---|---|---|---|---|
| | NO | YES | EXPLAIN | RECOMMEND INDEXES | EVALUATE INDEXES |
| NO | • Results of query returned. | • Explain tables populated<br>• Results of query returned. | • Explain tables populated.<br>• Results of query not returned (Dynamic statements not executed). | • Explain tables populated.<br>• Results of query not returned (Dynamic statements not executed).<br>• Indexes recommended. | • Explain tables populated.<br>• Results of query not returned (Dynamic statements not executed).<br>• Indexes evaluated. |
| YES | • Explain Snapshot taken.<br>• Results of query returned. | • Explain tables populated<br>• Explain Snapshot taken<br>• Results of query returned. | • Explain tables populated<br>• Explain Snapshot taken<br>• Results of query not returned (Dynamic statements not executed). | • Explain tables populated<br>• Explain Snapshot taken<br>• Results of query not returned (Dynamic statements not executed).<br>• Indexes recommended. | • Explain tables populated<br>• Explain Snapshot taken<br>• Results of query not returned (Dynamic statements not executed).<br>• Indexes evaluated. |

*Table 126. Interaction of Explain Special Register Values for Dynamic SQL  (continued)*

| EXPLAIN SNAPSHOT values | EXPLAIN MODE values | | | | |
|---|---|---|---|---|---|
| | **NO** | **YES** | **EXPLAIN** | **RECOMMEND INDEXES** | **EVALUATE INDEXES** |
| EXPLAIN | • Explain Snapshot taken<br>• Results of query not returned (Dynamic statements not executed). | • Explain tables populated<br>• Explain Snapshot taken<br>• Results of query not returned (Dynamic statements not executed). | • Explain tables populated<br>• Explain Snapshot taken<br>• Results of query not returned (Dynamic statements not executed). | • Explain tables populated<br>• Explain Snapshot taken<br>• Results of query not returned (Dynamic statements not executed).<br>• Indexes recommended. | • Explain tables populated<br>• Explain Snapshot taken<br>• Results of query not returned (Dynamic statements not executed).<br>• Indexes evaluated. |

The CURRENT EXPLAIN MODE special register interacts with the EXPLAIN bind option in the following way for dynamic SQL.

*Table 127. Interaction of EXPLAIN Bind Option and CURRENT EXPLAIN MODE*

| EXPLAIN MODE values | EXPLAIN Bind option values | | |
|---|---|---|---|
| | **NO** | **YES** | **ALL** |
| NO | • Results of query returned. | • Explain tables populated for static SQL<br>• Results of query returned. | • Explain tables populated for static SQL<br>• Explain tables populated for dynamic SQL<br>• Results of query returned. |
| YES | • Explain tables populated for dynamic SQL<br>• Results of query returned. | • Explain tables populated for static SQL<br>• Explain tables populated for dynamic SQL<br>• Results of query returned. | • Explain tables populated for static SQL<br>• Explain tables populated for dynamic SQL<br>• Results of query returned. |
| EXPLAIN | • Explain tables populated for dynamic SQL<br>• Results of query not returned (Dynamic statements not executed). | • Explain tables populated for static SQL<br>• Explain tables populated for dynamic SQL<br>• Results of query not returned (Dynamic statements not executed). | • Explain tables populated for static SQL<br>• Explain tables populated for dynamic SQL<br>• Results of query not returned (Dynamic statements not executed). |

*Table 127. Interaction of EXPLAIN Bind Option and CURRENT EXPLAIN MODE (continued)*

| EXPLAIN MODE values | EXPLAIN Bind option values | | |
|---|---|---|---|
| | NO | YES | ALL |
| RECOMMEND INDEXES | • Explain tables populated for dynamic SQL<br>• Results of query not returned (Dynamic statements not executed).<br>• Recommend indexes | • Explain tables populated for static SQL<br>• Explain tables populated for dynamic SQL<br>• Results of query not returned (Dynamic statements not executed).<br>• Recommend indexes | • Explain tables populated for static SQL<br>• Explain tables populated for dynamic SQL<br>• Results of query not returned (Dynamic statements not executed).<br>• Recommend indexes |
| EVALUATE INDEXES | • Explain tables populated for dynamic SQL<br>• Results of query not returned (Dynamic statements not executed).<br>• Evaluate indexes | • Explain tables populated for static SQL<br>• Explain tables populated for dynamic SQL<br>• Results of query not returned (Dynamic statements not executed).<br>• Evaluate indexes | • Explain tables populated for static SQL<br>• Explain tables populated for dynamic SQL<br>• Results of query not returned (Dynamic statements not executed).<br>• Evaluate indexes |

The CURRENT EXPLAIN SNAPSHOT special register interacts with the EXPLSNAP bind option in the following way for dynamic SQL.

*Table 128. Interaction of EXPLSNAP bind Option and CURRENT EXPLAIN SNAPSHOT*

| EXPLAIN SNAPSHOT values | EXPLSNAP Bind option values | | |
|---|---|---|---|
| | NO | YES | ALL |
| NO | • Results of query returned. | • Explain Snapshot taken for static SQL<br>• Results of query returned. | • Explain Snapshot taken for static SQL<br>• Explain Snapshot taken for dynamic SQL<br>• Results of query returned. |
| YES | • Explain Snapshot taken for dynamic SQL<br>• Results of query returned. | • Explain Snapshot taken for static SQL<br>• Explain Snapshot taken for dynamic SQL<br>• Results of query returned. | • Explain Snapshot taken for static SQL<br>• Explain Snapshot taken for dynamic SQL<br>• Results of query returned. |
| EXPLAIN | • Explain Snapshot taken for dynamic SQL<br>• Results of query not returned (Dynamic statements not executed). | • Explain Snapshot taken for static SQL<br>• Explain Snapshot taken for dynamic SQL<br>• Results of query not returned (Dynamic statements not executed). | • Explain Snapshot taken for static SQL<br>• Explain Snapshot taken for dynamic SQL<br>• Results of query not returned (Dynamic statements not executed). |

# Appendix M. 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);
```

In order to give query results for this example, assume 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
```

## Recursion Example: Bill of Materials

```
            FROM RPL PARENT, PARTLIST CHILD
            WHERE  PARENT.SUBPART = CHILD.PART
       )
SELECT DISTINCT PART, SUBPART, QUANTITY
 FROM RPL
   ORDER BY PART, SUBPART, QUANTITY;
```

The above 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 refer to 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 from part '01' we go to '02' which goes to '06' and so on. Further, notice that part '06' is reached twice, once through '01' directly

1170

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. However, care should also be exercised in determining what to code so that there is a definite end of the recursion cycle.

The result produced by this example query could 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 need 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 above query, the select list of the second operand of the UNION in the recursive common table expression, identified by the name *RPL*, shows the

## Recursion Example: Bill of Materials

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 column 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 may 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;
```

### Recursion Example: Bill of Materials

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
```

**Recursion Example: Bill of Materials**

# Appendix N. Exception Tables

Exception tables are user-created tables that mimic the definition of the tables that are specified to be checked using SET INTEGRITY 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 used with LOAD are identical to the ones used here. They 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:

1. The first "n" columns of the exception table are the same as the columns of the table being checked. All column attributes including name, type and length should be identical.

2. All the columns of the exception table must be free of any constraints and triggers. Constraints include referential integrity, check constraints as well as unique index constraints that could cause errors on insert.

3. 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.

4. 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.

5. The exception table should be created with both "(n+1)" and the "(n+2)" columns.

6. There is no enforcement of any particular name for the above additional columns. However, the type specification must be exactly followed.

7. No additional columns are allowed.

8. If the original table has DATALINK columns, the corresponding columns in the exception table should specify NO LINK CONTROL. This ensures

that a file is not linked when a row (with DATALINK column) is inserted and an access token is not generated when rows are selected from the exception table.

9. It should also be noted that users invoking SET INTEGRITY to check the data must have INSERT privilege on the exception tables.

The information in the "message" column will be according to the following structure :

*Table 129. Exception Table Message Column Structure*

| Field number | Contents | Size | Comments |
|---|---|---|---|
| 1 | Number of constraint violations | 5 characters | Right justified padded with '0' |
| 2 | Type of first constraint violation | 1 character | 'K' - Check Constraint violation<br>'F' - Foreign Key violation<br>'I' - Unique Index violation<br>    (See Note)<br>'L' - DATALINK load violation |
| 3 | Length of constraint/index ID [a]/ DLVDESC [b] | 5 characters | Right justified padded with '0' |
| 4 | Constraint name/index ID [a]/ DLVDESC [b] | length from the previous field | |
| 5 | Separator | 3 characters | <space><colon><space> |
| 6 | Type of next constraint violation | 1 character | 'K' - Check Constraint violation<br>'F' - Foreign Key violation<br>'I' - Unique Index violation<br>'L' - DATALINK load violation |
| 7 | Length of constraint/index name/ DLVDESC | 5 characters | Right justified padded with '0' |
| 8 | Constraint name/ Index name/ DLVDESC | length from the previous field | |
| ..... | ..... | ..... | Repeat Field 5 through 8 for each violation |

**Note:**

- In DB2 Version 6, unique index violations will not occur with checking using SET INTEGRITY. This will be reported, however, when running LOAD if the FOR EXCEPTION option is chosen. LOAD on the other hand will not report check constraint and foreign key violations in the exception tables in DB2 Version 6.
- [a] 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`.
- [b]DLVDESC is a DATALINK Load Violation DESCriptor described below.

*Table 130. DATALINK Load Violation DESCriptor (DLVDESC)*

| Field number | Contents | Size | Comments |
|---|---|---|---|
| 1 | Number of violating DATALINK columns | 4 characters | Right justified padded with '0' |
| 2 | DATALINK column number of the first violating column | 4 characters | Right justified padded with '0' |
| 2 | DATALINK column number of the second violating column | 4 characters | Right justified padded with '0' |
| ..... | ..... | ..... | Repeat for each violating column number |

**Note:**

• DATALINK column number is COLNO in SYSCAT.COLUMNS for the appropriate table.

## Handling Rows in the Exception Tables

The information in the exception tables can be processed in any manner desired. The rows could be used to correct the data and re-insert the rows 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 wish to complete the load with the corrected rows from exception tables without firing the triggers, the following ways are suggested:
• Design the INSERT triggers to be fired depending on the value in a column defined explicitly for the purpose.
• Unload the data from the exception tables and append them using LOAD. In this case if we re-check the data, it should be noted that in DB2 Version 6 the constraint violation checking is not confined to the appended rows only.
• Save the trigger text from the relevant catalog table. Then drop the INSERT trigger and use INSERT to transfer the corrected rows from the exception tables. Finally recreate the trigger using the saved information.

In DB2 Version 6, no explicit provision is made to prevent the firing of triggers when inserting rows from the exception tables.

Only one violation per row will be reported for unique index violations.

If values with long string or LOB data types are in the table, the values will not be inserted into the exception table in case of unique index violation.

## Querying the Exception Tables

The message column structure in an exception table is a concatenated list of constraint names, lengths and delimiters as described earlier. You may wish to write a query on this information.

For example, let's write a query to obtain a list of all the violations, repeating each row with only the constraint name along with it. Let us assume that our original table T1 had two columns C1 and C2. Assume also, that the corresponding exception table E1 has columns C1, C2 pertaining to those in T1 and MSGCOL as the message column. The following query (using recursion) will list one constraint name per row (repeating the row for 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;
```

If we want all the rows that violated a particular constraint, we could extend this query 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';
```

To obtain all the Check Constraint violations, one could execute the following:

```
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';
```

# Appendix O. Japanese and Traditional-Chinese EUC Considerations

Extended Unix Code (EUC) for Japanese and Traditional-Chinese defines a set of encoding rules that can support from 1 to 4 character sets. In some cases, such as Japanese EUC (eucJP) and Traditional-Chinese EUC (eucTW), a character may be encoded using more than two bytes. Use of such an encoding scheme has implications when used as the code page of the database server or the database client. The key considerations involve the following:

- expansion or contraction of strings when converting between EUC code pages and double-byte code pages
- use of Universal Character Set-2 (UCS-2) as the code page for graphic data stored in a database server defined with the eucJP (Japanese) or eucTW (Traditional-Chinese) code pages.

With the exception of these considerations, the use of EUC is consistent with the double-byte character set (DBCS) support. Throughout this book (and others), references to *double-byte* have been changed to *multi-byte* to reflect support for encoding rules that allow for character representations that require more than 2 bytes. Detailed considerations for support of Japanese and Traditional-Chinese EUC are included here, organized in the same way as the contents of the chapters of this book. This information should be considered by anyone using SQL with an EUC database server or an EUC database client and used in conjunction with application development information in the *Application Development Guide*

## Language Elements

### Characters

Each multi-byte character is considered a *letter* with the exception of the double-byte blank character which is considered a *special character*.

### Tokens

Multi-byte lowercase alphabetic letters are not folded to uppercase. This differs from the single byte lowercase alphabetic letters in tokens which are generally folded to uppercase.

# Japanese and Traditional-Chinese EUC Considerations

## Identifiers

### SQL Identifiers

Conversion between a double-byte code page and an EUC code page may result in the conversion of double-byte characters to multi-byte characters encoded with more than 2 bytes. As a result, an identifier that fits the length maximum in the double-byte code page may exceed the length in the EUC code page. Selecting identifiers for this type of environment must be done carefully to avoid expansion beyond the maximum identifier length.

## Data Types

### Character Strings

In an MBCS database, character strings may contain a mixture of characters from a single-byte character set (SBCS) and from multi-byte character sets (MBCS). When using such strings, operations may provide different results if they are character based (treat the data as characters) or byte based (treat the data as bytes). Check the function or operation description to determine how mixed strings are processed.

### Graphic Strings

A graphic string is defined as a sequence of double-byte character data. In order to allow Japanese or Traditional-Chinese EUC data to be stored in graphic columns, EUC characters are encoded in UCS-2. Characters that are not double-byte characters under all supported encoding schemes (for example, PC or EBCDIC DBCS) should not be used with graphic columns. The results of using other than double-byte characters may result in replacement by substitution characters during conversion. Retrieval of such data will not return the same value as was entered. Refer to the *Application Development Guide* programming language sections for details on handling graphic data in host variables.

## Assignments and Comparisons

### String Assignments

Conversion of a string is performed prior to the assignment. In cases involving an eucJP/eucTW code page and a DBCS code page, a character string may become longer (DBCS to eucJP/eucTW) or shorter (eucJP/eucTW to DBCS). This may result in errors on storage assignment and truncation on retrieval assignment. When the error on storage assignment is due to expansion during conversion, SQLSTATE 22524 is returned instead of SQLSTATE 22001.

## Japanese and Traditional-Chinese EUC Considerations

Similarly, assignments involving graphic strings may result in the conversion of a UCS-2 encoded double-byte character to a substitution character in a PC or EBCDIC DBCS code page for characters that do not have a corresponding double-byte character. Assignments that replace characters with substitution characters will indicate this by setting the SQLWARN10 field of the SQLCA to 'W'.

In cases of truncation during retrieval assignment involving multi-byte character strings, the point of truncation may be part of a multi-byte character. In this case, each byte of the character fragment is replaced with a single-byte blank. This means that more than one single-byte blank may appear at the end of a truncated character string.

### String Comparisons

String comparisons are performed on a byte basis. Character strings also use the collating sequence defined for the database. Graphic strings do not use the collating sequence and, in an eucJP or eucTW database, are encoded using UCS-2. Thus, the comparison of two mixed character strings may have a different result from the comparison of two graphic strings even though they contain the same characters. Similarly, the resulting sort order of a mixed character column and a graphic column may be different.

## Rules for Result Data Types

The resulting data type for character strings is not affected by the possible expansion of the string. For example, a union of two CHAR operands will still be a CHAR. However, if one of the character string operands will be converted such that the maximum expansion makes the length attribute the largest of the two operands, then the resulting character string length attribute is affected. For example, consider the result expressions of a CASE expression that have data types of VARCHAR(100) and VARCHAR(120). Assume the VARCHAR(100) expression is a mixed string host variable (that may require conversion) and the VARCHAR(120) expression is a column in the eucJP database. The resulting data type is VARCHAR(200) since the VARCHAR(100) is doubled to allow for possible conversion. The same scenario without the involvement of an eucJP or eucTW database would have a result type of VARCHAR(120).

