

Note

Before using this information and the product it supports, read the general information under Appendix O, "Notices," on page 765.

Edition Notice

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

You can order IBM publications online or through your local IBM representative.

- To order publications online, go to the IBM Publications Center at www.ibm.com/shop/publications/order
- To find your local IBM representative, go to the IBM Directory of Worldwide Contacts at www.ibm.com/planetwide

To order DB2 publications from DB2 Marketing and Sales in the United States or Canada, call 1-800-IBM-4YOU (426-4968).

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1993, 2009.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this book	ix
Who should use this book	ix
How this book is structured	ix
How to read the syntax diagrams	x
Conventions used in this manual	xii
Error conditions	xii
Highlighting conventions	xii
Related documentation	xii

Chapter 1. Concepts 1

Databases	1
Structured Query Language (SQL)	1
Queries and table expressions	2
DB2 Call level interface (CLI) and open database connectivity (ODBC)	2
Java database connectivity (JDBC) and embedded SQL for Java (SQLJ) programs	2
Schemas	3
Tables	4
Constraints	4
Indexes	5
Triggers	6
Views	8
Table or view aliases	9
Packages	9
Authorization, privileges, and object ownership	9
System catalog views	15
Application processes, concurrency, and recovery	15
Isolation levels	17
Table spaces	20
Character conversion	22
National language support and SQL statements	24
Connecting to distributed relational databases	25
Event monitors	26
Database partitioning across multiple database partitions	27
Large object behavior in partitioned tables	28
DB2 federated systems	29
Federated systems	29
What is a data source?	30
The federated database	31
The SQL compiler	31
Wrappers and wrapper modules	31
Server definitions and server options	32
User mappings	33
Nicknames and data source objects	34
Nickname column options	34
Data type mappings	35
The federated server	36
Supported data sources	36
The federated database system catalog	40
The query optimizer	41
Collating sequences	41

Chapter 2. Language elements 45

Characters	45
Tokens	46
Identifiers	47
Data types	69
Data type list	70
Promotion of data types	84
Casting between data types	85
Assignments and comparisons	92
Rules for result data types	106
Rules for string conversions	109
String comparisons in a Unicode database	111
Database partition-compatible data types	112
Constants	113
Special registers	117
CURRENT CLIENT_ACCTNG	120
CURRENT CLIENT_APPLNAME	120
CURRENT CLIENT_USERID	120
CURRENT CLIENT_WRKSTNNAME	121
CURRENT DATE	121
CURRENT DBPARTITIONNUM	121
CURRENT DECFLOAT ROUNDING MODE	122
CURRENT DEFAULT TRANSFORM GROUP	123
CURRENT DEGREE	123
CURRENT EXPLAIN MODE	124
CURRENT EXPLAIN SNAPSHOT	125
CURRENT FEDERATED ASYNCHRONY	125
CURRENT IMPLICIT XMLPARSE OPTION	126
CURRENT ISOLATION	126
CURRENT LOCK TIMEOUT	127
CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	127
CURRENT MDC ROLLOUT MODE	128
CURRENT OPTIMIZATION PROFILE	128
CURRENT PACKAGE PATH	128
CURRENT PATH	129
CURRENT QUERY OPTIMIZATION	129
CURRENT REFRESH AGE	130
CURRENT SCHEMA	130
CURRENT SERVER	130
CURRENT TIME	131
CURRENT TIMESTAMP	131
CURRENT TIMEZONE	132
CURRENT USER	132
SESSION_USER	132
SYSTEM_USER	133
USER	133
Global variables	133
Functions	134
Methods	142
Expressions	149
Datetime operations and durations	158
CASE expression	163
CAST specification	165
XMLCAST specification	169
ARRAY element specification	170
Dereference operation	170

OLAP specifications	172
Method invocation	180
Subtype treatment	181
Sequence reference	182
ROW CHANGE expression	185
Predicates	186
Predicate processing for queries	187
Search conditions	190
Basic predicate	192
Quantified predicate	193
BETWEEN predicate	195
EXISTS predicate	195
IN predicate	196
LIKE predicate	197
NULL predicate	202
TYPE predicate	203
VALIDATED predicate	204
XMLEXISTS predicate	206

Chapter 3. Functions 209

Functions overview	209
Supported functions and administrative SQL routines and views	210
Aggregate functions	237
ARRAY_AGG	238
AVG	239
CORRELATION	240
COUNT	241
COUNT_BIG	242
COVARIANCE	243
GROUPING	244
MAX	245
MIN	246
Regression functions	247
STDDEV	250
SUM	251
VARIANCE	252
XMLAGG	252
XMLGROUP	254
Scalar functions	257
ABS or ABSVAL	257
ACOS	257
ASCII	258
ASIN	258
ATAN	259
ATAN2	259
ATANH	259
BIGINT	260
BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT	261
BLOB	263
CARDINALITY	263
CEILING	264
CHAR	264
CHARACTER_LENGTH	268
CHR	270
CLOB	270
COALESCE	271
COLLATION_KEY_BIT	271
COMPARE_DECFLOAT	273
CONCAT	274

COS	274
COSH	274
COT	275
DATAPARTITIONNUM	275
DATE	276
DAY	277
DAYNAME	278
DAYOFWEEK	278
DAYOFWEEK_ISO	278
DAYOFYEAR	279
DAYS	279
DBCLOB	280
DBPARTITIONNUM	280
DECFLOAT	282
DECIMAL	283
DECODE	286
DECRYPT_BIN and DECRYPT_CHAR	287
DEGREES	288
DEREF	289
DIFFERENCE	289
DIGITS	290
DOUBLE	291
ENCRYPT	292
EVENT_MON_STATE	293
EXP	294
FLOAT	294
FLOOR	295
GENERATE_UNIQUE	295
GETHINT	297
GRAPHIC	297
GREATEST	299
HASHEDVALUE	299
HEX	300
HOUR	302
IDENTITY_VAL_LOCAL	302
INSERT	306
INTEGER	309
JULIAN_DAY	310
LCASE	311
LCASE (Locale sensitive)	311
LEAST	311
LEFT	312
LENGTH	314
LN	316
LOCATE scalar function	317
LOG10	320
LONG_VARCHAR	320
LONG_VARGRAPHIC	321
LOWER	321
LOWER (Locale sensitive)	321
LTRIM	323
MAX	324
MAX_CARDINALITY	324
MICROSECOND	325
MIDNIGHT_SECONDS	325
MIN	326
MINUTE	327
MOD	328
MONTH	328
MONTHNAME	329
MULTIPLY_ALT	329

NORMALIZE_DECFLOAT	330
NULLIF	331
NVL	331
OCTET_LENGTH	332
OVERLAY	332
PARAMETER	336
POSITION scalar function	336
POSSTR scalar function	339
POWER	341
QUANTIZE	342
QUARTER	343
RADIANS	343
RAISE_ERROR	344
RAND	345
REAL	345
REC2XML	346
REPEAT	350
REPLACE	350
RID_BIT and RID	352
RIGHT	353
ROUND	356
RTRIM	358
SECLABEL	358
SECLABEL_BY_NAME	359
SECLABEL_TO_CHAR	360
SECOND	361
SIGN	362
SIN	363
SINH	363
SMALLINT	363
SOUNDEX	364
SPACE	365
SQRT	365
STRIP	366
SUBSTR	366
SUBSTRING	369
TABLE_NAME	371
TABLE_SCHEMA	372
TAN	373
TANH	374
TIME	374
TIMESTAMP	375
TIMESTAMP_FORMAT	376
TIMESTAMP_ISO	379
TIMESTAMPDIFF	379
TO_CHAR	380
TO_DATE	381
TOTALORDER	381
TRANSLATE scalar function	382
TRIM	384
TRUNCATE	385
TYPE_ID	386
TYPE_NAME	387
TYPE_SCHEMA	387
UCASE	388
UCASE (Locale sensitive)	388
UPPER	388
UPPER (Locale sensitive)	388
VALUE	390
VARCHAR	390
VARCHAR_BIT_FORMAT	392