Notice that the doubling of the host variable length is based on the fact that the database server is Japanese EUC or Traditional-Chinese EUC. Even if the client is also eucJP or eucTW, the doubling is still applied. This allows the same application package to be used by double-byte or multi-byte clients.

## Japanese and Traditional-Chinese EUC Considerations

### Rules for String Conversions

The types of operations listed in the corresponding section of the SQL Reference may convert operands to either the application or the database code page.

If such operations are done in a mixed code page environment that includes Japanese or Traditional-Chinese EUC, expansion or contraction of mixed character string operands may occur. Therefore the resulting data type has a length attribute that accomodates the maximum expansion, if possible. In the cases where there are restrictions on the length attribute of the data type, the maximum allowed length for the data type is used. For example in an environment where maximum growth is double, a VARCHAR(200) host variable is treated as if it is a VARCHAR(400), but CHAR(200) host variable is treated as if it is a CHAR(254). A run-time error may occur when conversion is performed if the converted string would exceed the maximum length for the data type. For example, the union of CHAR(200) and CHAR(10) would have a result type of CHAR(254). When the value from the left side of the UNION is converted, if more than 254 characters are required, an error occurs.

In some cases, allowing for the maximum growth for conversion will cause the length attribute to exceed a limit. For example, UNION only allows columns up to 254 bytes. Thus, a query with a union that included a host variable in the column list (call it :hv1) that was a DBCS mixed character string defined as a varying length character string 128 bytes long, would set the data type to VARCHAR(256) resulting in an error preparing the query, even though the query in the application does not appear to have any columns greater than 254. In a situation where the actual string is not likely to cause expansion beyond 254 bytes the following can be used to prepare the statement.

```
SELECT CAST(:hv1 CONCAT ' AS VARCHAR(254)), C2 FROM T1
UNION
SELECT C1, C2 FROM T2
```

The concatenation of the null string with the host variable will force the conversion to occur before the cast is done. This query can be prepared in the DBCS to eucJP/eucTW environment although a truncation error may occur at run-time.

This technique (null string concat with cast) can be used to handle the similar 254 byte limit for SELECT DISTINCT or use of the column in ORDER BY or GROUP BY clauses.

# Japanese and Traditional-Chinese EUC Considerations

## Constants

### Graphic String Constants

Japanese or Traditional-Chinese EUC client, may contain single or multi-byte characters (like a mixed character string). The string should not contain more than 2 000 characters. It is recommended that only characters that convert to double-byte characters in all related PC and EBCDIC double-byte code pages be used in graphic constants. A graphic string constant in an SQL statement is converted from the client code page to the double-byte encoding at the database server. For a Japanese or Traditional-Chinese EUC server, the constant is converted to UCS-2, the double-byte encoding used for graphic strings. For a double-byte server, the constant is converted from the client code page to the DBCS code page of the server.

## Functions

The design of user-defined functions should consider the impact of supporting Japanese or Tradition-Chinese EUC on the parameter data types. One part of function resolution considers the data types of the arguments to a function call. Mixed character string arguments involving a Japanese or Traditional-Chinese EUC client may require additional bytes to specify the argument. This may require that the data type change to allow the increased length. For example, it may take 4001 bytes to represent a character string in the application (a LONG VARCHAR) that fits into a VARCHAR(4000) string at the server. If a function signature is not included that allows the argument to be a LONG VARCHAR, function resolution will fail to find a function.

Some functions exist that do not allow long strings for various reasons. Use of LONG VARCHAR or CLOB arguments with such functions will not succeed. For example, LONG VARCHAR as the second argument of the built-in POSSTR function, will fail function resolution (SQLSTATE 42884).

## Expressions

### With the Concatenation Operator

The potential expansion of one of the operands of concatenation may cause the data type and length of concatenated operands to change when in an environment that includes a Japanese or Traditional-Chinese EUC database server. For example, with an EUC server where the value from a host variable may double in length, consider the following example.

```
CHAR200 CONCAT :char50
```

The column *CHAR200* is of type CHAR(200). The host variable *char50* is defined as CHAR(50). The result type for this concatenation operation would normally be CHAR(250). However, given an eucJP or eucTW database server,

## Japanese and Traditional-Chinese EUC Considerations

the assumption is that the string may expand to double the length. Hence *char50* is treated as a CHAR(100) and the resulting data type is VARCHAR(300). Note that even though the result is a VARCHAR, it will always have 300 bytes of data including trailing blanks. If the extra trailing blanks are not desired, define the host variable as VARCHAR(50) instead of CHAR(50).

## Predicates

### LIKE Predicate

For a LIKE predicate involving mixed character strings in an EUC database:
- single-byte underscore represents any one single-byte character
- single-byte percent represents a string of zero or more characters (single-byte or multi-byte characters).
- double-byte underscore represents any one multi-byte character
- double-byte percent represents a string of zero or more characters (single-byte or multi-byte characters).

The escape character must be one single-byte character or one double-byte character.

Note that use of the underscore character may produce different results depending on the code page of the LIKE operation. For example, Katakana characters in Japanese EUC are multi-byte characters (CS2) but in the Japanese DBCS code page they are single-byte characters. A query with the single-byte underscore in the *pattern-expression* would return occurrences of Katakana character in the position of the underscore from a Japanese DBCS server. However, the same rows from the equivalent table in a Japanese EUC server would not be returned, since the Katakana characters will only match with a double-byte underscore.

For a LIKE predicate involving graphic strings in an EUC database:
- the character used for underscore and percent must map to the underscore and percent character respectively
- underscore represents any one UCS-2 character
- percent represents a string of zero or more UCS-2 characters.

---

## Functions

### LENGTH

The processing of this function is no different for mixed character strings in an EUC environment. The value returned is the length of the string in the code page of the argument. When using this function to determine the length

of a value, careful consideration should be given to how the length is used. This is especially true for mixed string constants since the length is given in bytes, not characters. For example, the length of a mixed string column in a DBCS database returned by the LENGTH function may be less than the length of the retrieved value of that column on an eucJP or eucTW client due to the conversion of some DBCS characters to multi-byte eucJP or eucTW characters.

## SUBSTR

The SUBSTR function operates on mixed character strings on a byte basis. The resulting string may therefore include fragments of multi-byte characters at the beginning or end of the resulting string. No processing is provided to detect or process fragments of characters.

## TRANSLATE

The TRANSLATE function supports mixed character strings including multi-byte characters. The corresponding characters of the *to-string-exp* and the *from-string-exp* must have the same number of bytes and cannot end with part of a multi-byte character.

The *pad-char-exp* must result in a single-byte character when the *char-string-exp* is a character string. Since TRANSLATE is performed in the code page of the *char-string-exp*, the *pad-char-exp* may be converted from a multi-byte character to a single-byte character.

A *char-string-exp* that ends with part of a multi-byte character will not have those bytes translated.

## VARGRAPHIC

The VARGRAPHIC function on a character string operand in a Japanese or Traditional-Chinese EUC code page returns a graphic string in the UCS-2 code page.
- Single-byte characters are converted first to their corresponding double-byte character in the code set to which they belong (eucJP or eucTW). Then, they are converted to the corresponding UCS-2 representation. If there is no double-byte representation, the character is converted to the double-byte substitution character defined for that code set before being converted to UCS-2 representation.
- Characters from eucJP that are Katakana (eucJP CS2) are actually single byte characters in some encoding schemes. They are thus converted to corresponding double-byte characters in eucJP or to the double-byte substitution character before converting to UCS-2.
- Multi-byte characters are converted to their UCS-2 representations.

## Statements

### CONNECT

The processing of a successful CONNECT statement returns information in the SQLCA that is important when the possibility exists for applications to process data in an environment that includes a Japanese or Traditional-Chinese EUC code page at the client or server. The *SQLERRD(1)* field gives the maximum expansion of a mixed character string when converted from the application code page to the database code page. The *SQLERRD(2)* field gives the maximum expansion of a mixed character string when converted from the database code page to the application code page. The value is positive if expansion could occur and negative if contraction could occur. If the value is negative, the value is always -1 since the worst case is that no contraction occurs and the full length of the string is required after conversion. Positive values may be as large as 2, meaning that in the worst case, double the string length may be required for the character string after conversion.

The code page of the application server and the application client are also available in the SQLERRMC field of the SQLCA.

### PREPARE

The data types determined for untyped parameter markers (as described in Table 25 on page 826) are not changed in an environment that includes Japanese or Traditional-Chinese EUC. As a result, it may be necessary in some cases to use typed parameter markers to provide sufficient length for mixed character strings in eucJP or eucTW. For example, consider an insert to a CHAR(10) column. Preparing the statement:

```
INSERT INTO T1 (CH10) VALUES (?)
```

would result in a data type of CHAR(10) for the parameter marker. If the client was eucJP or eucTW, more than 10 bytes may be required to represent the string to be inserted but the same string in the DBCS code page of the database is not more than 10 bytes. In this case, the statement to prepare should include a typed parameter marker with a length greater than 10. Thus, preparing the statement:

```
INSERT INTO T1 (CH10) VALUES (CAST(? AS VARCHAR(20))
```

would result in a data type of VARCHAR(20) for the parameter marker.

# Appendix P. Glossary

## A

**abnormal termination.**  (1) The cessation of processing prior to planned termination. (T) (2) A system failure or operator action that causes a job to end unsuccessfully. (3) In DB2, exits that are not under program control, such as a trap or segv.

**access path.**  The method that is selected by the optimizer for retrieving data from a specific table. For example, an access path can involve the use of an index, a sequential scan, or a combination of the two.

**access plan.**  The set of access paths that are selected by the optimizer to evaluate a particular SQL statement. The access plan specifies the order of operations to resolve the execution plan, the implementation methods (such as JOIN), and the access path for each table referenced in the statement.

**accounting string.**  User-defined accounting information that is sent to DRDA servers by DB2 Connect. This information can be specified at one of these locations:
- The client workstation using the SQLESACT API or the DB2ACCOUNT environment variable
- The DB2 Connect workstation using the DFT_ACCOUNT_STR database manager configuration parameter.

**active log.**  The subset of files consisting of primary and secondary log files that are currently needed by the database manager for crash recovery and rollback. Contrast with archive log.

**adjacent nodes.**  Two nodes connected together by at least one path that connects no other nodes. (T)

**administrative authority.**  A level of authority that gives a user privileges over a set of objects. For example, DBADM authority gives privileges over all objects in a database, and SYSADM authority gives privileges over all objects in a system.

**ADSM.**  ADSTAR Distributed Storage Manager.

**ADSTAR Distributed Storage Manager (ADSM).**  A client/server product that provides storage management and data access services in a heterogeneous environment. It supports various communication methods, provides administrative facilities to manage the backup and storage of files, and provides facilities for scheduling backup operations.

**Advanced Peer-to-Peer Networking (APPN).**  An extension to SNA that features greater distributed network control, dynamic definition of network resources, and automated resource registration and directory lookup.

**Advanced Peer-to-Peer Networking (APPN) network.**  A collection of interconnected network nodes and their client end nodes. See also network node.

**Advanced program-to-program communication (APPC).**  The general facility that characterizes the LU 6.2 architecture and its various implementations in products.

**1189**

# Glossary

**after-image.** The updated content of a source table element that is recorded in a change data table or in a database log or journal. Contrast with before-image.

**agent.** A separate process or thread that carries out all DB2 requests that are made by a particular client application.

**aggregate function.** Synonym for column function.

**alert.** An action, such as a beep or warning, that is generated when a performance variable exceeds or falls below its warning or alarm threshold.

**alias.** In DB2, an alternative name used to identify a table, view, or database.

**alias chain.** A series of table aliases that refer to each other in a sequential, non-repeating fashion.

**alphanumeric.** Pertaining to data that consists of letters, digits, and usually other characters, such as punctuation marks. (T) (A)

**ambiguous cursor.** A cursor that cannot be determined to be updateable or read-only from its definition or context.

**API.** Application programming interface.

**APPC.** Advanced program-to-program communication.

**application ID.** A string that uniquely identifies an application across networks. An ID is generated at the time that the application connects to the database. This ID is known on both the client and the server and can be used to correlate the two parts of the application.

**application process.** The unit to which resources and locks are allocated. An application process involves the running of one or more programs.

**application programming interface (API).** (1) A functional interface supplied by the operating system or by a separately orderable licensed program. It allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed programs. (2) In DB2, a function within the interface. For example, the get error message API.

**application requester.** A facility that accepts a database request from an application process and passes it to an application server.

**application server.** The local or remote database manager to which the application process is connected.

**Apply program.** A replication program that is used to refresh or update a target table, depending on the applicable source-to-target rules. Contrast with Capture program.

**Apply qualifier.** A character string that identifies subscription definitions that are unique to each instance of the Apply program.

**Apply trail table.** A replication source table at the control server that records a history of the refreshes and updates performed against target tables.

**APPN.** Advanced Peer-to-Peer Networking.

# Glossary

**APPN network node.** Advanced Peer-to-Peer Networking (APPN) network node. Synonymous with network node.

**archive log.** The set of log files that are closed and are no longer needed for normal processing. These files are retained for use in roll-forward recovery. Contrast with active log.

**argument.** A value passed to or returned from a function or procedure at run time.

**asynchronous.** Without regular time relationship; unexpected and unpredictable with respect to the processing of program instructions. Contrast with synchronous.

**asynchronous batched update.** A process in which all changes to the source are recorded and applied to existing target data at specified intervals. Contrast with asynchronous continuous update.

**asynchronous continuous update.** A process in which all changes to the source are recorded and applied to existing target data after being committed in the base table. Contrast with asynchronous batched update.

**attach.** In DB2, to remotely access objects at the instance level.

**audit trail.** Data, in the form of a logical path linking a sequence of events, used for tracing the transactions that have affected the contents of a record. (T)

**authorization.** In computer security, the right granted to a user to communicate with or make use of a computer system. (T)

**authorization ID.** A character string in a statement that designates a set of privileges. It is used by the database manager for authorization checking and as an implicit qualifier for the names of objects such as tables, views, and indexes.

**autocommit.** To automatically commit the current unit of work after each SQL statement.

**automatic rebind.** A feature that automatically rebinds an invalidated package without requiring a **bind** command to be entered manually or a bind file to be present.

**auto-registration.** A process in which replica and consistent change data tables are automatically defined as replication sources at the target server.

# B

**backup pending.** The state of a database or table space that prevents an operation from being performed until a backup is made of the database or table space.

**base aggregate table.** A target table type that contains data aggregated from a source table or a point-in-time table at intervals.

**base table.** A table created with the CREATE TABLE statement. Such a table has both its description and data physically stored in the database. Contrast with view.

**basic conversation.** An LU 6.2 conversation type between two transaction programs using the APPC basic conversation API. Contrast with mapped conversation.

**basic predicate.** A predicate that compares two values.

## Glossary

**before-image.** The content of a source table element prior to a refresh or update, as recorded in a change data table, or in a database log or journal. Contrast with after-image.

**binary large object (BLOB).** A sequence of bytes, where the size of the sequence ranges from 0 bytes to 2 gigabytes. This string does not have an associated code page and character set. Image, audio, and video objects are stored in BLOBs.

**bind.** In SQL, the process by which the output from the SQL precompiler is converted to a usable structure called an access plan. During this process, access paths to the data are selected and some authorization checking is performed.

**bindery object name.** A 48-byte character string that contains the name of a bindery object on the NetWare** file server. The database manager configuration field, objectname, uniquely represents a DB2 server instance, and is stored as an object in the bindery on a NetWare file server.

**bind file.** A file produced by the precompiler when the **bind** command or API is used with the BINDFILE option. This file includes information on all SQL statements in the application program.

**bit data.** Data that is not associated with a coded character set and is therefore never converted.

**BLOB.** Binary large object.

**block.** A string of data elements recorded or transmitted as a unit.

**blocking.** An option that is specified when binding an application. It allows caching of multiple rows of information by the communications subsystem so that each FETCH statement does not require the transmission of one row for each request across the network.

**broadcast join.** A join in which all partitions of a table are sent to all nodes.

**built-in function.** An SQL function that is provided by DB2 and appears in the SYSIBM schema. Contrast with user-defined function.

**byte reversal.** A technique in which numeric data is stored with the least significant byte first.

## C

**cache.** A buffer that contains frequently accessed instructions and data; it is used to reduce access time.

**Call Level Interface (CLI).** A callable API for database access, which is an alternative to an embedded SQL API. In contrast to embedded SQL, CLI does not require precompiling or binding by the user, but instead provides a standard set of functions to process SQL statements and related services at run time.

**Capture program.** A replication program that reads database log or journal records to capture data about changes made to source tables. Contrast with Apply program.

**cardinality.** The number of rows in a database table.

**cascade rejection.** The process of rejecting a replication transaction because it is associated with a transaction that had a conflict detected and was itself rejected.

**Glossary**

**cast function.** A function used to convert instances of a data type (origin) into instances of a different data type (target). In general, cast functions have the name of the target data type. They have one single argument whose type is the origin data type; their return type is the target data type.

**catalog.** A set of tables and views maintained by the database manager. These tables and views contain information such as descriptions of tables, views, and packages.

**catalog node.** The node at which the catalog tables reside. The catalog node can be a different node for each database.

**CCSID.** Coded character set identifier.

**CDRA.** Character Data Representation Architecture.

**change aggregate table.** A type of target table that contains data aggregations based on changes recorded for a source table.

**change data (CD) table.** A replication control table at the source server that contains changed data for a replication source table. The Capture program populates the CD table by copying the changes from the database log or journal. The contents of the CD table are then copied by the Apply program to the target table.

**Character Data Representation Architecture (CDRA).** An architecture used to achieve consistent representation, processing, and interchange of string data.

**character large object (CLOB).** A sequence of characters (single-byte, multi-byte, or both) where the length can be up to 2 gigabytes. A data type that can be used to store large text objects. Also called character large object string.

**character string.** A sequence of bytes or characters.

**character string delimiter.** The characters used to enclose character strings in delimited ASCII files that are imported or exported. See delimiter.

**check condition.** A restricted form of search condition used in check constraints.

**check constraint.** Specifies a check condition that is not false for each row of the table on which the constraint has been defined.

**check pending.** A state into which a table can be put where only limited activity is allowed on the table and constraints are not checked when the table is updated.

**circular log.** A database log in which records are overwritten if they are no longer needed by an active database. Consequently, if a failure occurs, lost data cannot be restored during forward recovery. Contrast with recoverable log.

**CLI.** Call Level Interface.

**client.** Any program (or workstation it is running on) that communicates with and accesses a database server.

**CLOB.** Character large object.

# Glossary

**CLP.** Command Line Processor.

**clustered index.** An index whose sequence of key values closely corresponds to the sequence of rows stored in a table. The degree to which this correspondence exists is measured by statistics that are used by the optimizer.

**coded character set.** A set of unambiguous rules that establishes a character set and the one-to-one relationships between the characters of the set and their coded representations.

**coded character set identifier (CCSID).** A number that includes an encoding scheme identifier, character set identifiers, code page identifiers, and other information that uniquely identifies the coded graphic character representation.

**code page.** A set of assignments of characters to code points.

**code point.** In CDRA, a unique bit pattern that represents a character in a code page.

**code set.** Encoding values for a character set that provides the interface between the system and its input and output devices. ISO uses code set as the term equivalent to the IBM-defined term code page.

**cold start.** A system start, using an initial program load procedure. Contrast with warm start.

**collating sequence.** The sequence in which the characters are ordered for the purpose of sorting, merging, comparing, and processing indexed data sequentially.

**collocated join.** The result of two tables being joined in which:
* The tables reside in a single-partition nodegroup in the same database partition; or they are in the same partitioned nodegroup and have the same number of partitioning columns, the columns are partition compatible, and both tables use the same partitioning function.
* All pairs of the corresponding partitioning key columns participate in the equijoin predicates.

**column distribution value.** Statistics describing the most frequent values of some column or the quantile values. These values are used in the optimizer to help determine the best access plan.

**column function.** An operation used in queries that applies to the values from several rows. Column functions include SUM, AVG, MIN, MAX, COUNT, STDDEV, and VARIANCE. Synonymous with aggregate function.

**Command Line Processor (CLP).** A character-based interface for entering SQL statements and database manager commands.

**commit.** The operation that ends a unit of work by releasing locks so that the database changes made by that unit of work can be perceived by other processes. This operation makes the data changes permanent.

**commit point.** A point in time when data is considered to be consistent.

**common critical section table.** A replication control table at the source server that is used to establish concurrency control between the Capture and Apply programs and to prevent an update replication cycle.

# Glossary

**Common Programming Interface Communications (CPI-C).** An API for applications that require program-to-program communication, making use of SNA's LU 6.2 to create a set of interprogram services.

**common pruning control table.** A replication control table at the source server that coordinates the pruning of the change data and unit-of-work control tables. The values in this table indicate how much data has been replicated by the Apply program and can be safely pruned by the Capture program.

**common registrations table.** A replication control table at the source server that relates each source table or view to an associated change data table and consistent change data table, if applicable.

**common subscription columns table.** A replication control table that contains column details of target tables.

**common subscription events table.** A replication control table that defines the events that trigger replication, including the event name and time.

**common subscription set table.** A replication control table that defines the members of a subscription set including the set name, Apply qualifier, source server, target server, and status.

**common subscription statements table.** A replication control table used to store the optional SQL statements that can be run at the beginning or end of the set subscription cycle.

**common subscription targets member table.** A replication control table that maps the source and target table relationships within a subscription set.

**common table expression.** An expression that defines a result table with a name (qualified SQL identifier) that can be specified as a table name in any FROM clause in the fullselect that follows the WITH clause.

**comparison operator.** An infix operator used in comparison expressions. Comparison operators are ¬< (not less than), <= (less than or equal to), ¬= (not equal to), = (equal to), >= (greater than or equal to), > (greater than), and ¬> (not greater than).

**complete.** A table attribute indicating that the table contains a row for every primary key value of interest. As a result, a complete source table can be used to perform a refresh of a target table.

**composite key.** A key composed of more than one column from a database table.

**compound SQL statement.** A block of SQL statements that are executed in a single call to the application server.

**concurrency.** The shared use of resources by multiple interactive users or application processes at the same time.

**condensed.** A table attribute indicating that the table contains current data rather than a history of changes to the data. A condensed table includes no more than one row for each primary key value in the table. As a result, a condensed table can be used to supply current information for a refresh.

**conflict detection.** The process of detecting an out-of-date row in a replica that was updated by a user application. When a conflict is detected, the transaction that caused the conflict is rejected. See also enhanced conflict detection, standard conflict detection.

# Glossary

**connect.** In DB2, to access objects at the database level.

**connection.** (1) An association between an application process and an application server. (2) In data communications, an association established between functional units for conveying information.

**connection handle.** Within the CLI, the data object that contains information associated with a connection. This includes general status information, transaction status, and diagnostic information.

**consistent change data (CCD) table.** A replication table that is used for staging data, with four replication control columns. It can be one of the following types:
- An internal CCD table that is a join of the change data table and the unit-of-work table at the source server.
- A regular CCD table that is a copy of the internal CCD table on a remote server.
- An external source table that is not a DB2 origin table; it is manually updated with four replication columns and defined as a replication source table.

**consolidation replication.** A replication model in which the data from multiple source tables is replicated to a single target table. Contrast with fan-out replication.

**constraint.** A rule that limits the values that can be inserted, deleted, or updated in a table. See check constraint, referential constraint, and unique constraint.

**container.** See table space container.

**contention.** In the database manager, a situation in which a transaction attempts to lock a row or table that is already locked.

**Control Center.** A graphical interface that shows database objects (such as databases and tables) and their relationship to each other. From the Control Center you can perform the tasks provided by the DBA Utility, Visual Explain, and Performance Monitor tools.

**control point.** (1) In APPN, a component of a node that manages resources of that node and optionally provides services to other nodes in the network. Examples are a system services control point (SSCP) in a type 5 node, a physical unit control point (PUCP) in a type 4 node, a network node control point (NNCP) in a type 2.1 (T2.1) network node, and an end node control point (ENCP) in a T2.1 end node. An SSCP and an NNCP can provide services to other nodes. (2) A component of a T2.1 node that manages the resources of that node. If the T2.1 node is an APPN node, the control point is capable of engaging in control point-to-control point sessions with other APPN nodes. If the T2.1 node is a network node, the control point also provides services to adjacent end nodes in the T2.1 network. See physical unit.

**control privilege.** The authority to completely control an object. This includes the authority to access, drop, or alter an object, and the authority to extend or revoke privileges on the object to other users.

**control server.** The database location of the applicable subscription definitions and apply trail table.

**control table.** A table in which replication source and subscription definitions or other replication control information is stored.

**conversation.** In APPC, a connection between two transaction programs over a logical unit-logical unit (LU-LU) session that allows them to communicate with each other while processing a transaction.

**conversational transaction.** In APPC, two or more programs communicating using the services of logical units (LUs).

**conversation security.** In APPC, a process that allows validation of a user ID or group ID and password before establishing a connection.

**conversation security profile.** The set of user IDs or group IDs and passwords that are used by APPC for conversation security.

**Coordinated Universal Time (UTC).** Synonym for Greenwich Mean Time.

**coordinating agent.** The agent that is spawned when a request is received by the database manager from an application. It remains associated with the application during the life of the application. This agent coordinates subagents that work for the application. See also subagent.

**coordinator node.** The node to which the application originally connected and on which the coordinating agent resides.

**coordinator subsection.** The subsection of an application that starts other subsections (if any) and returns results to the application.

**copy table.** See target table.

**correlated reference.** Reference to a column of a table that is outside of a subquery.

**correlated subquery.** A subquery that contains a correlated reference to a column of a table that is outside of the subquery.

**correlation name.** An identifier designating a table or view within a single SQL statement. It can be defined in any FROM clause or in the first clause of an UPDATE or DELETE statement.

**country code.** When accessing the database, the country code of the application is used to determine the date and time presentation (display and print) formats. It is also used with the code page to determine the default collating sequence for the database.

**CP.** Control point.

**CPI-C.** Common Programming Interface Communications.

**CPI-C side information profile.** In SNA, the profile that specifies the conversation characteristics to use when allocating a conversation with a remote transaction program. The profile is used by local transaction programs that communicate through CPI Communications. It specifies the partner LU name (the name of the connection profile that contains the remote LU name), the mode name, and the remote transaction program name.

**CP name.** Control point name. A network-qualified name of a control point consisting of a network ID qualifier identifying the network to which the control point node belongs.

**crash recovery.** The process of recovering from an immediate failure.

**CS.** Cursor stability.

# Glossary

**current function path.** An ordered list of schema names used in the resolution of unqualified references to functions and data types. In dynamic SQL, the current function path is found in the CURRENT FUNCTION PATH special register. In static SQL, it is defined in the FUNCPATH option for PREP and BIND commands.

**cursor.** A named control structure used by an application program to point to a specific row within some ordered set of rows. The cursor is used to retrieve rows from a set. See also unambiguous cursor, ambiguous cursor.

**cursor stability (CS).** An isolation level that locks any row accessed by a transaction of an application while the cursor is positioned on the row. The lock remains in effect until the next row is fetched or the transaction is terminated. If any data is changed in a row, the lock is held until the change is committed to the database.

# D

**DARI.** Database Application Remote Interface. Former term for stored procedure.

**data area.** A memory area used by a program to hold information.

**database administrator (DBA).** An individual responsible for the design, development, operation, safeguarding, maintenance, and use of a database. (T)

**Database Application Remote Interface (DARI).** Former term for stored procedure.

**database client.** A workstation used to access a database residing on a database server.

**database connection services (DCS) directory.** A directory that contains entries for remote databases and the corresponding application requester used to access them.

**database directory.** A directory that contains database access information for all databases to which a client can connect.

**database engine.** The part of the database manager providing the base functions and configuration files needed to use the database.

**database log.** A set of primary and secondary log files consisting of log records that record all changes to a database. The database log is used to roll back changes for units of work that are not committed and to recover a database to a consistent state. See also primary log, secondary log.

**database managed space (DMS) table space.** A table space whose space is managed by the database. Contrast with system managed space (SMS) table space.

**database management system (DBMS).** Synonym for database manager.

**database manager.** A computer program that manages data by providing the services of centralized control, data independence, and complex physical structures for efficient access, integrity, recovery, concurrency control, privacy, and security.

**database manager instance.** A logical database manager environment similar to an image of the actual database manager environment. You can have several instances of the database manager product on the

same workstation. You can use these instances to separate the development environment from the production environment, tune the database manager to a particular environment, and protect sensitive information from a particular group of people.

**database object.** Anything that can be created or manipulated with SQL—for example, tables, views, indexes, packages, triggers, or table spaces.

**database partition.** A part of the database that consists of its own user data, indexes, configuration files, and transaction logs. Sometimes called a node or database node.

**database server.** A functional unit that provides database services for databases.

**database system monitor.** A collection of programming APIs that monitor performance and status information about the database manager, databases, and applications using the database manager and DB2 Connect.

**data definition language (DDL).** A language for describing data and its relationships in a database. Synonymous with data description language. (T)

**data description language.** Synonym for data definition language. (T)

**data enhancement.** The modification of data as it is copied between the base table and the target table, including:
- Subsetting of data
- Generating multiple copies of data
- Converting data, such as code translation and value substitution
- Combining data in base tables
- Producing sums or averages of data

**DataJoiner.** A separately available product that provides client applications integrated access to distributed data and provides a single database image of a heterogeneous environment. With DataJoiner, a client application can join data that is distributed across multiple database management systems (using a single SQL statement) or update a single remote data source, as if the data were local.

**data link control (DLC).** In SNA, the protocol layer that consists of the link stations that schedule data transfer over a link between two nodes and perform error control for the link.

**data manipulation language (DML).** A subset of SQL statements used to manipulate data.

**date.** A three-part value that designates a day, month, and year.

**date duration.** A DECIMAL(8,0) value that represents a number of years, months, and days.

**datetime value.** A value of the data type DATE, TIME, or TIMESTAMP.

**DB2 CLI.** DB2 Call Level Interface. An alternative SQL interface for the DB2 family of products that takes full advantage of DB2 capability.

**DB2 Connect.** A product that provides the function necessary (DRDA application requester support) for client applications to read and update data stored in DRDA application servers, such as other members of the DB2 family.

# Glossary

**DB2 SDK.** DB2 Software Developer's Kit.

**DB2 Software Developer's Kit (DB2 SDK).** A collection of tools that help developers create database applications.

**DB2UEXIT.** An optional, user-written executable program that the database manager invokes to move or retrieve archive log files.

**DBA.** Database administrator.

**DBA Utility.** A tool that lets DB2 users configure databases and database manager instances, manage the directories necessary for accessing local and remote databases, back up and recover databases or table spaces, and manage media on a system using a graphical interface. The tasks provided by this tool can be accessed from the Control Center.

**DBCLOB.** Double-byte character large object.

**DBCS.** Double-byte character set.

**DBMS.** Database management system.

**DBMS instance connection.** A logical connection between an application and an agent process or thread owned by a DB2 instance.

**DCE\*\*.** Distributed Computing Environment.

**DDL.** Data definition language.

**deadlock.** A condition under which a transaction cannot proceed because it is dependent on exclusive resources that are locked by some other transaction, which in turn is dependent on exclusive resources in use by the original transaction.

**deadlock detector.** A process within the database manager that monitors the states of the locks to determine if a deadlock condition has occurred. When a deadlock condition is detected, the detector stops one of the transactions involved in the deadlock. This transaction is rolled back and the other transactions proceed.

**delete rule.** A rule associated with a referential constraint that either restricts the deletion of a parent row or specifies the effect of such a deletion on the dependent rows.

**delimited identifier.** A sequence of characters enclosed within quotation marks ("). The sequence must consist of a letter followed by zero or more characters, each of which is a letter, digit, or the underscore character.

**delimiter.** A character or flag that groups or separates items of data.

**delimiter token.** A string constant, a delimited identifier, an operator symbol, or any of the special characters shown in syntax diagrams.

**dependent logical unit (DLU).** An LU that requires assistance from a system services control point (SSCP) to instantiate an LU-LU session.

**dependent row.** A row containing a foreign key that matches the value of a parent key in the parent row. The foreign key value thus represents a reference from the dependent row to the parent row.

**dependent table.** A table that is a dependent in at least one referential constraint.

**descendent row.** A row that is dependent on another row or a row that is a descendent of a dependent row.

**descendent table.** A table that is a dependent of another table or a descendent of a dependent table.

**deterministic function.** See not-variant function.

**device name.** A name reserved by the system, or a device driver that refers to a specific device.

**directed join.** A relational operation in which all of the rows in one or both of the joined tables are rehashed and directed to new database partitions based on the join predicate. If all of the partitioning key columns in a table participate in the equijoin predicates, the other table is rehashed; otherwise (if there is at least one equijoin predicate), both tables are rehashed.