VARCHAR_FORMAT	393
VARCHAR_FORMAT_BIT	395
VARGRAPHIC	396
WEEK	398
WEEK_ISO	398
XMLATTRIBUTES	399
XMLCOMMENT	400
XMLCONCAT	401
XMLDOCUMENT	402
XMLELEMENT	403
XMLFOREST	408
XMLNAMESPACES	411
XMLPARSE	413
XMLPI	415
XMLQUERY	416
XMLROW	418
XMLSERIALIZE	420
XMLTEXT	422
XMLVALIDATE	423
XMLXSROBJECTID	427
XSLTRANSFORM	428
YEAR	431
Table functions	432
XMLTABLE	432
User-defined functions	436

Chapter 4. Procedures 439

Procedures overview	439
XSR_ADDSCHEMADOC procedure	439
XSR_COMPLETE procedure	440
XSR_DTD procedure	441
XSR_EXTENTITY procedure	442
XSR_REGISTER procedure	444
XSR_UPDATE procedure	445

Chapter 5. SQL queries 447

Subselect	447
Fullselect	485
Select-statement	489

Appendix A. SQL and XML limits . . . 501

Appendix B. SQLCA (SQL communications area) 511

Appendix C. SQLDA (SQL descriptor area) 517

Appendix D. System catalog views 527

Road map to the catalog views	528
SYSCAT.ATTRIBUTES	532
SYSCAT.AUDITPOLICIES	533
SYSCAT.AUDITUSE	535
SYSCAT.BUFFERPOOLDBPARTITIONS	535
SYSCAT.BUFFERPOOLS	536
SYSCAT.CASTFUNCTIONS	536
SYSCAT.CHECKS	537
SYSCAT.COLAUTH	538
SYSCAT.COLCHECKS	539

SYSCAT.COLDIST	539	SYSCAT.SECURITYLABELACCESS	601
SYSCAT.COLGROUPCOLS	540	SYSCAT.SECURITYLABELCOMPONENTELEMENTS	601
SYSCAT.COLGROUPDIST	540	SECURITYLABELCOMPONENTELEMENTS	601
SYSCAT.COLGROUPDISTCOUNTS	541	SYSCAT.SECURITYLABELCOMPONENTS	602
SYSCAT.COLGROUPS	541	SYSCAT.SECURITYLABELS	602
SYSCAT.COLIDENTATTRIBUTES	542	SYSCAT.SECURITYPOLICIES	602
SYSCAT.COLOPTIONS	542	SYSCAT.SECURITYPOLICYCOMPONENTRULES	603
SYSCAT.COLUMNS	543	SYSCAT.SECURITYPOLICYEXEMPTIONS	604
SYSCAT.COLUSE	547	SYSCAT.SEQUENCEAUTH	605
SYSCAT.CONSTDEP	548	SYSCAT.SEQUENCES	605
SYSCAT.CONTEXTATTRIBUTES	548	SYSCAT.SERVEROPTIONS	607
SYSCAT.CONTEXTS	548	SYSCAT.SERVERS	607
SYSCAT.DATAPARTITIONEXPRESSION	549	SYSCAT.SERVICECLASSES	607
SYSCAT.DATAPARTITIONS	549	SYSCAT.STATEMENTS	608
SYSCAT.DATATYPES	550	SYSCAT.SURROGATEAUTHIDS	609
SYSCAT.DBAUTH	552	SYSCAT.TABAUTH	610
SYSCAT.DBPARTITIONGROUPDEF	554	SYSCAT.TABCONST	611
SYSCAT.DBPARTITIONGROUPS	554	SYSCAT.TABDEP	612
SYSCAT.EVENTMONITORS	555	SYSCAT.TABDETACHEDDEP	613
SYSCAT.EVENTS	556	SYSCAT.TABLES	614
SYSCAT.EVENTTABLES	557	SYSCAT.TABLESPACES	619
SYSCAT.FULLHIERARCHIES	558	SYSCAT.TABOPTIONS	621
SYSCAT.FUNCMAPOPTIONS	559	SYSCAT.TBSPACEAUTH	621
SYSCAT.FUNCMAPPARMOPTIONS	559	SYSCAT.THRESHOLDS	621
SYSCAT.FUNCMAPPINGS	559	SYSCAT.TRANSFORMS	623
SYSCAT.HIERARCHIES	560	SYSCAT.TRIGDEP	624
SYSCAT.HISTOGRAMTEMPLATEBINS	561	SYSCAT.TRIGGERS	624
SYSCAT.HISTOGRAMTEMPLATES	561	SYSCAT.TYPEMAPPINGS	626
SYSCAT.HISTOGRAMTEMPLATEUSE	561	SYSCAT.USEROPTIONS	628
SYSCAT.INDEXAUTH	562	SYSCAT.VARIABLEAUTH	629
SYSCAT.INDEXCOLUSE	562	SYSCAT.VARIABLEDEP	629
SYSCAT.INDEXDEP	563	SYSCAT.VARIABLES	630
SYSCAT.INDEXES	564	SYSCAT.VIEWS	631
SYSCAT.INDEXEXPLOITRULES	569	SYSCAT.WORKACTIONS	632
SYSCAT.INDEXEXTENSIONDEP	570	SYSCAT.WORKACTIONSETS	634
SYSCAT.INDEXEXTENSIONMETHODS	571	SYSCAT.WORKCLASSES	634
SYSCAT.INDEXEXTENSIONPARMS	571	SYSCAT.WORKCLASSETS	635
SYSCAT.INDEXEXTENSIONS	572	SYSCAT.WORKLOADAUTH	636
SYSCAT.INDEXOPTIONS	572	SYSCAT.WORKLOADCONNATTR	636
SYSCAT.INDEXXMLPATTERNS	573	SYSCAT.WORKLOADS	637
SYSCAT.KEYCOLUSE	573	SYSCAT.WRAPOPTIONS	638
SYSCAT.NAMEMAPPINGS	574	SYSCAT.WRAPPEES	638
SYSCAT.NICKNAMES	574	SYSCAT.XDBMAPGRAPHS	638
SYSCAT.PACKAGEAUTH	577	SYSCAT.XDBMAPSHREDTREES	638
SYSCAT.PACKAGEDEP	578	SYSCAT.XSROBJECTAUTH	639
SYSCAT.PACKAGES	579	SYSCAT.XSROBJECTCOMPONENTS	639
SYSCAT.PARTITIONMAPS	584	SYSCAT.XSROBJECTDEP	640
SYSCAT.PASSTHROUGHAUTH	584	SYSCAT.XSROBJECTHIERARCHIES	641
SYSCAT.PREDICATESPECS	584	SYSCAT.XSROBJECTS	641
SYSCAT.REFERENCES	585	SYSIBM.SYSDUMMY1	642
SYSCAT.ROLEAUTH	586	SYSSTAT.COLDIST	642
SYSCAT.ROLES	586	SYSSTAT.COLGROUPDIST	643
SYSCAT.ROUTINEAUTH	586	SYSSTAT.COLGROUPDISTCOUNTS	643
SYSCAT.ROUTINEDEP	587	SYSSTAT.COLGROUPS	644
SYSCAT.ROUTINEOPTIONS	588	SYSSTAT.COLUMNS	644
SYSCAT.ROUTINEPARMOPTIONS	589	SYSSTAT.INDEXES	645
SYSCAT.ROUTINEPARMS	589	SYSSTAT.ROUTINES	648
SYSCAT.ROUTINES	591	SYSSTAT.TABLES	649
SYSCAT.ROUTINESFEDERATED	598		
SYSCAT.SCHEMAAUTH	600		
SYSCAT.SCHEMATA	600		