**directory services.** A portion of the APPN protocols that maintains information about the location of resources in an APPN network.

**distinct type.** A user-defined data type that is internally represented as an existing type (its source type), but is considered to be a separate and incompatible type for semantic purposes.

**Distributed Computing Environment (DCE\*\*).** A set of services and tools that support the creation, use, and maintenance of distributed applications in a heterogeneous computing environment. DCE is independent of the operating system and network; it provides interoperability and portability across heterogeneous platforms.

**distributed directory database.** The complete listing of all the resources in the network as maintained in the individual directories scattered throughout an APPN network. Each node has a piece of the complete directory, but it is not necessary for any one node to have the entire list. Entries are created, modified, and deleted through system definition, operator action, automatic registration, and ongoing network search procedures. Synonymous with distributed network directory.

**distributed network directory.** Synonym for distributed directory database.

**distributed relational database.** A database whose tables are stored on different but interconnected computing systems.

**Distributed Relational Database Architecture (DRDA).** Architecture that defines formats and protocols for providing transparent access to remote data. DRDA defines two types of functions, the application requester function and the application server function.

**distributed unit of work (DUOW).** A unit of work that allows SQL statements to be submitted to multiple relational database management systems, but no more than one system per SQL statement.

**DLC.** Data link control.

**DLU.** Dependent logical unit.

**DML.** Data manipulation language.

## Glossary

**DMS table space.** Database managed space table space.

**DNS.** Domain name system.

**Domain Name.** A mechanism, in TCP/IP, that tracks the host names in a network. A domain name consists of a sequence of names separated by dots.

**Domain Name System.** The distributed database system used by TCP/IP to map human-readable machine names into IP addresses.

**double-byte character large object (DBCLOB).** A sequence of double-byte characters, where the size can be up to 2 gigabytes. A data type that can be used to store large double-byte text objects. Also called double-byte character large object string. Such a string always has an associated code page.

**double-byte character set (DBCS).** A set of characters in which each character is represented by two bytes.

**DRDA.** Distributed Relational Database Architecture.

**DUOW.** Distributed unit of work.

**dynamic SQL.** SQL statements that are prepared and run within a running program. In dynamic SQL, the SQL source is contained in host language variables rather than being coded into the program. The SQL statement might change several times while the program is running.

# E

**EBCDIC.** Extended binary-coded decimal interchange code. A coded character set of 256 8-bit characters.

**embedded SQL.** SQL statements coded within an application program. See static SQL.

**EN.** End node.

**encoding scheme.** A set of rules to represent character data.

**end node (EN).** In APPN, a node that supports sessions between its local control point (CP) and the CP in an adjacent network node.

**enhanced conflict detection.** Conflict detection that guarantees data integrity among all replicas and the origin table. The Apply program locks all replicas in the subscription set against further transactions, and begins detection after all changes made prior to locking have been captured. See also standard conflict detection, conflict detection.

**environment handle.** A handle that identifies the global context for database access. All data that is pertinent to all objects in the environment is associated with this handle.

**equijoin.** A join in which the predicate contains an equals operator. For example, T1.C1 = T2.C2.

**EUC.** Extended UNIX code.

**event monitor.** A database object for monitoring and collecting data on database activities over a period of time.

Glossary

**exclusive lock.**  A lock that prevents concurrently executing application processes from accessing database data.

**executable statement.**  An SQL statement that can be embedded in an application program, dynamically prepared and executed, or issued interactively.

**explain.**  To capture detailed information about the access plan that was chosen by the SQL compiler to resolve an SQL statement. The information describes the decision criteria used to choose the access plan.

**explainable statement.**  An SQL statement for which the explain operation can be performed. Explainable statements are SELECT, UPDATE, INSERT, DELETE, and VALUES.

**explained statement.**  An SQL statement for which an explain operation was performed.

**explained statistics.**  Statistics for a database object that was referenced in an SQL statement at the time the statement was explained.

**explain snapshot.**  A capture of the current internal representation of an SQL query and related information. This information is required by the Visual Explain tool.

**export.**  To copy data from database manager tables to a file using formats such as PC/IXF, DEL, WSF, or ASC. Contrast with import.

**exposed name.**  A correlation name, a table, or a view name specified in a FROM clause for which a correlation name is not specified.

**Extended UNIX Code (EUC).**  A protocol that can support sets of characters from 1 to 4 bytes in length. EUC is a means of specifying a collection of code pages rather than actually being a code page encoding scheme itself. This is the UNIX alternative to the PC double-byte (DBCS) code page encoding schemes.

**extent.**  An allocation of space, within a container of a table space, to a single database object. This allocation consists of multiple pages.

**extent map.**  A meta-data structure stored within a table space that records the allocation of extents to each object in the table space.

**external source table.**  A non-DB2 table that is manually updated to match the consistent change data table structure and defined as a replication source. See also consistent change data (CCD) table.

## F

**fan-out replication.**  A replication model in which data from one source table is copied to multiple target tables, thereby distributing the data to multiple locations. Contrast with consolidation replication.

**fast communication manager (FCM).**  A group of functions that provide internodal communication support.

**fenced.**  A type of user-defined function or stored procedure that is defined to protect the DBMS from modifications by the function. The DBMS is isolated from the function or stored procedure by a barrier. Contrast with not-fenced.

**file reference variable.**  A host variable that is used to indicate that data resides in a file on the client rather than in a client memory buffer.

# Glossary

**file server.** A workstation that runs the NetWare** operating system software and acts as a network server. DB2 uses the file server to store DB2 server address information, which a DB2 client retrieves to establish an IPX**/SPX client-server connection.

**filter.** A device or program that separates data, signals, or material in accordance with specified criteria. (A)

**First Failure Service Log.** A file (db2diag.log) that contains diagnostic messages, diagnostic data, alert information, and related dump information. This file is used by database administrators.

**fixed-length string.** A character or graphic string whose length is specified and cannot be changed. Contrast with varying-length string.

**flagger.** A precompiler option that identifies SQL statements in applications that do not conform to selected validation criteria (for example, the ISO/ANSI SQL92 Entry level standard).

**foreign key.** A key that is part of the definition of a referential constraint and that consists of one or more columns of a dependent table.

**foreign update.** An update that was applied to a target table and replicated to the local table.

**forward recovery.** A process used to roll forward a database or table space. It allows a restored database or table space to be rebuilt to a specified point in time by applying the changes recorded in the database log.

**fullselect.** A subselect, a values-clause, or a number of both that are combined by set operators.

**fully qualified LU name.** See network-qualified name.

**function.** A mapping, embodied as a program (the function body), invocable by means of zero or more input values (arguments) to a single value (the result).

**function body.** The piece of code that implements a function.

**function family.** A set of functions with the same function name. The context determines whether the usage refers to a set of functions within a particular schema, or all the relevant functions with the same name within the current function path.

**function invocation.** The use of a function together with any argument values being passed to the function body. The function is invoked by its name.

**function path.** An ordered list of schema names that restricts the search scope for unqualified function invocations and provides a final arbiter for the function selection process.

**function path family.** All the functions of the given name in all the schemas identified (or used by default) in the user's function path.

**function resolution.** The process, internal to the DBMS, for which a particular function instance is selected for invocation. The function name, the data types of the arguments, and the function path are used to make the selection. Synonymous with function selection.

**function selection.** Synonym for function resolution.

**function shipping.** The shipping of the subsections of a request to the specific node that contains the applicable data.

**function signature.** The logical concatenation of a fully qualified function name with the data types of all of its parameters. Each function in a schema must have a unique signature.

## G

**gap.** A situation in which the Capture program is not able to read a range of log or journal records and there is potential loss of change data.

**global table lock.** A table lock that is acquired on all nodes in a table's nodegroup.

**global transaction.** A unit of work in a distributed transaction processing environment in which multiple resource managers are required.

**grant.** To give a privilege or authority to an authorization ID.

**graphic character.** Synonym for DBCS character.

**graphic string.** A sequence of DBCS characters.

**group.** A logical organization of users that have IDs according to activity or resource access authority.

## H

**handle.** A variable that represents an internal structure within a software system.

**hash partitioning.** A partitioning strategy in which a hash function is applied to the partitioning key value to determine the database partition to which the row is assigned.

**hop.** In APPN, a portion of a route that has no intermediate nodes. A hop consists of only a single transmission group connecting adjacent nodes.

**host computer.** (1) In a computer network, a computer providing services such as computation, database access, and network control functions. (2) The primary or controlling computer in a multiple computer installation.

**host identifier.** A name declared in the host program.

**host language.** Any programming language in which you can embed SQL statements.

**host node.** In SNA, a subarea node that contains a system services control point (SSCP); for example, an IBM System/390 computer with MVS and VTAM.

**host program.** A program written in a host language that contains embedded SQL statements.

**host variable.** In an application host program, a variable referenced by embedded SQL statements. Host variables are programming variables in the application program and are the primary mechanism for transmitting data between tables in the database and application program work areas.

## Glossary

## I

**identifier.** One or more characters used to identify or name a data element and possibly to indicate certain properties of that data element. (A)

**ILU.** Independent logical unit.

**import.** To copy data from an external file, using formats such as PC/IXF, DEL, WSF or ASC, into database manager tables. Contrast with export.

**import utility.** Transactional utility that inserts user-supplied record data into a table.

**independent logical unit (ILU).** An LU that is able to activate an LU-LU session without assistance from a system services control point (SSCP). It does not have an SSCP-LU session. Contrast with dependent logical unit.

**index.** A set of pointers that are logically ordered by the values of a key. Indexes provide quick access to data and can enforce uniqueness on the rows in the table.

**index key.** The set of columns in a table used to determine the order of index entries.

**index partition.** The part of an index that is associated with a table partition at a given node. An index defined on a table is implemented by multiple index partitions, one per table partition.

**index sargable predicates.** Predicates that are applied to index entries, in index leaf pages, to reduce the number of index entries that qualify the SQL request. This helps reduce the number of data rows accessed.

**indicator variable.** A variable used to represent the null value in an application program. If the value for the selected column is null, a negative value is placed in the indicator variable.

**indoubt transaction.** Pertains to a transaction in which one phase of a two-phase commit completes successfully but the system fails before a subsequent phase can complete.

**initialization fullselect.** The first fullselect in a recursive common table expression that gets the direct children of the initial value from the source table.

**inner join.** A join method in which a column that is not common to all of the tables being joined is dropped from the resultant table. Contrast with outer join.

**inoperative package.** A package that cannot be used because one or more functions that it depends on have been dropped. Such a package must be explicitly rebound. Contrast with invalid package.

**inoperative trigger.** A trigger that is no longer usable. A trigger will be marked inoperative when an object on which it depends is dropped (or made inoperative) or a privilege on which it depends is revoked.

**inoperative view.** A view that is no longer usable. A view will be marked as inoperative when either:
- SELECT privilege on a table or view that the view is dependent on is revoked from the definer of the view.
- An object on which the view definition is dependent was dropped (or possibly made inoperative in the case of another view).

**insert rule.** A condition enforced by the database manager that must be met before a row can be inserted into a table.

**instance.** See database manager instance.

**intermediate network node.** In APPN, a node that is part of a route between an origin LU (OLU) and a destination LU (DLU) but that neither contains the OLU or the DLU nor serves as the network server for either the OLU or DLU.

**internal CCD table.** A consistent change data table that is a join of the change data table and the unit-of-work table at the source server.

**Internet Protocol (IP).** A protocol used to route data from its source to its destination in an Internet environment.

**Internetwork Packet Exchange\*\* (IPX\*\*).** A connectionless datagram protocol, used in a NetWare\*\* LAN environment, to transfer data to a remote node. IPX makes a best-effort attempt to send data packets, but does not guarantee reliable delivery of the data.

**inter-partition parallelism.** Refers to the ability to perform multiple database operations (such as index creation, database load, and SQL queries) at the same time across multiple partitions of a partitioned database. Contrast with intra-partition parallelism.

**Inter-Process Communication (IPC).** A mechanism of an operating system that allows processes to communicate with each other.

**intra-partition parallelism.** Refers to the ability to perform multiple database operations (such as index creation, database load, SQL queries) at the same time within a single database partition. Contrast with inter-partition parallelism.

**intra-query parallelism.** Processing parts of a single query at the same time using either intra-partition parallelism, inter-partition parallelism, or both.

**invalid package.** A package that becomes invalid when an object that the package depends on is dropped. (The object is of a type other than function; for example, index.) Such a package is implicitly rebound upon invocation. Contrast with inoperative package.

**I/O parallelism.** See parallel I/O.

**IP.** Internet Protocol.

**IPX\*\*.** Internetwork Packet Exchange\*\*.

**isolation level.** An attribute that defines the degree to which an application process is isolated from other concurrently executing application processes.

# J

**job scheduler.** A program used to automate certain tasks for running and managing database jobs.

**join.** A relational operation that allows for retrieval of data from two or more tables based on matching column values.

## Glossary

## K

**key.** A column or an ordered collection of columns that are identified in the description of a table, index, or referential constraint.

**key-value based partitioning strategy.** A strategy for assigning rows in a table to database partitions. Rows are assigned based on the values of the partitioning key columns.

**keyword.** One of the predefined words of a computer, command language, or an application.

## L

**labeled duration.** A number that represents a duration of years, months, days, hours, minutes, seconds, or microseconds.

**large object (LOB).** A sequence of bytes, where the length can be up to 2 gigabytes. It can be any of three types: BLOB (binary), CLOB (single-byte character or mixed) or DBCLOB (double-byte character).

**length attribute.** A value associated with a string that represents the declared fixed length or maximum length of the string.

**LEN node.** Low-entry networking node.

**list prefetch.** An access method that takes advantage of prefetching even in queries that do not access data sequentially. This is done by scanning the index and collecting RIDs in advance of accessing any data pages. These RIDs are then sorted, and data is prefetched using this list.

**load copy.** A backup image of data that was loaded at a previous time and can be restored during roll-forward recovery.

**load utility.** A nontransactional utility that performs block updates of table data. Contrast with import utility.

**LOB.** Large object.

**LOB locator.** A mechanism that allows an application program to manipulate a large object value in the database system. A LOB locator is a simple token value that represents a single LOB value. An application program retrieves a LOB locator into a host variable and can then apply SQL functions to the associated LOB value via the locator.

**local database.** A database that is physically located on the workstation in use. Contrast with remote database.

**local database directory.** A directory where a database physically resides. Databases that are displayed in the local database directory are located on the same node as the system database directory.

**local table lock.** A table lock that is acquired only on a single database partition.

**local update.** An update to the base table, not to the replica.

**locator.** See LOB locator.

**Glossary**

**lock.** (1) A means of serializing events or access to data. (2) A means of preventing uncommitted changes made by one application process from being perceived by another application process and for preventing one application process from updating data that is being accessed by another process.

**lock escalation.** In the database manager, the response that occurs when the number of locks issued for one agent exceeds the limit specified in the database configuration; the limit is defined by the MAXLOCKS configuration parameter. During a lock escalation, locks are freed by converting locks on rows of a table into one lock on a table. This is repeated until the limit is no longer exceeded.

**locking.** The mechanism used by the database manager to ensure the integrity of data. Locking prevents concurrent users from accessing inconsistent data.

**log.** (1) See database log. (2) A file used to record changes made in a system.

**log head.** The oldest written log record in the active log.

**logical node.** A node on a processor when more than one node is assigned to that processor. See also node.

**logical operator.** A keyword that specifies how multiple search conditions are to be evaluated (AND, OR) or if the logical sense of a search condition is to be inverted (NOT).

**logical unit (LU).** In SNA, a port through which an end user accesses the SNA network to communicate with another end user. An LU may be capable of supporting many sessions with other LUs.

**logical unit 6.2 (LU 6.2).** The LU type that supports sessions between two applications using APPC.

**log partition.** The log file on each database partition that records database activity for that database partition.

**log record.** A record of an update to a database performed during a unit of work. This record is written after the log tail of the active log.

**log tail.** The log record that was written most recently in an active log.

**long string.** A varying-length string whose maximum length is greater than 254 bytes.

**long table space.** A table space that can store only long string or large object (LOB) data.

**low-entry networking node (LEN node).** A type 2.1 node that supports independent LU protocols but does not support CP-CP sessions. It can be a peripheral node attached to a boundary node in a subarea network, an end node attached to an APPN network node in an APPN network, or a peer-connected node directly attached to another LEN node or APPN end node.