Appendix E. Federated systems 651

Valid server types in SQL statements 651

Function mapping options for federated systems	652	EXPLAIN_ARGUMENT table	711
Default forward data type mappings	652	EXPLAIN_DIAGNOSTIC table	717
DB2 Database for Linux, UNIX, and Windows		EXPLAIN_DIAGNOSTIC_DATA table	718
data sources	652	EXPLAIN_INSTANCE table	719
DB2 for System i data sources	653	EXPLAIN_OBJECT table	722
DB2 for VM and VSE data sources	654	EXPLAIN_OPERATOR table	724
DB2 for z/OS data sources	655	EXPLAIN_PREDICATE table	726
Informix data sources	655	EXPLAIN_STATEMENT table	729
Microsoft SQL Server data sources	656	EXPLAIN_STREAM table	731
ODBC data sources	658		
Oracle NET8 data sources	659	Appendix J. Explain register values	735
Sybase data sources	659		
Teradata data sources	661	Appendix K. Exception tables	741
Default reverse data type mappings	661		
DB2 Database for Linux, UNIX, and Windows		Appendix L. SQL statements allowed	
data sources	662	in routines	745
DB2 for System i data sources	663		
DB2 for VM and VSE data sources	663	Appendix M. CALL invoked from a	
DB2 for z/OS data sources	664	compiled statement.	749
Informix data sources	665		
Microsoft SQL Server data sources	665	Appendix N. Overview of the DB2	
Oracle NET8 data sources	666	technical information	755
Sybase data sources	667	DB2 technical library in hardcopy or PDF format	755
Teradata data sources	667	Ordering printed DB2 books	758
		Displaying SQL state help from the command line	
Appendix F. The SAMPLE database	669	processor	759
		Accessing different versions of the DB2	
Appendix G. Reserved schema names		Information Center	759
and reserved words	697	Displaying topics in your preferred language in the	
		DB2 Information Center	759
Appendix H. Examples of interaction		Updating the DB2 Information Center installed on	
between triggers and referential		your computer or intranet server	760
constraints	701	DB2 tutorials	762
		DB2 troubleshooting information	762
Appendix I. Explain tables	703	Terms and Conditions	762
ADVISE_INDEX table	703		
ADVISE_INSTANCE table	707	Appendix O. Notices	765
ADVISE_MQT table	707		
ADVISE_PARTITION table	708	Index	769
ADVISE_TABLE table	710		
ADVISE_WORKLOAD table	710		

About this book

The SQL Reference in its two volumes defines the SQL language used by DB2[®] Database for Linux[®], UNIX[®], and Windows[®]. It includes:

- Information about relational database concepts, language elements, functions, and the forms of queries (Volume 1)
- Information about the syntax and semantics of SQL statements (Volume 2)

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 those who access databases through the command line processor (CLP).

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 this book is structured

The first volume of the SQL Reference contains information about relational database concepts, language elements, functions, and the forms of queries. The specific chapters and appendixes in that volume are briefly described here.

- “Concepts” discusses the basic concepts of relational databases and SQL.
- “Language elements” describes the basic syntax of SQL and the language elements that are common to many SQL statements.
- “Functions” contains syntax diagrams, semantic descriptions, rules, and usage examples of SQL aggregate and scalar functions.
- “Procedures” contains syntax diagrams, semantic descriptions, rules, and usage examples of procedures.
- “SQL queries” describes the various forms of a query.
- “SQL and XML limits” lists the SQL limitations.
- “SQLCA (SQL communications area)” describes the SQLCA structure.
- “SQLDA (SQL descriptor area)” describes the SQLDA structure.
- “System catalog views” describes the system catalog views.
- “Federated systems” describes options and type mappings for federated systems.
- “The SAMPLE database” introduces the SAMPLE database, which contains the tables that are used in many examples.
- “Reserved schema names and reserved words” contains the reserved schema names and the reserved words for the IBM[®] SQL and ISO/ANSI SQL2003 standards.
- “Examples of interaction between triggers and referential constraints” discusses the interaction of triggers and referential constraints.
- “Explain tables” describes the explain tables.
- “Explain register values” 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.

How this book is structured

- “Exception tables” contains information about user-created tables that are used with the SET INTEGRITY statement.
- “SQL statements allowed in routines” lists the SQL statements that are allowed to execute in routines with different SQL data access contexts.
- “CALL invoked from a compiled statement” describes the CALL statement that can be invoked from a compiled statement.

How to read the syntax diagrams

The following information will help you to understand the structure of syntax diagrams:

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

The \blacktriangleright — symbol indicates the beginning of a syntax diagram.

The — \blacktriangleright symbol indicates that the syntax is continued on the next line.

The \blacktriangleright — symbol indicates that the syntax is continued from the previous line.

The — \blacktriangleleft symbol indicates the end of a syntax diagram.

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

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



Optional items appear below the main path.

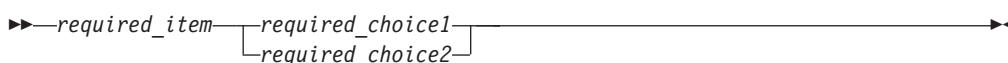


If an optional item appears above the main path, that item has no effect on execution, and is used only for readability.



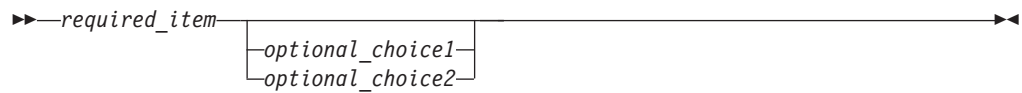
If you can choose from two or more items, they appear in a stack.

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

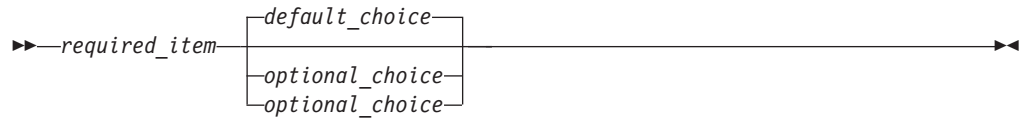


If choosing one of the items is optional, the entire stack appears below the main path.

How to read the syntax diagrams



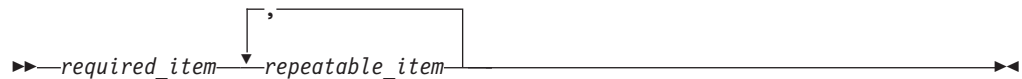
If one of the items is the default, it will appear above the main path, and the remaining choices will be shown below.



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



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



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

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

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

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



parameter-block:



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

How to read the syntax diagrams



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

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

Conventions used in this manual

Error conditions

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

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

Highlighting conventions

The following conventions are used in this book.