**LU.** Logical unit.

**LU 6.2.** See logical unit 6.2.

**LU type.** The classification of a logical unit in terms of the specific subset of SNA protocols and options it supports for a given session, specifically:
- The values allowed in the session activation request

## Glossary

- The usage of data stream controls, function management headers, request unit parameters, and sense data values
- Presentation services protocols such as those associated with function management headers

## M

**mapped conversation.** In APPC, a conversation between two transaction programs (TPs) using the APPC mapped conversation API. In typical situations, end-user TPs use mapped conversation and service TPs use basic conversations. However, either type of program can use either type of conversation. Contrast with basic conversation.

**masking character.** A character used to represent optional characters at the front, middle, and end of a search term. Masking characters are normally used for finding variations of a term in a precise index.

**MBCS.** Multi-byte character set.

**migration.** (1) The process of moving data from one computer system to another without converting the data. (2) Installation of a new version or release of a program to replace an earlier version or release.

**mixed-character string.** A string containing a mixture of single-byte and multi-byte characters.

**mixed-data string.** Synonym for mixed-character string.

**mobile client.** The node, usually a lap top computer, where the mobile enabler and replication source and target tables used in a mobile environment are located. The mobile replication mode is invoked from the mobile client.

**mobile replication enabler.** A replication program that starts the mobile replication mode at the mobile client.

**mobile replication mode.** A mode of replication in which the Capture and Apply programs operate as needed rather than autonomously and continuously. This mode is invoked from the mobile client and allows data to be replicated when the mobile client is available for a connection to the source or target server.

**modelled statistics.** Statistics for a database object that may or may not be referenced in an SQL statement, yet currently exist in an explain model. The object may or may not currently exist in the database.

**mode name.** In APPC, the name used by the initiator of a session to designate the characteristics desired for the session, such as message length limits, sync point, class of service within the transport network, and session routing and delay characteristics.

**monitoring session.** The act of monitoring a database manager or of playing back information from a previously monitored database manager. The DB2 Performance Monitor is used for creating a monitoring session and for selecting which database objects to monitor.

**monitor switch.** Database manager parameters manipulated by the user to control the type of information and the quantity of information returned in performance snapshots.

**multi-byte character set (MBCS).** A set of characters in which each character is represented by two or more bytes. Those using only two bytes are more commonly known as double-byte character sets.

**multitasking.** A mode of operation that provides for concurrent performance, or interleaved execution of two or more tasks. (I) (A)

# N

**NAU.** Network addressable unit.

**NDS.** Network Directory Services.

**nested table expression.** A result table obtained directly or indirectly from one or more other tables through the evaluation of a fullselect that is specified in the FROM clause.

**NETID.** Network identifier. Synonym for network name.

**network address.** An identifier for a node in a network.

**network addressable unit (NAU).** A logical unit (LU), physical unit (PU), control point (CP), or system services control point (SSCP). It is the origin or the destination of information transmitted by the path control network. See also network name.

**Network Directory Services (NDS).** A global, distributed, replicated database introduced in NetWare** Version 4.0 that maintains information about, and provides access to, every resource on the network. The NetWare Directory database organizes objects, independent of their physical location, in a hierarchical tree structure called the directory tree.

**network name.** In SNA, a symbolic name by which end users refer to a network addressable unit (NAU), a link station, or a link. Synonymous with NETID.

**network node (NN).** In APPN, a node on the network that provides distributed directory services, topology database exchanges with other APPN network nodes, and session and routing services. Synonym for APPN network node.

**network node server.** An APPN network node that provides network services for its local logical units and adjacent end nodes.

**network-qualified name.** The name by which an LU is known throughout an interconnected SNA network. A network-qualified name consists of a network name identifying the individual subnetwork, and a network LU name. Network-qualified names are unique throughout an interconnected network. Also known as the network-qualified LU name, or fully qualified LU name.

**network services.** The services within network addressable units that control network operation through SSCP-SSCP, SSCP-PU, SSCP-LU, and CP-CP sessions.

**NN.** Network node.

**node.** (1) In database partitioning, synonymous with database partition. (2) In hardware, a uniprocessor or symmetric multiprocessor (SMP) computer that is part of a clustered system or a massively parallel processing (MPP) system. For example, RS/6000 SP is an MPP system that consists of a number of nodes connected by a high-speed network. (3) In communications, an end point of a communications link, or a junction common to two or more links in a network. Nodes can be processors, communication controllers, cluster controllers, terminals, or workstations. Nodes can vary in routing and other functional capabilities.

## Glossary

**node directory.**  A directory that contains information necessary to establish communications from a client workstation to all applicable database servers.

**nodegroup.**  A named group of one or more database partitions.

**noncondensed attribute.**  A table attribute indicating that the table contains a history of changes to the data, not current data. A table that has this attribute set includes more than one row for each key value.

**nondelimited ASCII (ASC) format.**  A file format used to import data. Nondelimited ASCII is a sequential ASCII file with row delimiters used for data exchange with any ASCII product.

**normalization.**  In databases, the process of restructuring a data model by reducing its relations to their simplest forms. (T)

**not-fenced.**  A type of user-defined function or stored procedure that is defined to be run in the DBMS process. Contrast with fenced.

**not-variant function.**  A user-defined function whose result is solely dependent on the values of the input arguments. Successive invocations with the same argument values always produce the same results. Contrast with variant function.

**nullable.**  The condition where a value for a column, function parameter, or result can have an absence of a value. For example, a field for a person's middle initial does not require a value.

**null value.**  A parameter position for which no value is specified.

## O

**object.**  (1) Anything that can be created or manipulated with SQL—for example, tables, views, indexes, or packages. (2) In object-oriented design or programming, an abstraction consisting of data and operations associated with that data. (3) For NetWare**, an entity that is defined on the network and thus given access to the file server.

**object property.**  A property that identifies a category of information associated with an object. A NetWare** bindery object can be assigned one or more properties. The DB2 server instance object has an object property NET_ADDR, which denotes the location of the record within the object.

**object type.**  (1) A 2-byte number that classifies an object in the bindery on a NetWare** file server. 062B represents the DB2 database server object type. (2) A categorization or grouping of object instances that share similar behaviors and characteristics.

**ODBC.**  Open Database Connectivity.

**ODBC driver.**  A driver that implements ODBC function calls and interacts with a data source.

**offline backup.**  A backup of the database or table space that was made when the database or table space was not being accessed by applications. The Backup Database utility acquires exclusive use of the database until the backup is complete. Contrast with online backup.

**offline restore.**  A restoration of a copy of a database or table space from a backup. The Backup Database utility has exclusive use of the database until the restore is completed.

**online backup.** A backup of the database or table space that is made while the database or table space is being accessed by other applications. Contrast with offline backup.

**online monitor.** See Performance Monitor.

**online restore.** A restoration of a copy of a database or table space while the database or table space is being accessed by other applications. Contrast with offline restore.

**Open Database Connectivity (ODBC).** An API that allows access to database management systems using callable SQL, which does not require the use of a SQL preprocessor. The ODBC architecture allows users to add modules, called database drivers, that link the application to their choice of database management systems at run time. Applications do not have to be linked directly to the modules of all the supported database management systems.

**operand.** An entity on which an operation is performed.

**optimized SQL text.** SQL text, produced by the Explain facility, based on the query actually used by the optimizer to choose the access plan. This query has been supplemented and rewritten by the various components of the SQL compiler during statement compilation. The text has been reconstructed from its internal representation, and differs from the original SQL text. The optimized statement produces the same result as the original statement.

**optimizer.** A component of the SQL compiler that chooses an access plan for a DML statement by modelling the execution cost of many alternative access plans and choosing the one with the minimal estimated cost.

**ordinary identifier.** In SQL, a letter, which might be followed by zero or more characters, each of which is a letter (a-z and A-Z), a symbol, a number, or the underscore character, used to form a name.

**ordinary token.** A numeric constant, an ordinary identifier, a host identifier, or a keyword.

**outer join.** A join method in which a column that is not common to all of the tables being joined becomes part of the resultant table. Contrast with inner join.

**output file.** A database or device file that is opened with the option to allow the writing of records.

**overflow record.** (1) On an indirectly addressed file, a record whose key is randomized to the address of a full track or to the address of a home record. (2) In DB2, an updated record that is too large to fit on the page it is currently stored in. The record is copied to a different page and its original location is replaced with a pointer to the new location. (3) In Database Monitor, a record inserted in the event monitor data stream to indicate that records were discarded because the named pipe was full and records were not processed in time. An overflow record indicates how many records were discarded.

**overloaded function name.** A function name for which there are multiple functions within a function path or schema. Those within the same schema must have different signatures.

# P

**package.** A control structure produced during program preparation that is used to execute SQL statements.

## Glossary

**packet.** In data communication, a sequence of binary digits, including data and control signals, that is transmitted and switched as a composite whole.

**page.** A block of storage within a table or index whose size is 4096 bytes (4 K).

**parallel I/O.** Refers to the process of reading from or writing to two or more I/O devices at the same time to reduce response time.

**parallelism.** The ability to perform multiple database operations at the same time (in parallel). See inter-partition parallelism, intra-partition parallelism, and parallel I/O.

**parallel session.** In SNA, two or more concurrently active sessions between the same two logical units (LUs). Each session can have different session parameters. Contrast with session.

**parameterized data type.** A data type that can be defined with a specific length, scale, or precision. These are the string and decimal data types.

**parameter marker.** A question mark (?) that appears in a statement string of a dynamic SQL statement. The question mark can appear where a host variable could appear if the statement string was a static SQL statement.

**parent key.** A primary key or unique key that is used in a referential constraint. The values of a parent key determine the valid values of the foreign key in the constraint.

**parent row.** A row that has at least one dependent row.

**parent table.** A table that is a parent in at least one referential constraint.

**partition compatible join.** A join where all of the rows that are joined reside in the same database partition.

**partitioned database.** A database with two or more database partitions. Data in user tables can be located in one or more database partitions. When a table is on multiple partitions, some of its rows are stored in one partition and others are stored in other partitions. See database partition.

**partitioning function.** A function that takes a partitioning key value of a row as input and produces a partition number as output.

**partitioning key.** (1) An ordered set of one or more columns in a given table. For each row in the table, the values in the partitioning key columns are used to determine on which database partition the row belongs. (2) In replication, an ordered set of one or more columns in a given table. For each row in the source table, the values in the partitioning key columns are used to determine in which target table the row belongs.

**partitioning map.** A vector of partition numbers that maps a partitioning map index to database partitions in the nodegroup.

**partitioning map index.** A number assigned to a hash partition or range partition.

**partner logical unit (LU).** In SNA, the remote participant in a session.

**peer-to-peer communication.** Communication between two SNA logical units (LUs) that is not managed by a host; commonly used when referring to LU 6.2 nodes.

# Glossary

**performance metrics.** A collection of all performance variables belonging to the same database object.

**Performance Monitor.** A tool that lets database administrators use a graphical interface to monitor the performance of a DB2 system for tuning purposes. The tasks provided by this tool can be accessed from the Control Center.

**performance snapshot.** Performance data for a set of database objects that is retrieved from the database manager at a point in time.

**performance variable.** A statistic derived from performance data obtained from the database manager. The expression for this variable can be user-defined.

**performance variable profile.** A flat file that contains definitions of performance variables. This file can be edited, copied, and shared. Different profiles can be used by the same Performance Monitor so that different calculations can be performed.

**phantom row.** A table row that can be read by application processes that are executing with any isolation level except repeatable read. When an application process issues the same query multiple times within a single unit of work, additional rows can appear between queries because of the data being inserted and committed by application processes that are running concurrently.

**physical unit.** The component that manages and monitors the resources (such as attached links and adjacent link stations) associated with a node, as requested by an SSCP through an SSCP-PU session. An SSCP activates a session with the PU in order to indirectly manage, through PU, resources of the node such as attached links. This term applies to types 2.0, 4, and 5 nodes only. See also control point.

**point-in-time table.** A type of target table whose content matches all or part of a source table, with an added system column that identifies the approximate time when the particular row was inserted or updated at the source system.

**point of consistency.** A point in time when all the recoverable data a program accesses is consistent. The point of consistency occurs when updates, inserts, and deletions are either committed to the physical database or rolled back. Synonym for commit point and sync point. See also rollback.

**precision.** In numeric data types, the total number of binary or decimal digits excluding the sign.

**precompile.** To process programs containing SQL statements before they are compiled. SQL statements are replaced with statements that will be recognized by the host language compiler. The output from a precompile includes source code that can be submitted to the compiler and used in the bind process.

**predicate.** An element of a search condition that expresses or implies a comparison operation.

**prefetch.** To read data ahead of, and in anticipation of, its use.

**prepare.** To convert an SQL statement from text form to an executable form, by submitting it to the SQL compiler.

**primary key.** A unique key that is part of the definition of a table. A primary key is the default parent key of a referential constraint definition.

**primary log.** A set of one or more log files used to record changes to a database. Storage for these files is allocated in advance. Contrast with secondary log.

# Glossary

**privilege.** The right to access a specific database object in a specific way. These rights are controlled by users with SYSADM (system administrator) authority or DBADM (database administrator) authority or by creators of objects. Privileges include rights such as creating, deleting, and selecting data from tables.

**procedure.** Synonym for stored procedure.

**protocol.ini.** A file that contains LAN configuration and binding information for all the protocol and medium access control (MAC) system modules.

**PU.** Physical unit.

**public authority.** The authority for an object granted to all users.

**PU type.** In SNA, the classification of a physical unit (PU) according to the type of node on which it resides.

## Q

**quantified predicate.** A predicate that compares a value with a set of values.

**query.** A request for information from the database based on specific conditions; for example, a request for a list of all customers in a customer table whose balance is greater than $1000.

**quiesce.** To end a process by allowing operations to complete normally, while rejecting any new requests for work.

**quoted name.** Synonym for delimited identifier.

## R

**RDBMS.** Relational database management system.

**read stability (RS).** An isolation level that locks only those rows that an application retrieves within a transaction. It ensures that any qualifying row read during a transaction is not changed by other application processes until the transaction is completed, and that any row changed by another application process is not read until the change is committed by that process. Read stability allows more concurrency than repeatable read, and less than cursor stability.

**rebind.** To create a new package for an application program that was previously bound. For example, if an index is added for a table accessed by a program, the package must be rebound for it to take advantage of the new index.

**record.** The storage representation of a single row of a table.

**recording.** The information from performance snapshots that can be viewed at a later time.

**recoverable log.** A database log in which all log records are retained so that, in the event of a failure, lost data can be recovered during forward recovery. Contrast with circular log.

**recovery.** (1) The act of resetting a system, or data that is stored in a system, to an operable state following damage. (2) The process of rebuilding databases by restoring a backup and rolling forward the logs associated with it.

## Glossary

**recovery log.** See database log.

**recovery pending.** A state of the database or table space. A database or table space is put in recovery pending state when it is restored from a backup. While the database or table space is in this state, its data cannot be accessed.

**recursion cycle.** The cycle that occurs when a fullselect within a common table expression includes the name of the common table expression in a FROM clause.

**recursive common table expression.** A common table expression that refers to itself in a FROM clause from the fullselect. Recursive common table expressions are used to write recursive queries.

**recursive query.** A fullselect that uses a recursive common table expression.

**referential constraint.** The referential integrity rule that the nonnull values of the foreign key are valid only if they also appear as values of a parent key.

**referential integrity.** The state of a database in which all values of all foreign keys are valid.

**refresh.** A process in which all of the data of interest in a user table is copied to the target table, replacing existing data.

**regular table space.** A table space that can store any nontemporary data.

**rejected transaction.** A transaction containing one or more updates from replica tables that are out of date in comparison to the origin table.

**relational database.** A database that can be perceived as a set of tables and manipulated in accordance with the relational model of data.

**remote database.** A database that is physically located on a workstation other than the one in use. Contrast with local database.

**remote unit of work (RUOW).** A unit of work that allows for the remote preparation and execution of SQL statements.

**repeatable read (RR).** An isolation level that locks all the rows in an application that are referenced within a transaction. When a program uses repeatable read protection, rows referenced by the program cannot be changed by other programs until the program ends the current transaction.

**replica.** A type of target table that can be updated locally and receives updates from a user table through a subscription definition. It can be a source for updating the user table or read-only target tables.

**replication.** The process of taking changes that are stored in the database log or journal at the source server and applying them to the target server.

**replication administrator.** The user responsible for defining replication sources and subscriptions. This user can also run the Capture and Apply programs.

**replication source.** A database table that is defined as a source for replication. This type of table can accept copy requests and is the source table in a replication subscription set. See also subscription set.

# Glossary

**replication subscription.** A specification for copying changed data from replication sources to target tables at a specified time and frequency, with the option of enhancing data. It defines all of the information that is required by the Apply program to copy data.

**reserved word.** (1) In programming languages, a keyword that may not be used as an identifier. (I) (2) A word used in a source program to describe an action to be taken by the program or compiler. It must not appear in the program as a user-defined name or a system name. (3) A word that has been set aside for special use in the SQL standard.

**restore.** To return a backup copy to the active storage location for use.

**restore set.** A backup copy of a database or table space plus zero or more log files that, when restored and rolled forward, bring the database or table space back to a consistent state.

**result set.** Synonym for result table.

**result table.** The set of rows produced by the evaluation of a SELECT statement. Synonymous with result set.

**revoke.** To remove a privilege or authority from an authorization ID.

**RID (Record ID).** A number that is used internally by DB2 to uniquely identify a record in a table. It contains enough information to address the page in which the record is stored.

**rollback.** The process of restoring data changed by SQL statements to the state at its last commit point. See also point of consistency.

**roll-forward.** The process of updating the data in a restored database by applying changes recorded in the database log. See forward recovery.

**row.** The horizontal component of a table consisting of a sequence of values, one for each column of the table.

**RR.** Repeatable read.

**RS.** Read stability.

**RUOW.** Remote unit of work.

# S

**sargable.** A predicate that can be evaluated as a search argument. See predicate.

**SBCS.** Single-byte character set.

**scalar fullselect.** A fullselect that returns a single value— one row of data consisting of exactly one column.

**scalar function.** An SQL operation that produces a single value from another value and is expressed as a function name followed by a list of arguments enclosed in parentheses.

**scale.** The number of digits in the fractional part of a number.

**schema.** A collection of database objects such as tables, views, indexes, or triggers. It provides a logical classification of database objects.

**SDK.** Software Developer's Kit.

**search condition.** A criterion for selecting rows from a table. A search condition consists of one or more predicates.

**secondary log.** A set of one or more log files used to record changes to a database. Storage for these files is allocated as needed when the primary log is full.

**self-referencing row.** A row that is a parent of itself.

**self-referencing subquery.** A subselect or fullselect within a DELETE, INSERT, or UPDATE statement that refers to the same table that is the object of the SQL statement.

**self-referencing table.** A table that is both a parent and a dependent table in the same referential constraint.

**server.** In a network, a node that provides facilities to other stations; for example, a file server, a printer server, a mail server.

**service name.** A name that provides a symbolic method of specifying the port number to be used at a remote node. The TCP/IP connection requires:
- The address of the remote node.
- The port number to be used on the remote node to identify an application.

**session.** A logical connection between two stations or SNA network addressable units (NAUs) that allows the two stations or NAUs to communicate.

**session limit.** In SNA, the maximum number of concurrently active logical unit-logical unit (LU-LU) sessions that a particular logical unit (LU) can support.

**session partner.** In SNA, one of the two network addressable units (NAUs) participating in an active session.

**session security.** For LU 6.2, partner LU verification and session data encryption. A Systems Network Architecture (SNA) function that allows data to be transmitted in encrypted form.

**set operator.** The SQL operators UNION, EXCEPT, and INTERSECT corresponding to the relational operators union, difference, and intersection. A set operator derives a result table by combining two other result tables.

**shadowing.** A recovery technique in which current page contents are never overwritten. Instead, new pages are allocated and written while the pages whose values are being replaced are retained as shadow copies until they are no longer needed to support the restoration of the system state due to a transaction rollback.

**share lock.** A lock that limits concurrently executing application processes to read-only operations on database data.

**short string.** A fixed-length string or a varying-length string whose maximum length is less than or equal to 254 bytes.

# Glossary

**single-byte character set (SBCS).** A character set in which each character is represented by a one-byte code. Contrast with double-byte character set, multi-byte character set.

**SMS table space.** System managed space table space.

**SNA.** Systems Network Architecture.

**SNA network.** The part of the user application network that conforms to the formats and protocols of Systems Network Architecture (SNA). It enables reliable transfer of data among users and provides protocols for controlling the resources of various network configurations. The SNA network consists of network addressable units (NAUs), gateway function, intermediate session routing function components, and the transport network.

**snapshot.** See performance snapshot, explain snapshot.

**Performance Monitor.** An API used to assess the status of the database manager, individual databases, tables, table spaces, and individual applications.

**soft checkpoint.** The process of writing some information to the log file header; this information is used to determine the starting point in the log in case a database restart is required.

**Software Developer's Kit (SDK).** An application development product that allows applications to be developed on a client workstation to access remote database servers including host relational databases through the DB2 Connect products.

**source function.** A user-defined function (UDF) whose body is used to implement one or more other UDFs.

**source server.** The database location of the replication source and the Capture program.

**source table.** A table that contains the data that is to be copied to a target table. The source table can be a replication source table, a change data table, or a consistent change data table. Contrast with target table.

**source type.** An existing type that is used to internally represent a distinct type.

**special register.** A storage area that is defined for an application process by the database manager and is used to store information that can be referenced in SQL statements. Examples are USER and CURRENT DATE.

**specific function name.** The name that uniquely identifies a function to the system.

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

**SQLCA.** SQL communication area.

**SQL communication area (SQLCA).** A set of variables that provides an application program with information about the execution of its SQL statements or its requests from the database manager.

**SQLDA.** SQL descriptor area.

**SQL descriptor area (SQLDA).** A set of variables that is used in the processing of certain SQL statements. The SQLDA is intended for dynamic SQL programs.

# Glossary

**SSCP.** System services control point.

**stack.** An area in memory that stores temporary register information, parameters, and return addresses of subroutines.

**staging table.** A consistent change target table that is used as the source for updating data to multiple target tables.

**standard conflict detection.** Conflict detection in which the Apply program searches for conflicts in rows that are already captured in the replica's change data tables. See also conflict detection, enhanced conflict detection.

**statement.** An instruction in a program or procedure.

**statement handle.** In CLI, a handle that refers to the data object that contains information about an SQL statement. This includes information such as dynamic arguments, bindings for dynamic arguments and columns, cursor information, result values, and status information. Each statement handle is associated with a connection handle.

**static SQL.** SQL statements that are embedded within a program, and are prepared during the program preparation process before the program is executed. After being prepared, a static SQL statement does not change, although values of host variables specified by the statement may change.

**stored procedure.** A block of procedural constructs and embedded SQL statements that is stored in a database and can be called by name. Stored procedures allow an application program to be run in two parts. One part runs on the client and the other on the server. This allows one call to produce several accesses to the database. Synonymous with procedure.

**string.** In programming languages, the form of data used for storing and manipulating text.

**Structured Query Language.** See SQL.

**subagent.** A type of agent that works on subrequests. A single application can make many requests, and each request can be broken into many subrequests. Therefore, there can be multiple subagents working on behalf of the same application. All subagents working for the application are coordinated by the coordinating agent for that application. See also coordinating agent.

**subordinate agent.** See subagent.

**subquery.** A fullselect used within a search condition of an SQL statement.

**subscription cycle.** The processing of a subscription set by the Apply program.

**subscription set.** The specification of a group of source tables, target tables, and the control information that governs the replication of changed data. Updates are committed in the same transaction.

**subselect.** That form of a query that does not include an ORDER BY clause, an UPDATE clause, or UNION operators.

**symbolic destination name.** Specifies the name of a remote partner. The name corresponds to an entry in the CPI Communications side information table that contains the necessary information (partner LU name, mode name, partner TP name) for the client to set up an APPC connection to the server.

## Glossary

**synchronization level.** In APPC, the specification indicating whether the corresponding transaction programs exchange confirmation requests and replies.

**synchronous.** Pertaining to two or more processes that depend upon the occurrences of specific events, such as a common timing signal. Contrast with asynchronous.

**sync point.** Synonym for point of consistency.

**system catalog.** See catalog.

**system database directory.** A directory that contains entries for every database that can be accessed using the database manager. It is created when the first database is created or cataloged on the system.

**system managed space (SMS) table space.** A table space whose space is managed by the operating system. This storage model is based on files created under subdirectories, and managed by the file system. Contrast with database managed space (DMS) table space.

**system services control point (SSCP).** The control point in a SNA network that provides network services for dependent nodes.

**Systems Network Architecture (SNA).** The description of the logical structure, formats, protocols, and operational sequences for transmitting information units through the networks and also the operational sequences for controlling the configuration and operation of networks.

## T

**table.** A named data object consisting of a specific number of columns and some unordered rows. See also base table.

**table designator.** A column name qualifier that designates a specific object table.

**table queue.** A mechanism for transferring rows between database nodes. Table queues are distributed row streams with simplified rules for the insertion and removal of rows. Table queues can also be used to deliver rows between different processes in the serial database.

**table space.** An abstraction of a collection of containers into which database objects are stored. A table space provides a level of indirection between a database and the tables stored within the database. A table space:
- Has space on media storage devices assigned to it.
- Has tables created within it. These tables will consume space in the containers that belong to the table space. The data, index, long field, and LOB portions of a table can be stored in the same table space, or can be individually broken out into separate table spaces.

See also system managed space (SMS) table space, database managed space (DMS) table space, long table space, regular table space, temporary table space.

**table space container.** A generic term describing an allocation of space to a table space. Depending on the table space type, the container can be a directory, device, or file.

**target server.** The database location of the target table. Normally this is also the location of the Apply program.

**1222**

**target table.** The table on the target server to which data is copied. It can be a user copy table, a point-in-time table, a base aggregate table, a change aggregate table, a consistent change data table, or a replica table.

**TCP/IP.** Transmission Control Protocol/Internet Protocol.

**temporary table.** A table created during the processing of an SQL statement to hold intermediate results.

**temporary table space.** A table space that can store only temporary tables.

**territory.** A portion of the POSIX** locale that is mapped to the country code for internal processing by the database manager.

**thread.** In some operating systems, the smallest unit of operation to be performed in a process.

**threshold trigger.** An event occurring when the value of a performance variable exceeds or falls below a user-defined threshold value. The action that occurs as a result of a threshold trigger could be:
- Logging information in an alert log file
- Displaying information in an alert log window
- Generating an audio alarm
- Issuing a message window
- Invoking a predefined command or program

**time.** A three-part value that designates a time of day in hours, minutes, and seconds.

**time duration.** A DECIMAL(6,0) value that represents a number of hours, minutes, and seconds.

**timeron.** A unit of measurement used to give a rough relative estimate of the resources, or cost, required by the database server to execute two plans for the same query. The resources calculated in the estimate include weighted CPU and I/O costs.

**timestamp.** A seven-part value that consists of a date and time expressed in years, months, days, hours, minutes, seconds, and microseconds.

**timestamp duration.** A DECIMAL(20,6) value that represents a number of years, months, days, hours, minutes, seconds, and microseconds.

**TM Database.** Transaction Manager Database.

**TP.** Transaction program.

**token.** The basic syntactic unit of a computing language. A token consists of one or more characters, excluding the blank character, and excluding characters within a string constant or delimited identifier.

**topology and routing services (TRS).** An APPN control point component that manages the topology database and computes routes.

**trace table.** A table that contains a high-level record of the execution of the Capture program.

# Glossary

**transaction.** An exchange between a workstation and a program, two workstations, or two programs that accomplish a particular action or result. An example is the entry of a customer's deposit and the update of the customer's balance. Synonym for unit of work.

**transaction compensation.** A process that restores rows that are affected by a committed transaction that is rejected. When a committed transaction is rejected, the rows are restored to the state that they were in before the transaction was committed.

**transaction manager.** Assigns identifiers to transactions, monitors their progress, and takes responsibility for transaction completion and failure recovery.

**Transaction Manager Database (TM Database).** A database that is used to log transactions when a two-phase commit (SYNCPOINT TWOPHASE) is used with DB2 databases. In the event of transaction failure, the TM Database information can be accessed to resynchronize databases involved in the failed transaction.

**transaction program (TP).** An application program that uses APPC to communicate with a partner application program.

**transition table.** A named temporary table that contains the transition values for each row affected by the triggering modification. An old transition table contains the values of affected rows before the modification is applied, and a new transition table contains the values of the affected rows after the modification is applied.

**transition variable.** A variable that is valid only in FOR EACH ROW triggers. It allows access to the transition values for the current row. An old transition variable is the value of the row before the modification is applied, and the new transition variable is the value of the row after the modification is applied.

**Transmission Control Protocol/Internet Protocol (TCP/IP).** A set of communications protocols that provide peer-to-peer connectivity functions for both local and wide area networks.

**trigger.** In DB2, an object in a database that is invoked indirectly by the database manager when a particular SQL statement is run.

**triggered action.** The action that is executed when the trigger event occurs.

**triggered-action condition.** The search condition that controls the execution of the SQL statements within the triggered action.

**trigger event.** In a trigger definition, an update operation (INSERT, UPDATE, or DELETE statement) that causes the trigger to be run.

**truncation.** The process of discarding part of a result from an operation when it exceeds memory or storage capacity.

**tuning parameters table.** A table at the source server that contains timing information used by the Capture program. The information includes:
- How long to keep rows in the change data table.
- How much time can elapse before changes are stored in a database log or journal.
- How often to commit changed data to the unit of work tables.

**two-phase commit.** A two-step process by which recoverable resources and an external subsystem are committed. During the first step, the database manager subsystems are polled to ensure that they are ready to commit. If all subsystems respond positively, the database manager instructs them to commit.

**typed parameter marker.** A parameter marker that is specified along with its target data type. It has the general form:

```
CAST(? AS data-type)
```

# U

**UDF.** User-defined function.

**UDT.** User-defined type.

**unambiguous cursor.** A cursor that has been defined in a manner that allows a relational database to determine whether blocking can be used with the answer set. A cursor defined FOR FETCH ONLY or FOR READ ONLY can be used with blocking, whereas a cursor defined FOR UPDATE cannot.

**unbind session (UNBIND).** A request to deactivate a session between two logical units (LUs).

**uncommitted read (UR).** An isolation level that allows an application to access uncommitted changes of other transactions. The application does not lock other applications out of the row it is reading, unless the other application attempts to drop or alter the table.

**uncoordinated transaction.** A transaction that accesses more than one resource but its commit or rollback is not being coordinated by a Transaction Manager.

**Unicode**\*\*.** An international character encoding scheme that is a subset of the ISO 10646 standard. Each character supported is defined using a unique 2-byte code.

**unique constraint.** The rule that no two values in a primary key or key of a unique index can be the same.

**unique index.** An index that ensures that no identical key values are stored in a table.

**unique key.** A key that is constrained so that no two of its values are equal.

**unit of work.** A recoverable sequence of operations within an application process. At any time, an application process is a single unit of work, but the life of an application process can involve many units of work as a result of commit or rollback operations. Synonym for transaction.

**unit-of-work table.** A replication control table at the source server that contains commit records read from the database log or journal. The records include a unit-of-recovery ID that can be used to join the unit-of-work table and the change data table to produce transaction-consistent change data. For DB2, the unit-of-work table optionally includes the correlation ID, which can be useful for auditing purposes.

**untyped parameter marker.** A parameter marker that is specified without its target data type. It has the form of a single question mark.

**update rule.** A condition enforced by the database manager that must be met before a column can be updated.

## Glossary

**UR.**   Uncommitted read.

**user copy table.**   A target table whose content matches all or part of a source table and contains only user data columns.

**user-defined distinct type.**   See distinct type.

**user-defined function (UDF).**   A function that is defined to the database management system and can be referenced thereafter in SQL queries. It can be one of the following functions:

- An external function, in which the body of the function is written in a programming language whose arguments are scalar values and a scalar result is produced for each invocation.
- A sourced function, implemented by another built-in or user-defined function already known to the DBMS. This function can be either a scalar function or column (aggregating) function, and returns a single value from a set of values (for example, MAX or AVG).

**user-defined performance variable.**   A performance variable created by a user and added to the performance variable profile.

**user-defined target table.**   Any type of target table that is created by a user for an application and can be used in the replication process.

**user-defined type (UDT).**   A data type that is not native to the database manager and was created by a user. See also distinct type.