Bold	Indicates commands, keywords, and other items whose names are predefined by the system.
<i>Italics</i>	Indicates one of the following: <ul style="list-style-type: none">Names or values (variables) that must be supplied by the userGeneral emphasisThe introduction of a new termA reference to another source of information
Monospace	Indicates one of the following: <ul style="list-style-type: none">Files and directoriesInformation that you are instructed to type at a command prompt or in a windowExamples of specific data valuesExamples of text similar to what might be displayed by the systemExamples of system messages

Related documentation

The following publications might prove useful when you are preparing applications:

- *Getting Started with Database Application Development*
 - Provides an introduction to DB2 application development, including platform prerequisites; supported development software; and guidance on the benefits and limitations of the supported programming APIs.
- *DB2 for i5/OS SQL Reference*
 - This book defines SQL as supported by DB2 Query Manager and SQL Development Kit on System i[™]. It contains reference information for the tasks of system administration, database administration, application programming, and operation. This manual includes syntax, usage notes, keywords, and examples for each of the SQL statements used on i5/OS[®] systems running DB2.
- *DB2 for z/OS SQL Reference*

- This book defines SQL used in DB2 for z/OS[®]. It provides query forms, SQL statements, SQL procedure statements, DB2 limits, SQLCA, SQLDA, catalog tables, and SQL reserved words for z/OS systems running DB2.
- *DB2 Spatial Extender User's Guide and Reference*
 - This book discusses how to write applications to create and use a geographic information system (GIS). Creating and using a GIS involves supplying a database with resources and then querying the data to obtain information such as locations, distances, and distributions within areas.
- *IBM SQL Reference*
 - This book contains all the common elements of SQL that span IBM's database products. It provides limits and rules that assist in preparing portable programs using IBM databases. This manual provides a list of SQL extensions and incompatibilities among the following standards and products: SQL92E, XPG4-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 1992 ISO standard definition of SQL.
- *ISO/IEC 9075-2:2003, Information technology -- Database Languages -- SQL -- Part 2: Foundation (SQL/Foundation)*
 - Contains a large portion of the 2003 ISO standard definition of SQL.
- *ISO/IEC 9075-4:2003, Information technology -- Database Languages -- SQL -- Part 4: Persistent Stored Modules (SQL/PSM)*
 - Contains the 2003 ISO standard definition for SQL procedure control statements.

Related documentation

Chapter 1. Concepts

Databases

A DB2 database is a *relational database*. The *database* stores all data in tables that are related to one another. Relationships are established between tables such that data is shared and duplication is minimized.

A *relational database* is a database that is treated 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. Objects can be either defined by the system (system-defined objects) or defined by the user (user-defined objects).

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.

A *partitioned relational database* is a relational database whose data is managed across multiple database partitions. This separation of data across database partitions is transparent to users of most SQL statements. However, some data definition language (DDL) statements take database partition information into consideration (for example, CREATE DATABASE PARTITION GROUP). DDL is the subset of SQL statements used to describe data relationships in a database.

A *federated database* is a relational database whose data is stored in multiple data sources (such as separate relational databases). The data appears as if it were all in a single large database and can be accessed through traditional SQL queries. Changes to the data can be explicitly directed to the appropriate data source.

Structured Query Language (SQL)

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

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

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

Queries and table expressions

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

A *table expression* creates a temporary result table from a simple query. Clauses further refine the result table. For example, you can use a table expression as a query to select all of the managers from several departments, specify that they must have over 15 years of working experience, and be located at the New York branch office.

A *common table expression* is like a temporary view within a complex query. It can be referenced in other places within the query, and can be used in place of a view. Each use of a specific common table expression within a complex query shares the same temporary view.

Recursive use of a common table expression within a query can be used to support applications such as airline reservation systems, bill of materials (BOM) generators, and network planning.

DB2 Call level interface (CLI) and open database connectivity (ODBC)

The DB2 call level interface is an application programming interface that provides functions for processing dynamic SQL statements to application programs. CLI programs can also be compiled using an open database connectivity Software Developer's Kit (available from Microsoft® or other vendors), which enables access to ODBC data sources. Unlike embedded SQL, this interface requires no precompilation. Applications can be run against a variety of databases without having to be compiled against each of these databases. Applications use procedure calls at run time to connect to databases, issue SQL statements, and retrieve data and status information.

The DB2 CLI interface provides many features not available in embedded SQL. For example:

- CLI provides function calls that support a way of querying database catalogs that is consistent across the DB2 family. This reduces the need to write catalog queries that must be tailored to specific database servers.
- CLI provides the ability to scroll through a cursor:
 - Forward by one or more rows
 - Backward by one or more rows
 - Forward from the first row by one or more rows
 - Backward from the last row by one or more rows
 - From a previously stored location in the cursor.
- Stored procedures called from application programs that were written using CLI can return result sets to those programs.

Java database connectivity (JDBC) and embedded SQL for Java (SQLJ) programs

DB2 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:

Java database connectivity (JDBC) and embedded SQL for Java (SQLJ) programs

- JDBC calls are translated into DB2 CLI calls through Java native methods. JDBC requests flow from the DB2 client through DB2 CLI to the DB2 server. JDBC cannot use static SQL.
- 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 must be translated by the SQLJ translator before the resulting Java source code can be compiled.

Schemas

A *schema* is a collection of named objects; it provides a way to group those objects logically. A schema is also a name qualifier; it provides a way to use the same natural name for several objects, and to prevent ambiguous references to those objects.

For example, the schema names 'INTERNAL' and 'EXTERNAL' make it easy to distinguish two different SALES tables (INTERNAL.SALES, EXTERNAL.SALES).

Schemas also enable multiple applications to store data in a single database without encountering namespace collisions.

A schema is distinct from, and should not be confused with, an *XML schema*, which is a standard that describes the structure and validates the content of XML documents.

A schema can contain tables, views, nicknames, triggers, functions, packages, and other objects. A schema is itself a database object. It is explicitly created using the CREATE SCHEMA statement, with the current user or a specified authorization ID recorded as the schema owner. It can also be implicitly created when another object is created, if the user has IMPLICIT_SCHEMA authority.

A *schema name* is used as the high order part of a two-part object name. If the object is specifically qualified with a schema name when created, the object is assigned to that schema. If no schema name is specified when the object is created, the default schema name is used (specified in the CURRENT SCHEMA special register).

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 (provided that user A has the CREATETAB database authority):

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

Some schema names are reserved. For example, built-in functions belong to the SYSIBM schema, and the pre-installed user-defined functions belong to the SYSFUN schema.

When a database is created, if it is not created with the RESTRICTIVE option, all users have IMPLICIT_SCHEMA authority. With this authority, users implicitly create a schema whenever they create an object with a schema name that does not already exist. When schemas are implicitly created, CREATEIN privileges are granted which allows any user to create other objects in this schema. The ability to create objects such as aliases, distinct types, functions, and triggers is extended to

Schemas

implicitly-created schemas. The default privileges on an implicitly-created schema provide backward compatibility with previous versions.

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

Schemas also have privileges, allowing the schema owner to control which users have the privilege to create, alter, copy, and drop objects in the schema. This provides a way to control the manipulation of a subset of objects in the database. A schema owner is initially given all of these privileges on the schema, with the ability to grant the privileges 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 `SYSADM` or `DBADM` authority can change the privileges that are held by users on any schema. Therefore, access to create, alter, copy, and drop objects in any schema (even one that was implicitly created) can be controlled.