**user table.**   A table created for and used by an application before it is defined as a replication source. It is used as the source for updates to read-only target tables, consistent change data tables, and replicas.

**UTC.**   Coordinated Universal Time.

## V

**value.**   (1) Smallest unit of data manipulated in SQL. (2) A specific data item at the intersection of a column and a row.

**variable.**   A data element that specifies a value that can be changed.

**variant function.**   A user-defined function whose result is dependent on its input parameter values as well as other factors. Successive invocations with the same parameter values might produce different results. Contrast with not-variant function.

**varying-length string.**   A character, graphic, or binary string whose length is not fixed but can range within set limits.

**view.**   A logical table that consists of data that is generated by a query. Contrast with base table.

**Visual Explain.**   A tool that lets database administrators and application programmers use a graphical interface to display and analyze detailed information on the access plan of a given SQL statement. The tasks provided by this tool can be accessed from the Control Center.

# W

**warm start.**   A restart that allows reuse of previously initialized input and output work queues. Contrast with cold start.

**warm start table.**   A table used by the Capture program to save position in a DBMS log for later reference during warm start.

**well known address.**   An address used to uniquely identify a particular node in the network to establish connections between nodes. The well known address is a combination of the network address and the port used on the logical node.

# X

**XID.**   Exchange station ID.

# Appendix Q. How the DB2 Library Is Structured

The DB2 Universal Database library consists of SmartGuides, online help, books and sample programs in HTML format. This section describes the information that is provided, and how to access it.

To access product information online, you can use the Information Center. You can view task information, DB2 books, troubleshooting information, sample programs, and DB2 information on the Web. See "Accessing Information with the Information Center" on page 1240 for details.

## Completing Tasks with SmartGuides

SmartGuides help you complete some administration tasks by taking you through each task one step at a time. SmartGuides are available through the Control Center and the Client Configuration Assistant. The following table lists the SmartGuides.

**Note:** Create Database, Index, and Configure Multisite Update SmartGuide are available for the partitioned database environment.

| SmartGuide | Helps You to... | How to Access... |
|---|---|---|
| **Add Database** | Catalog a database on a client workstation. | From the Client Configuration Assistant, click **Add**. |
| **Back up Database** | Determine, create, and schedule a backup plan. | From the Control Center, click with the right mouse button on the database you want to back up and select **Backup**->**Database using SmartGuide**. |
| **Configure Multisite Update SmartGuide** | Perform a multi-site update, a distributed transaction, or a two-phase commit. | From the Control Center, click with the right mouse button on the **Database** icon and select **Multisite Update**. |
| **Create Database** | Create a database, and perform some basic configuration tasks. | From the Control Center, click with the right mouse button on the **Databases** icon and select **Create**->**Database using SmartGuide**. |

| SmartGuide | Helps You to... | How to Access... |
|---|---|---|
| **Create Table** | Select basic data types, and create a primary key for the table. | From the Control Center, click with the right mouse button on the **Tables** icon and select **Create**->**Table using SmartGuide**. |
| **Create Table Space** | Create a new table space. | From the Control Center, click with the right mouse button on the **Table spaces** icon and select **Create**->**Table space using SmartGuide**. |
| **Index** | Advise which indexes to create and drop for all your queries. | From the Control Center, click with the right mouse button on the **Index** icon and select **Create**->**Index using SmartGuide**. |
| **Performance Configuration** | Tune the performance of a database by updating configuration parameters to match your business requirements. | From the Control Center, click with the right mouse button on the database you want to tune and select **Configure using SmartGuide**. |
| **Restore Database** | Recover a database after a failure. It helps you understand which backup to use, and which logs to replay. | From the Control Center, click with the right mouse button on the database you want to restore and select **Restore**->**Database using SmartGuide**. |

## Accessing Online Help

Online help is available with all DB2 components. The following table describes the various types of help. You can also access DB2 information through the Information Center. For information see "Accessing Information with the Information Center" on page 1240.

| Type of Help | Contents | How to Access... |
|---|---|---|
| **Command Help** | Explains the syntax of commands in the command line processor. | From the command line processor in interactive mode, enter:<br><br>? *command*<br><br>where *command* is a keyword or the entire command.<br><br>For example, ? `catalog` displays help for all the CATALOG commands, while ? `catalog database` displays help for the CATALOG DATABASE command. |

| Type of Help | Contents | How to Access... |
|---|---|---|
| **Control Center Help** <br><br> **Client Configuration Assistant Help** <br><br> **Event Analyzer Help** <br><br> **Command Center Help** | Explains the tasks you can perform in a window or notebook. The help includes prerequisite information you need to know, and describes how to use the window or notebook controls. | From a window or notebook, click the **Help** push button or press the F1 key. |
| **Message Help** | Describes the cause of a message, and any action you should take. | From the command line processor in interactive mode, enter: <br><br> ? *XXXnnnnn* <br><br> where *XXXnnnnn* is a valid message identifier. <br><br> For example, ? `SQL30081` displays help about the SQL30081 message. <br><br> To view message help one screen at a time, enter: <br><br> ? *XXXnnnnn* `| more` <br><br> To save message help in a file, enter: <br><br> ? *XXXnnnnn* **>** *filename.ext* <br><br> where *filename.ext* is the file where you want to save the message help. |
| **SQL Help** | Explains the syntax of SQL statements. | From the command line processor in interactive mode, enter: <br><br> `help` *statement* <br><br> where *statement* is an SQL statement. <br><br> For example, **help** SELECT displays help about the SELECT statement. <br> **Note:** SQL help is not available on UNIX-based platforms. |

| Type of Help | Contents | How to Access... |
|---|---|---|
| **SQLSTATE Help** | Explains SQL states and class codes. | From the command line processor in interactive mode, enter: |
| | | **?** *sqlstate* or **?** *class-code* |
| | | where *sqlstate* is a valid five-digit SQL state and *class-code* is the first two digits of the SQL state. |
| | | For example, ? 08003 displays help for the 08003 SQL state, while ? 08 displays help for the 08 class code. |

## DB2 Information – Hardcopy and Online

The table in this section lists the DB2 books. They are divided into two groups:

**Cross-platform books**
These books contain the common DB2 information for all platforms.

**Platform-specific books**
These books are for DB2 on a specific platform. For example, there are separate *Quick Beginnings* books for DB2 on OS/2, on Windows NT, and on the UNIX-based platforms.

**Cross-platform sample programs in HTML**
These samples are the HTML version of the sample programs that are installed with the SDK. They are for informational purposes and do not replace the actual programs.

Most books are available in HTML and PostScript format, or you can choose to order a hardcopy from IBM. The exceptions are noted in the table.

On OS/2 and Windows platforms, HTML documentation files can be installed under the doc\html subdirectory. Depending on the language of your system, some files may be in that language, and the remainder are in English.

On UNIX platforms, you can install multiple language versions of the HTML documentation files under the doc/%L/html subdirectories. Any documentation that is not available in a national language is shown in English.

You can obtain DB2 books and access information in a variety of different ways:

| | | | |
|---|---|---|---|
| **View** | See "Viewing Online Information" on page 1239. | | |
| **Search** | See "Searching Online Information" on page 1242. | | |
| **Print** | See "Printing the PostScript Books" on page 1242. | | |
| **Order** | See "Ordering the Printed Books" on page 1244. | | |

| Name | Description | Form Number<br><br>File Name for<br>Online Book | HTML<br>Directory |
|---|---|---|---|
| | **Cross-Platform Books** | | |
| *Administration Guide* | *Administration Guide, Design and Implementation* contains information required to design, implement, and maintain a database. It also describes database access using the Control Center(whether local or in a client/server environment), auditing, database recovery, distributed database support, and high availability.<br><br>*Administration Guide, Performance* contains information that focuses on the database environment, such as application performance evaluation and tuning.<br><br>You can order both volumes of the *Administration Guide* in the English language in North America using the form number SBOF-8922. | Volume 1<br>SC09-2839<br>db2d1x60<br><br>Volume 2<br>SC09-2840<br>db2d2x60 | db2d0 |
| *Administrative API Reference* | Describes the DB2 application programming interfaces (APIs) and data structures you can use to manage your databases. Explains how to call APIs from your applications. | SC09-2841<br><br>db2b0x60 | db2b0 |
| *Application Building Guide* | Provides environment setup information and step-by-step instructions about how to compile, link, and run DB2 applications on Windows, OS/2, and UNIX-based platforms.<br><br>This book combines the *Building Applications* books for the OS/2, Windows, and UNIX-based environments. | SC09-2842<br><br>db2axx60 | db2ax |

| Name | Description | Form Number<br><br>File Name for<br>Online Book | HTML<br>Directory |
|---|---|---|---|
| *APPC, CPI-C and SNA Sense Codes* | Provides general information about APPC, CPI-C, and SNA sense codes that you may encounter when using DB2 Universal Database products.<br>**Note:** Available in HTML format only. | No form number<br><br>db2apx60 | db2ap |
| *Application Development Guide* | Explains how to develop applications that access DB2 databases using embedded SQL or JDBC, how to write stored procedures, user-defined types, user-defined functions, and how to use triggers. It also discusses programming techniques and performance considerations.<br><br>This book was formerly known as the *Embedded SQL Programming Guide*. | SC09-2845<br><br>db2a0x60 | db2a0 |
| *CLI Guide and Reference* | Explains how to develop applications that access DB2 databases using the DB2 Call Level Interface, a callable SQL interface that is compatible with the Microsoft ODBC specification. | SC09-2843<br><br>db2l0x60 | db2l0 |
| *Command Reference* | Explains how to use the command line processor, and describes the DB2 commands you can use to manage your database. | SC09-2844<br><br>db2n0x60 | db2n0 |
| *Data Movement Utilities Guide and Reference* | Explains how to use the Load, Import, Export, Autoloader, and Data Propogation utilities to work with the data in the database. | SC09-2858<br><br>db2dmx60 | db2dm |
| *DB2 Connect Personal Edition Quick Beginnings* | Provides planning, installing, and configuring information for DB2 Connect Personal Edition. | GC09-2830<br><br>db2c1x60 | db2c1 |
| *DB2 Connect User's Guide* | Provides concepts, programming and general usage information about the DB2 Connect products. | SC09-2838<br><br>db2c0x60 | db2c0 |

| Name | Description | Form Number<br><br>File Name for<br>Online Book | HTML<br>Directory |
|---|---|---|---|
| *Connectivity Supplement* | Provides setup and reference information on how to use DB2 for AS/400, DB2 for OS/390, DB2 for MVS, or DB2 for VM as DRDA application requesters with DB2 Universal Database servers, and on how to use DRDA application servers with DB2 Connect application requesters.<br>**Note:** Available in HTML and PostScript formats only. | No form number<br><br>db2h1x60 | db2h1 |
| *Glossary* | Provides a comprehensive list of all DB2 terms and definitions.<br>**Note:** Available in HTML format only. | No form number<br><br>db2t0x50 | db2t0 |
| *Installation and Configuration Supplement* | Guides you through the planning, installation, and set up of platform-specific DB2 clients. This supplement contains information on binding, setting up client and server communications, DB2 GUI tools, DRDA AS, distributed installation, and the configuration of distributed requests and access methods to heterogeneous data sources. | GC09-2857<br><br>db2iyx60 | db2iy |
| *Message Reference* | Lists messages and codes issued by DB2, and describes the actions you should take. | GC09-2846<br><br>db2m0x60 | db2m0 |
| *Replication Guide and Reference* | Provides planning, configuration, administration, and usage information for the IBM Replication tools supplied with DB2. | SC26-9642<br><br>db2e0x60 | db2e0 |
| *SQL Getting Started* | Introduces SQL concepts, and provides examples for many constructs and tasks. | SC09-2856<br><br>db2y0x60 | db2y0 |
| *SQL Reference*, Volume 1 and Volume 2 | Describes SQL syntax, semantics, and the rules of the language. Also includes information about release-to-release incompatibilities, product limits, and catalog views.<br><br>You can order both volumes of the *SQL Reference* in the English language in North America with the form number SBOF-8923. | SBOF-8923<br><br>Volume 1<br>db2s1x60<br><br>Volume 2<br>db2s2x60 | db2s0 |

| Name | Description | Form Number / File Name for Online Book | HTML Directory |
|---|---|---|---|
| *System Monitor Guide and Reference* | Describes how to collect different kinds of information about databases and the database manager. Explains how to use the information to understand database activity, improve performance, and determine the cause of problems. | SC09-2849 / db2f0x60 | db2f0 |
| *Troubleshooting Guide* | Helps you determine the source of errors, recover from problems, and use diagnostic tools in consultation with DB2 Customer Service. | S10J-8169 | db2p0 |
| *What's New* | Describes the new features, functions, and enhancements in DB2 Universal Database, Version 6.0, including information about Java-based tools. | SC09-2851 / db2q0x60 | db2q0 |
| **Platform-Specific Books** | | | |
| *Administering Satellites Guide and Reference* | Provides planning, configuration, administration, and usage information for satellites. | GC09-2821 / db2dsx60 | db2ds |
| *DB2 Personal Edition Quick Beginnings* | Provides planning, installation, migration, and configuration information for DB2 Universal Database Personal Edition on the OS/2, Windows 95, and Windows NT operating systems. | GC09-2831 / db2i1x60 | db2i1 |
| *DB2 for OS/2 Quick Beginnings* | Provides planning, installation, migration, and configuration information for DB2 Universal Database on the OS/2 operating system. Also contains installing and setup information for many supported clients. | GC09-2834 / db2i2x60 | db2i2 |
| *DB2 for UNIX Quick Beginnings* | Provides planning, installation, migration, and configuration information for DB2 Universal Database on UNIX-based platforms. Also contains installing and setup information for many supported clients. | GC09-2836 / db2ixx60 | db2ix |
| *DB2 for Windows NT Quick Beginnings* | Provides planning, installation, migration, and configuration information for DB2 Universal Database on the Windows NT operating system. Also contains installing and setup information for many supported clients. | GC09-2835 / db2i6x60 | db2i6 |

| Name | Description | Form Number<br><br>File Name for<br>Online Book | HTML<br>Directory |
|---|---|---|---|
| *DB2 Enterprise - Extended Edition for UNIX Quick Beginnings* | Provides planning, installation, and configuration information for DB2 Enterprise - Extended Edition for UNIX. Also contains installing and setup information for many supported clients. | GC09-2832<br><br>db2v3x60 | db2v3 |
| *DB2 Enterprise - Extended Edition for Windows NT Quick Beginnings* | Provides planning, installation, and configuration information for DB2 Enterprise - Extended Edition for Windows NT. Also contains installing and setup information for many supported clients. | GC09-2833<br><br>db2v6x60 | db2v6 |
| *DB2 Connect Enterprise Edition for OS/2 and Windows NT Quick Beginnings* | Provides planning, migration, installation, and configuration information for DB2 Connect Enterprise Edition on the OS/2 and Windows NT operating systems. Also contains installation and setup information for many supported clients.<br><br>This book was formerly part of the *DB2 Connect Enterprise Edition Quick Beginnings.* | GC09-2828<br><br>db2c6x60 | db2c6 |
| *DB2 Connect Enterprise Edition for UNIX Quick Beginnings* | Provides planning, migration, installation, configuration, and usage information for DB2 Connect Enterprise Edition in UNIX-based platforms. Also contains installation and setup information for many supported clients.<br><br>This book was formerly part of the *DB2 Connect Enterprise Edition Quick Beginnings.* | GC09-2829<br><br>db2cyx60 | db2cy |
| *DB2 Data Links Manager for AIX Quick Beginnings* | Provides planning, installation, configuration, and task information for DB2 Data Links Manager for AIX. | GC09-2837<br><br>db2z0x60 | db2z0 |
| *DB2 Data Links Manager for Windows NT Quick Beginnings* | Provides planning, installation, configuration, and task information for DB2 Data Links Manager for Windows NT. | GC09-2827<br><br>db2z6x60 | db2z6 |
| *DB2 Query Patroller Administration Guide* | Provides administration information on DB2 Query Patrol. | SC09-2859<br><br>db2dwx60 | db2dw |

| Name | Description | Form Number<br>**File Name for Online Book** | HTML Directory |
|---|---|---|---|
| *DB2 Query Patroller Installation Guide* | Provides installation information on DB2 Query Patrol. | GC09-2860<br><br>db2iwx60 | db2iw |
| *DB2 Query Patroller User's Guide* | Describes how to use the tools and functions of the DB2 Query Patrol. | SC09-2861<br><br>db2wwx60 | db2ww |
| **Cross-Platform Sample Programs in HTML** | | | |
| Sample programs in HTML | Provides the sample programs in HTML format for the programming languages on all platforms supported by DB2 for informational purposes (not all samples are available in all languages). Only available when the SDK is installed.<br><br>See *Application Building Guide* for more information on the actual programs.<br>**Note:** Available in HTML format only. | No form number | db2hs/c<br>db2hs/cli<br>db2hs/clp<br>db2hs/cpp<br>db2hs/cobol<br>db2hs/cobol_mf<br>db2hs/fortran<br>db2hs/java<br>db2hs/rexx |

**Notes:**

1. The character in the sixth position of the file name indicates the language of a book. For example, the file name db2d0e60 indicates that the *Administration Guide* is in English. The following letters are used in the file names to indicate the language of a book:

| Language | Identifier |
|---|---|
| Brazilian Portuguese | b |
| Bulgarian | u |
| Czech | x |
| Danish | d |
| Dutch | q |
| English | e |
| Finnish | y |
| French | f |
| German | g |
| Greek | a |
| Hungarian | h |
| Italian | i |
| Japanese | j |
| Korean | k |
| Norwegian | n |
| Polish | p |

| Portuguese | v |
| Russian | r |
| Simp. Chinese | c |
| Slovenian | l |
| Spanish | z |
| Swedish | s |
| Trad. Chinese | t |
| Turkish | m |

2. For late breaking information that could not be included in the DB2 books:
   - On UNIX-based platforms, see the Release.Notes file. This file is located in the DB2DIR/Readme/%L directory, where %L is the locale name and DB2DIR is:
     - /usr/lpp/db2_06_01 on AIX
     - /opt/IBMdb2/V6.1 on HP-UX, Solaris, SCO UnixWare 7, and Silicon Graphics IRIX
     - /usr/IBMdb2/V6.1 on Linux.
   - On other platforms, see the RELEASE.TXT file. This file is located in the directory where the product is installed.
   - Under Windows `Start` menu

## Viewing Online Information

The manuals included with this product are in Hypertext Markup Language (HTML) softcopy format. Softcopy format enables you to search or browse the information, and provides hypertext links to related information. It also makes it easier to share the library across your site.