Tables

Tables are logical structures maintained by the database manager. Tables are made up of columns and rows.

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 type or one of its subtypes. A *row* is a sequence of values arranged so that the *n*th value is a value of the *n*th column of the table.

An application program can determine the order in which the rows are populated into the table, but the actual order of rows is determined by the database manager, and typically cannot be controlled. Multidimensional clustering (MDC) provides some sense of clustering, but not actual ordering between the rows.

Constraints

Within any business, data must often adhere to certain restrictions or rules. For example, an employee number must be unique. The database manager provides *constraints* as a way to enforce such rules.

The following types of constraints are available:

- NOT NULL constraints
- Unique (or unique key) constraints
- Primary key constraints
- Foreign key (or referential integrity) constraints
- (Table) Check constraints
- Informational constraints

Constraints are only associated with tables and are either defined as part of the table creation process (using the `CREATE TABLE` statement) or are added to a table's definition after the table has been created (using the `ALTER TABLE` statement). You can use the `ALTER TABLE` statement to modify constraints. In

most cases, existing constraints can be dropped at any time; this action does not affect the table's structure or the data stored in it.

Note: Unique and primary constraints are only associated with table objects, they are often enforced through the use of one or more unique or primary key indexes.

Indexes

An *index* is a set of one or more keys, each key pointing to a row in a table. The SQL *optimizer* automatically chooses the most efficient way to access data in tables. The optimizer takes indexes into consideration when determining the fastest access path to data.

Note: Not all indexes point to rows in a table. MDC block indexes point to extents (or blocks) of the data. XML indexes for XML data use particular XML pattern expressions to index paths and values in XML documents stored within a single column. The data type of that column must be XML. Both MDC block indexes and XML indexes are system generated indexes.

Indexes are used by the database manager to:

- Improve performance. In most cases, access to data is faster with an index. Although an index cannot be created for a view, an index created for the table on which a view is based can sometimes improve the performance of operations on that view.
- Ensure uniqueness. A table with a unique index cannot have rows with identical keys.

As data is added to a table, unless other actions have been carried out on the table or the data being added, the data is simply appended to the bottom of the table. There is no order to the data. When searching for a particular row of data, each row of the table from first to last must be checked. Indexes are used as a means to access the data within the table in an order that might otherwise not be available.

A column value in a row of data can be used to identify the entire row. One or more columns might be needed to identify the row. Such columns are known as a *key*. A column can be used in more than one key.

An index is ordered by the values within a key. Keys can be unique or non-unique. Each table should have at least one unique key; but can also have other, non-unique keys. Each index has exactly one key. For example, you might use the employee ID number (unique) as the key for one index and the department number (non-unique) as the key for a different index.

Example

Table A in Figure 1 on page 6 has an index based on the employee numbers in the table. This key value provides a pointer to the rows in the table. For example, employee number 19 points to employee KMP. An index allows efficient access to rows in a table by creating a path to the data through pointers.

Unique indexes can be created to ensure uniqueness of the index key. An *index key* is a column or an ordered collection of columns on which an index is defined. Using a unique index will ensure that the value of each index key in the indexed column or columns is unique.

Indexes

Figure 1 shows the relationship between an index and a table.

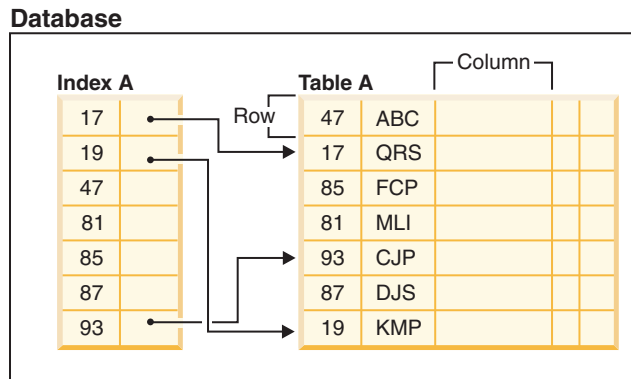


Figure 1. Relationship between an index and a table

Figure 2 illustrates the relationships among some database objects. It also shows that tables, indexes, and long data are stored in table spaces.

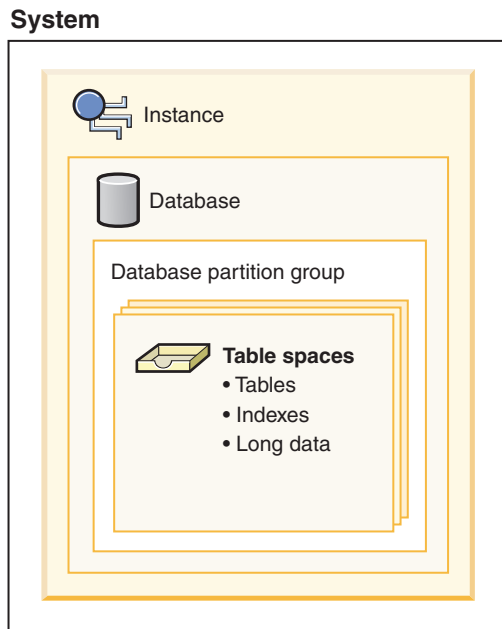


Figure 2. Relationships among selected database objects

Triggers

A *trigger* defines a set of actions that are performed in response to an insert, update, or delete operation on a specified table. When such an SQL operation is executed, the trigger is said to have been *activated*. Triggers are optional and are defined using the CREATE TRIGGER statement.

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 for defining and enforcing *transitional* business rules, which are rules that involve different states of the data (for example, a salary that cannot be increased by more than 10 percent).

Using triggers places the logic that enforces business rules inside the database. This means that applications are not responsible for enforcing these rules. Centralized logic that is enforced on all of the tables means easier maintenance, because changes to application programs are not required when the logic changes.

The following are specified when creating a trigger:

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

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

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

The triggered action can refer to the values in the set of affected rows using *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 (before the update) or the new value (after the update). The new value can also be changed using the SET Variable statement in before, insert, or update triggers.

Another means of referring to the values in the set of affected rows is to use *transition tables*. Transition tables also use the names of the columns in the subject table, but specify a name to allow the complete set of affected rows to be treated as a table. Transition tables can only be used in AFTER triggers (that is, not with BEFORE and INSTEAD OF triggers), and separate transition tables can be defined for old and new values.

Multiple triggers can be specified for a combination of table, event (INSERT, UPDATE, DELETE, INSTEAD OF), or activation time (BEFORE, AFTER). When more than one trigger exists for a particular table, event, and 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 is the last trigger to be activated.

The activation of a trigger might cause *trigger cascading*, which is the result of the activation of one trigger that executes statements that cause the activation of other triggers or even the same trigger again. The triggered actions might also cause updates resulting from the application of referential integrity rules for deletions that can, in turn, result in the activation of additional triggers. With trigger

Triggers

cascading, a chain of triggers and referential integrity delete rules can be activated, causing significant change to the database as a result of a single INSERT, UPDATE, or DELETE statement.

When multiple triggers have insert, update, or delete actions against the same object, conflict resolution mechanism, like temporary tables, are used to resolve access conflicts, and this can have a noticeable impact on performance, particularly in partitioned database environments.

Views

A *view* is an efficient way of representing data without the need to maintain it. A view is not an actual table and requires no permanent storage. A “virtual table” is created and used.

A *view* provides a different way of looking at the data in one or more tables; it is a named specification of a result table. The specification is a SELECT statement that is run whenever the view is referenced in an SQL statement. A view has columns and rows just like a table. All views can be used just like tables for data retrieval. Whether a view can be used in an insert, update, or delete operation depends on its definition.

A view can include all or some of the columns or rows contained in the tables on which it is based. For example, you can join a department table and an employee table in a view, so that you can list all employees in a particular department.

Figure 3 shows the relationship between tables and views.

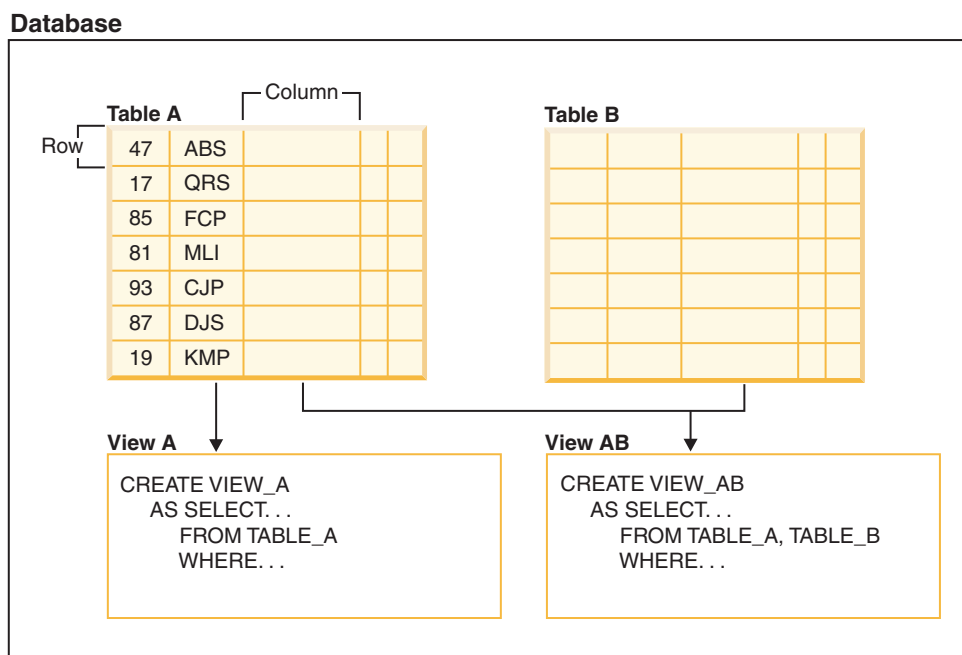


Figure 3. Relationship between tables and views

You can use views to control access to sensitive data, because views allow multiple users to see different presentations of the same data. For example, several users might be accessing a table of data about employees. A manager sees data about his or her employees but not employees in another department. A recruitment officer

sees the hire dates of all employees, but not their salaries; a financial officer sees the salaries, but not the hire dates. Each of these users works with a view derived from the table. Each view appears to be a table and has its own name.

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

A view can derive the data type of each column from the result table, or base the types 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* inherits columns from its *superview*. The term *subview* applies to a typed view and to 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 can become inoperative (for example, if the table is dropped); if this occurs, the view is no longer available for SQL operations.

Table or view aliases

An *alias* is an alternative name for a table or a view. It can be used to reference a table or a view if an existing table or view *can* be referenced.

An alias cannot be used in all contexts; for example, it cannot be used in the check condition of a check constraint. An alias cannot reference a declared temporary table.

Like tables or views, an alias can be created, dropped, and have comments associated with it. However, unlike tables, aliases can refer to each other in a process called *chaining*. Aliases are publicly referenced names, so no special authority or privilege is required to use them. Access to the table or the view referred to by an alias, however, does require the authorization associated with these objects.

There are other types of aliases, such as database and network aliases. Aliases can also be created for *nicknames* that refer to data tables or views located on federated systems.

Packages

A *package* is an object produced during program preparation that contains all of the sections in a single source file. A *section* is the compiled form of an SQL statement. Although every section corresponds to one statement, not every statement has a section. The sections created for static SQL are comparable to the bound, or operational, form of SQL statements. The sections created for dynamic SQL are comparable to placeholder control structures used at run time.

Authorization, privileges, and object ownership

Users (identified by an authorization ID) can successfully execute SQL or XQuery statements only if they have the authority to perform the specified function. 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 forth.

Authorization, privileges, and object ownership

There are three forms of authorization, *administrative authority*, *privileges*, and *LBAC credentials*, discussed below.

The database manager requires that each user be specifically authorized, either implicitly or explicitly, to use each database function needed to perform a specific task. *Explicit* authorities or privileges are granted to the user (GRANTEETYPE of U in the database catalogs). *Implicit* authorities or privileges are granted to a group to which the user belongs (GRANTEETYPE of G in the database catalogs) or to a role in which the user, the group or another role is a member (GRANTEETYPE of R in the database catalogs).

Administrative authority

The person or persons holding administrative authority are charged with the task of controlling the database manager and are responsible for the safety and integrity of the data. Those with administrative authority levels of SYSADM and DBADM implicitly have all privileges on all objects except objects pertaining to database security and control who will have access to the database manager and the extent of this access.

Authority levels provide a method of grouping privileges and higher-level database manager maintenance and utility operations. *Database authorities* enable users to perform activities at the database level. A user, group, or role can have one or more of the following authorities:

- Administrative authority level that operates at the instance level, SYSADM (system administrator)

The SYSADM authority level provides control over all the resources created and maintained by the database manager. The system administrator possesses all the authorities of DBADM, SYSCTRL, SYSMANT, and SYSMON, and the authority to grant and revoke DBADM authority and SECADM authority.

The user who has SYSADM authority is responsible both for controlling the database manager, and for ensuring the safety and integrity of the data.

SYSADM authority provides implicit DBADM authority within a database but does not provide implicit SECADM authority within a database.

- Administrative authority levels that operate at the database level:

- DBADM (database administrator)

The DBADM authority level applies at the database level and provides administrative authority over a single database. This database administrator possesses the privileges required to create objects, issue database commands, and access table data. The database administrator can also grant and revoke CONTROL and individual privileges.

- SECADM (security administrator)

The SECADM authority level applies at the database level and is the authority required to create, alter (where applicable), and drop roles, trusted contexts, audit policies, security label components, security policies, and security labels, which are used to protect tables. It is also the authority required to grant and revoke roles, security labels and exemptions as well as to grant and revoke the SETSESSIONUSER privilege. A user with the SECADM authority can transfer the ownership of objects that they do not own. They can also use the AUDIT statement to associate an audit policy with a particular database or database object at the server.

The SECADM authority has no inherent privilege to access data stored in tables and has no other additional inherent privilege. It can only be granted

Authorization, privileges, and object ownership

by a user with SYSADM authority. The SECADM authority can be granted to a user but cannot be granted to a group, a role or to PUBLIC.

- System control authority levels that operate at the instance level:

- SYSCTRL (system control)

The SYSCTRL authority level provides control over operations that affect system resources. For example, a user with SYSCTRL authority can create, update, start, stop, or drop a database. This user can also start or stop an instance, but cannot access table data. Users with SYSCTRL authority also have SYSMON authority.

- SYSMOINT (system maintenance)

The SYSMOINT authority level provides the authority required to perform maintenance operations on all databases associated with an instance. A user with SYSMOINT authority can update the database configuration, backup a database or table space, restore an existing database, and monitor a database. Like SYSCTRL, SYSMOINT does not provide access to table data. Users with SYSMOINT authority also have SYSMON authority.

- The SYSMON (system monitor) authority level

SYSMON provides the authority required to use the database system monitor. It operates at the instance level.

- Database authorities

To perform activities such as creating a table or a routine, or for loading data into a table, specific database authorities are required. For example, the LOAD database authority is required for use of the load utility to load data into tables (a user must also have INSERT privilege on the table).

Figure 4 illustrates the relationship between authorities and their span of control (database, database manager).

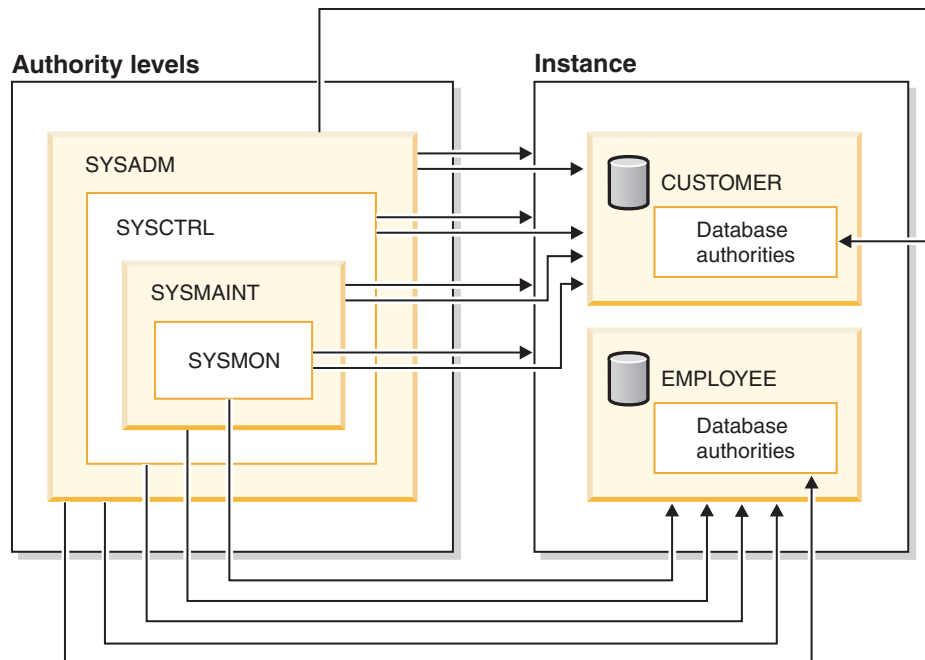


Figure 4. Hierarchy of Authorities

Authorization, privileges, and object ownership

Privileges

Privileges are those activities that a user is allowed to perform. Authorized users can create objects, have access to objects they own, and can pass on privileges on their own objects to other users by using the GRANT statement.

Privileges may be granted to individual users, to groups, or to PUBLIC. PUBLIC is a special group that consists of all users, including future users. Users that are members of a group will indirectly take advantage of the privileges granted to the group, where groups are supported.

The CONTROL privilege: Possessing the CONTROL privilege on an object allows a user to access that database object, and to grant and revoke privileges to or from other users on that object.

Note: The CONTROL privilege only applies to tables, views, nicknames, indexes, and packages.

If a different user requires the CONTROL privilege to that object, a user with SYSADM or DBADM authority could grant the CONTROL privilege to that object. The CONTROL privilege cannot be revoked from the object owner, however, the object owner can be changed by using the TRANSFER OWNERSHIP statement.

In some situations, the creator of an object automatically obtains the CONTROL privilege on that object.

Individual privileges: Individual privileges can be granted to allow a user to carry out specific tasks on specific objects. Users with administrative authority (SYSADM or DBADM) or the CONTROL privilege can grant and revoke privileges to and from users.

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, schema, package, routine, and sequence privileges to others can be extended to other users through the WITH GRANT OPTION on the GRANT statement. However, the WITH GRANT OPTION does not allow the person granting the privilege to revoke the privilege once granted. You must have SYSADM authority, DBADM authority, or the CONTROL privilege to revoke the privilege.

Privileges on objects in a package or routine: When a user has the privilege to execute a package or routine, they do not necessarily require specific privileges on the objects used in the package or routine. If the package or routine contains static SQL or XQuery statements, the privileges of the owner of the package are used for those statements. If the package or routine contains dynamic SQL or XQuery statements, the authorization ID used for privilege checking depends on the setting of the DYNAMICRULES bind option of the package issuing the dynamic query statements, and whether those statements are issued when the package is being used in the context of a routine.

A user or group can be authorized for any combination of individual privileges or authorities. When a privilege is associated with an object, that object must exist. For example, a user cannot be given the SELECT privilege on a table unless that table has previously been created.

Authorization, privileges, and object ownership

Note: Care must be taken when an authorization name representing a user or a group is granted authorities and privileges and there is no user, or group created with that name. At some later time, a user or a group can be created with that name and automatically receive all of the authorities and privileges associated with that authorization name.

The REVOKE statement is used to revoke previously granted privileges. The revoking of a privilege from an authorization name revokes the privilege granted by all authorization names.

Revoking a privilege from an authorization name does not revoke that same privilege from any other authorization names that were granted the privilege by that authorization name. For example, assume that CLAIRE grants SELECT WITH GRANT OPTION to RICK, then RICK grants SELECT to BOBBY and CHRIS. If CLAIRE revokes the SELECT privilege from RICK, BOBBY and CHRIS still retain the SELECT privilege.

LBAC credentials

Label-based access control (LBAC) lets the security administrator decide exactly who has write access and who has read access to individual rows and individual columns. The security administrator configures the LBAC system by creating security policies. A security policy describes the criteria used to decide who has access to what data. Only one security policy can be used to protect any one table but different tables can be protected by different security policies.

After creating a security policy, the security administrator creates database objects, called security labels and exemptions that are part of that policy. A security label describes a certain set of security criteria. An exemption allows a rule for comparing security labels not to be enforced for the user who holds the exemption, when they access data protected by that security policy.

Once created, a security label can be associated with individual columns and rows in a table to protect the data held there. Data that is protected by a security label is called protected data. A security administrator allows users access to protected data by granting them security labels. When a user tries to access protected data, that user's security label is compared to the security label protecting the data. The protecting label blocks some security labels and does not block others.

Object ownership

When an object is created, one authorization ID is assigned *ownership* of the object. Ownership means the user is authorized to reference the object in any applicable SQL or XQuery statement.

When an object is created within a schema, the authorization ID of the statement must have the required privilege to create objects in the implicitly or explicitly specified schema. That is, the authorization name must either be the owner of the schema, or possess the CREATEIN privilege on the schema.

Note: This requirement is not applicable when creating table spaces, buffer pools or database partition groups. These objects are not created in schemas.

When an object is created, the authorization ID of the statement is the definer of that object and by default becomes the owner of the object after it is created.

Authorization, privileges, and object ownership

Note: One exception exists. If the `AUTHORIZATION` option is specified for the `CREATE SCHEMA` statement, any other object that is created as part of the `CREATE SCHEMA` operation is owned by the authorization ID specified by the `AUTHORIZATION` option. Any objects that are created in the schema after the initial `CREATE SCHEMA` operation, however, are owned by the authorization ID associated with the specific `CREATE` statement.

For example, the statement `CREATE SCHEMA SCOTTSTUFF AUTHORIZATION SCOTT CREATE TABLE T1 (C1 INT)` creates the schema `SCOTTSTUFF` and the table `SCOTTSTUFF.T1`, which are both owned by `SCOTT`. Assume that the user `BOBBY` is granted the `CREATEIN` privilege on the `SCOTTSTUFF` schema and creates an index on the `SCOTTSTUFF.T1` table. Because the index is created after the schema, `BOBBY` owns the index on `SCOTTSTUFF.T1`.

Privileges are assigned to the object owner based on the type of object being created:

- The `CONTROL` privilege is implicitly granted on newly created tables, indexes, and packages. This privilege allows the object creator to access the database object, and to grant and revoke privileges to or from other users on that object. If a different user requires the `CONTROL` privilege to that object, a user with `SYSADM` or `DBADM` authority must grant the `CONTROL` privilege to that object. The `CONTROL` privilege cannot be revoked by the object owner.
- The `CONTROL` privilege is implicitly granted on newly created views if the object owner has the `CONTROL` privilege on all the tables, views, and nicknames referenced by the view definition.
- Other objects like triggers, routines, sequences, table spaces, and buffer pools do not have a `CONTROL` privilege associated with them. The object owner does, however, automatically receive each of the privileges associated with the object (and can provide these privileges to other users, where supported, by using the `WITH GRANT` option of the `GRANT` statement). In addition, the object owner can alter, add a comment on, or drop the object. These authorizations are implicit for the object owner and cannot be revoked.

Certain privileges on the object, such as altering a table, can be granted by the owner, and can be revoked from the owner by a user who has `SYSADM` or `DBADM` authority. Certain privileges on the object, such as commenting on a table, cannot be granted by the owner and cannot be revoked from the owner. Use the `TRANSFER OWNERSHIP` statement to move these privileges to another user. When an object is created, the authorization ID of the statement is the definer of that object and by default becomes the owner of the object after it is created. However, when a package is created and the `OWNER bind` option is specified, the owner of objects created by the static SQL statements in the package is the value of the `OWNER bind` option. In addition, if the `AUTHORIZATION` clause is specified on a `CREATE SCHEMA` statement, the authorization name specified after the `AUTHORIZATION` keyword is the owner of the schema.

A security administrator or the object owner can use the `TRANSFER OWNERSHIP` statement to change the ownership of a database object. An administrator can therefore create an object on behalf of an authorization ID, by creating the object using the authorization ID as the qualifier, and then using the `TRANSFER OWNERSHIP` statement to transfer the ownership that the administrator has on the object to the authorization ID.

System catalog views

The database manager maintains a set of tables and views that contain information about the data under its control. These tables and views are collectively known as the *system catalog*.

The system catalog contains information about the logical and physical structure of database objects such as tables, views, indexes, packages, and functions. It also contains statistical information. The database manager ensures that the descriptions in the system catalog are always accurate.

The system catalog views are like any other database view. SQL statements can be used to query the data in the system catalog views. A set of updatable system catalog views can be used to modify certain values in the system catalog.

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 to prevent uncommitted changes made by one application process from being accidentally perceived by any other process. 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 the case of a deadlock, or a lock time-out situation. An application process can explicitly request that its database changes be backed out. This is done using a *rollback* operation.

A *unit of work* is a recoverable sequence of operations within an application process. A unit of work is initiated when an application process is started, or 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 is ending.

As long as these changes remain uncommitted, other application processes are unable to perceive them, and they can be backed out. This is not true, however, when the isolation level is uncommitted read (UR). Once committed, these database changes are accessible by other application processes and can no longer be backed out through a rollback.

Application processes, concurrency, and recovery

Both DB2 call level interface (CLI) and embedded SQL allow for 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.

Locks acquired by the database manager on behalf of an application process are held until the end of a unit of work. This is not true, however, when the isolation level is cursor stability (CS, in which the lock is released as the cursor moves from row to row) or uncommitted read (UR, in which locks are not obtained).

An application process is never prevented from performing operations because of its own locks. However, if an application uses concurrent transactions, the locks from one transaction may affect the operation of a concurrent transaction.

The initiation and the termination of a unit of work define points of consistency within an application process. For example, a banking transaction may involve the transfer of funds from one account to another. Such a transaction would require that these funds be subtracted from the first account, and then added to the second account. Following the subtraction step, the data is inconsistent. Only after the funds 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.

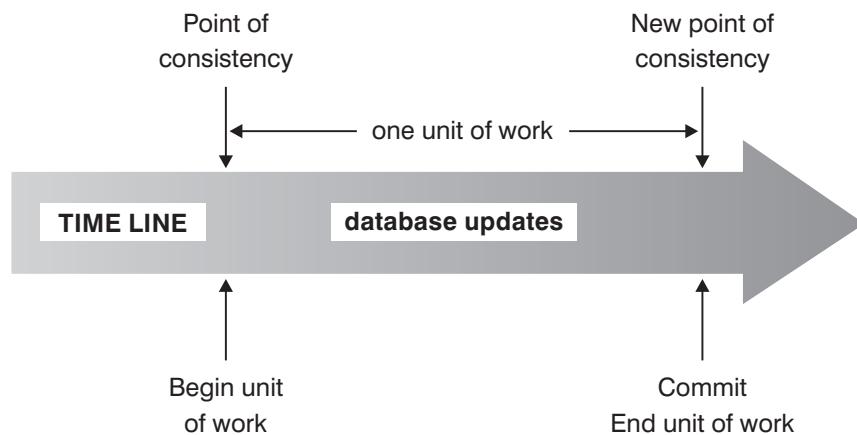


Figure 5. Unit of Work with a COMMIT Statement

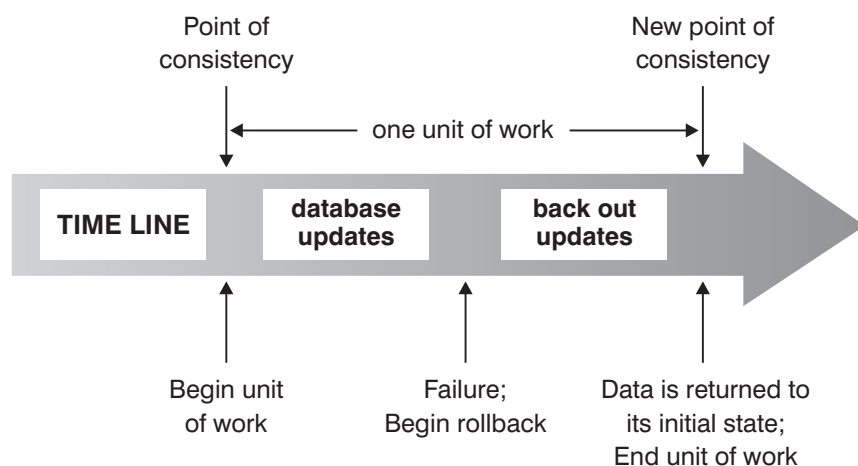


Figure 6. Unit of Work with a ROLLBACK Statement

Isolation levels

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 therefore specifies:

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

The isolation level for static SQL statements 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 by setting the ISOLATION bind or precompile option. For dynamic SQL statements, the default isolation level is the isolation level specified for the package preparing the statement. Use of the SET CURRENT ISOLATION statement allows for alternate isolation levels to be specified for dynamic SQL issued within a session. For more information, see “CURRENT ISOLATION special register”. For both static