You can view the online books or sample programs with any browser that conforms to HTML Version 3.2 specifications.

To view online books or sample programs on all platforms other than SCO UnixWare 7:
- If you are running DB2 administration tools, use the Information Center. See "Accessing Information with the Information Center" on page 1240 for details.
- Select the `Open Page` menu item of your Web browser. The page you open contains descriptions of and links to DB2 information:
  - On UNIX-based platforms, open the following page:

    `file:/INSTHOME/sqllib/doc/%L/html/index.htm`

    where *%L* is the locale name.

– On other platforms, open the following page:

        sqllib\doc\html\index.htm

  The path is located on the drive where DB2 is installed.

  If you have not installed the Information Center, you can open the page by double-clicking on the **DB2 Online Books** icon. Depending on the system you are using, the icon is in the main product folder or the Windows Start menu.

To view online books or sample programs on the SCO UnixWare 7:

- DB2 Universal Database for SCO UnixWare 7 uses the native SCOhelp utility to search the DB2 information. You can access SCOhelp by the following methods:
  – entering the ″scohelp″ command on the command line,
  – selecting the `Help` menu in the Control Panel of the CDE desktop or
  – selecting `Help` in the Root menu of the Panorama desktop

  For more information on SCOhelp, refer to the *Installation and Configuration Supplement*.

## Accessing Information with the Information Center

The Information Center provides quick access to DB2 product information. The Information Center is available on all platforms on which the DB2 administration tools are available.

Depending on your system, you can access the Information Center from the:
- Main product folder
- Toolbar in the Control Center
- Windows Start menu
- Help menu of the Control Center

The Information Center provides the following kinds of information. Click the appropriate tab to look at the information:

| | |
|---|---|
| **Tasks** | Lists tasks you can perform using DB2. |
| **Reference** | Lists DB2 reference information, such as keywords, commands, and APIs. |
| **Books** | Lists DB2 books. |
| **Troubleshooting** | Lists categories of error messages and their recovery actions. |
| **Sample Programs** | Lists sample programs that come with the |

DB2 Software Developer's Kit. If the Software Developer's Kit is not installed, this tab is not displayed.

**Web**    Lists DB2 information on the World Wide Web. To access this information, you must have a connection to the Web from your system.

When you select an item in one of the lists, the Information Center launches a viewer to display the information. The viewer might be the system help viewer, an editor, or a Web browser, depending on the kind of information you select.

The Information Center provides some search capabilities, so you can look for specific topics, and filter capabilities to limit the scope of your searches.

For a full text search, click the `Search` button of the Information Center follow the **Search DB2 Books** link in each HTML file.

The HTML search server is usually started automatically. If a search in the HTML information does not work, you may have to start the search server by double-clicking its icon on the Windows or OS/2 desktop.

Refer to the release notes if you experience any other problems when searching the HTML information.

**Note:** Search function is not available in the Linux and Silicon Graphics environments.

## Setting Up a Document Server

By default, the DB2 information is installed on your local system. This means that each person who needs access to the DB2 information must install the same files. To have the DB2 information stored in a single location, use the following instructions:

1. Copy all files and subdirectories from \sqllib\doc\html on your local system to a Web server. Each book has its own subdirectory containing all the necessary HTML and GIF files that make up the book. Ensure that the directory structure remains the same.
2. Configure the Web server to look for the files in the new location. For information, see the NetQuestion Appendix in *Installation and Configuration Supplement*.

3. If you are using the Java version of the Information Center, you can specify a base URL for all HTML files. You should use the URL for the list of books.

4. Once you are able to view the book files, you should bookmark commonly viewed topics. Among those, you will probably want to bookmark the following pages:
   - List of books
   - Tables of contents of frequently used books
   - Frequently referenced articles, such as the **ALTER TABLE** topic
   - The Search form

For information about setting up a search, see the NetQuestion Appendix in *Installation and Configuration Supplement* book.

## Searching Online Information

To search for information in the HTML books, you can do the following:
- Click on **Search the DB2 Books** at the bottom of any page in the HTML books. Use the search form to find a specific topic. This function is not available in the Linux or Silicon Graphics IRIX environments.
- Click on **Index** at the bottom of any page in an HTML book. Use the index to find a specific topic in the book.
- Display the table of contents or index of the HTML book, and then use the find function of the Web browser to find a specific topic in the book.
- Use the bookmark function of the Web browser to quickly return to a specific topic.
- Use the search function of the Information Center to find specific topics. See "Accessing Information with the Information Center" on page 1240 for details.

## Printing the PostScript Books

If you prefer to have printed copies of the manuals, you can decompress and print PostScript versions. For the file name of each book in the library, see the table in "DB2 Information – Hardcopy and Online" on page 1232. Specify the full path name for the file you intend to print.

On OS/2 and Windows platforms:
1. Copy the compressed PostScript files to a hard drive on your system. The files have a file extension of .exe and are located in the x:\doc\\*language*\books\ps directory, where x: is the letter representing the

CD-ROM drive and *language* is the two-character country code that represents your language (for example, EN for English).

2. Decompress the file that corresponds to the book that you want. Each compressed book is a self-extracting executable file. To decompress the book, simply run it as you would run any other executable program. The result from this step is a printable PostScript file with a file extension of .ps.

3. Ensure that your default printer is a PostScript printer capable of printing Level 1 (or equivalent) files.

4. Enter the following command from a command line:

```
print filename.ps
```

On UNIX-based platforms:

1. Mount the CD-ROM. Refer to your *Quick Beginnings* manual for the procedures to mount the CD-ROM.

2. Change to /cdrom/doc/%L/ps directory on the CD-ROM, where */cdrom* is the mount point of the CD-ROM and *%L* is the name of the desired locale. The manuals will be installed in the previously-mentioned directory with file names ending with .ps.Z.

3. Decompress and print the manual you require using the following command:
   - For AIX:
     ```
     zcat filename | qprt -P PSPrinter_queue
     ```
   - For HP-UX, Solaris, or SCO UnixWare 7:
     ```
     zcat filename | lp -d PSPrinter_queue
     ```
   - For Linux:
     ```
     zcat filename | lpr -P PSPrinter_queue
     ```
   - For Silicon Graphics IRIX:
     ```
     zcat < filename | lp -d PSPrinter_queue
     ```

   where *filename* is the full path name and extension of the compressed PostScript file and *PSprinter_queue* is the name of the PostScript printer queue.

   For example, to print the English version of *DB2 for UNIX Quick Beginnings* on AIX, you can use the following command:
   ```
   zcat /cdrom/doc/en/ps/db2ixe60.ps.Z | qprt -P ps1
   ```

## Ordering the Printed Books

You can order the printed DB2 manuals either as a set or individually. There are two sets of books available. The form number for the entire set of DB2 books is SB0F-8926-00. The form number for the books listed under the heading ″Cross-Platform Books″ is SB0F-8924-00.

**Note:** These form numbers only apply if you are ordering books that are printed in the English language in North America.

You can also order books individually by the form number listed in "DB2 Information – Hardcopy and Online" on page 1232. To order printed versions, contact your IBM authorized dealer or marketing representative, or phone 1-800-879-2755 in the United States or 1-800-IBM-4YOU in Canada.

# Appendix R. Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

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

> IBM Director of Licensing
> IBM Corporation, North Castle Drive
> Armonk, NY 10504-1785
> U.S.A.

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 Canada Limited
> Office of the Lab Director
> 1150 Eglinton Ave. East
> North York, Ontario
> M3C 1H7
> CANADA

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

This publication may contain 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 the names and addresses used by an actual business enterprise is entirely coincidental.

**1245**

## Trademarks

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States and/or other countries:

| | |
|---|---|
| ACF/VTAM | MVS/ESA |
| ADSTAR | MVS/XA |
| AISPO | OS/400 |
| AIX | OS/390 |
| AIXwindows | OS/2 |
| AnyNet | PowerPC |
| APPN | QMF |
| AS/400 | RACF |
| CICS | RISC System/6000 |
| C Set++ | SP |
| C/370 | SQL/DS |
| DATABASE 2 | SQL/400 |
| DataHub | S/370 |
| DataJoiner | System/370 |
| DataPropagator | System/390 |
| DataRefresher | SystemView |
| DB2 | VisualAge |
| DB2 Connect | VM/ESA |
| DB2 Universal Database | VSE/ESA |
| Distributed Relational Database Architecture | VTAM |
| DRDA | WIN-OS/2 |
| Extended Services | |
| FFST | |
| First Failure Support Technology | |
| IBM | |
| IMS | |
| LAN Distance | |

## Trademarks of Other Companies

The following terms are trademarks or registered trademarks of the companies listed:

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

HP-UX is a trademark of Hewlett-Packard.

Java, HotJava, Solaris, Solstice, and Sun are trademarks of Sun Microsystems, Inc.

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

PC Direct is a trademark of Ziff Communications Company in the United States, other countries, or both and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium, and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark in the United States, other countries or both and is licensed exclusively through X/Open Company Limited.

Other company, product, or service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

# Index

## Special Characters

* (asterisk)
    in subselect column names  369
    naming columns, use in
      select  369
? (question mark)  770

## A

ABS or ABSVAL function  188
    detailed format description  228
    values and arguments, rules
      for  228
ACOS function  188
    detailed format description  229
    values and arguments, rules
      for  229
ADD clause in ALTER TABLE  453
ADD column clause, order of
  processing  465
Administering Satellites Guide and
  Reference  1236
Administration Guide  1233
Administrative API Reference  1233
ADVISE_INDEX table  1149
ADVISE_INDEX table
  definition  1161
ADVISE_WORKLOAD table  1152
ADVISE_WORKLOAD table
  definition  1163
alias
    comment descriptions, adding to
      catalog  494
    CREATE ALIAS statement  528
    deleting, using DROP
      statement  749
    description  12, 68
    TABLE_NAME function  338
    TABLE_SCHEMA function  340
ALIAS clause
    COMMENT ON statement  496
    DROP statement  751
alias-name  65
ALL clause
    SELECT statement, use in  369,
      382
ALL in quantified predicate  168
ALL option
    comparison, set operator, effect
      on  409

ALL PRIVILEGES clause
    GRANT statement (table, view or
      nickname)  802
    REVOKE statement, table, view
      or nickname privileges  853
ALTER BUFFERPOOL
  statement  435, 436
ALTER clause
    GRANT statement (table or
      view)  802
    REVOKE statement, removing
      privilege for  853
ALTER NICKNAME statement  437
ALTER NODEGROUP
  statement  441, 444
ALTER SERVER statement  445
ALTER TABLE statement  471
    authorization required,
      summary  449
    examples of usage  468
    syntax diagram  453
ALTER TABLESPACE
  statement  471, 474
ALTER TYPE (Structured)
  statement  475, 476
ALTER USER MAPPING
  statement  477
ALTER VIEW statement  481
    authorization required,
      summary  480
    syntax diagram  480
ambiguous cursor  734
ambiguous reference, error
  conditions for  127
AND truth table  185
ANY in quantified predicate  168
APPC, CPI-C and SNA Sense
  Codes  1233
Application Building Guide  1233
Application Development
  Guide  1234
application process, definition of  22
application program
    concurrency  22
    uses SQLDA  933
application requester, overview  28
application server
    overview  28
    role of in connections  28

arguments of COALESCE
  result data type  105
arithmetic
    AVG function, operation of  206
    columns, adding values
      (SUM)  225
    constants
      definition of  111
      NOT NULL, required
        attribute for  111
    CORRELATION function,
      operation of  208
    COVARIANCE function,
      operation of  213
    date operations, rules for  157
    datetime, SQL rules for  154
    decimal operations, scale and
      precision formulas  152
    decimal value, precision and
      scale  79
    decimal values from numeric
      expressions  256
    distinct type operands  152
    expressions, adding values
      (SUM)  225
    floating point, range and
      precision  79
    floating point operands
      rules and precision
        values  152
      with integers, result of  152
    floating point values from
      numeric expressions  272, 320
    integer
      large integer, range and
        precision  79
      small integer, range and
        precision  79
    integer values, returning from
      expressions  234, 286
    maximum value, finding  216
    minimum value, finding  218
    operators, summary of
      results  150
    parameter marker, syntax and
      operations  825
    REGRESSION Functions function,
      operation of  220

1254

# Contacting IBM

This section lists ways you can get more information from IBM.

If you have a technical problem, please take the time to review and carry out the actions suggested by the *Troubleshooting Guide* before contacting DB2 Customer Support. Depending on the nature of your problem or concern, this guide will suggest information you can gather to help us to serve you better.

For information or to order any of the DB2 Universal Database products contact an IBM representative at a local branch office or contact any authorized IBM software remarketer.

**Telephone**

If you live in the U.S.A., call one of the following numbers:
- 1-800-237-5511 to learn about available service options.
- 1-800-IBM-CALL (1-800-426-2255) or 1-800-3IBM-OS2 (1-800-342-6672) to order products or get general information.
- 1-800-879-2755 to order publications.

For information on how to contact IBM outside of the United States, see Appendix A of the IBM Software Support Handbook. You can access this document by accessing the following page:

`http://www.ibm.com/support/`

then performing a search using the keyword "handbook".

Note that in some countries, IBM-authorized dealers should contact their dealer support structure instead of the IBM Support Center.

**World Wide Web**
> http://www.software.ibm.com/data/
> http://www.software.ibm.com/data/db2/library/

The DB2 World Wide Web pages provide current DB2 information about news, product descriptions, education schedules, and more. The DB2 Product and Service Technical Library provides access to frequently asked questions, fixes, books, and up-to-date DB2 technical information. (Note that this information may be in English only.)

**Anonymous FTP Sites**
> ftp.software.ibm.com

Log on as anonymous. In the directory /ps/products/db2, you can find demos, fixes, information, and tools concerning DB2 and many related products.

**Internet Newsgroups**

    comp.databases.ibm-db2, bit.listserv.db2-l

These newsgroups are available for users to discuss their experiences with DB2 products.

**CompuServe**

    **GO IBMDB2** to access the IBM DB2 Family forums

All DB2 products are supported through these forums.

To find out about the IBM Professional Certification Program for DB2 Universal Database, go to http://www.software.ibm.com/data/db2/db2tech/db2cert.html

**IBM** ®

Part Number: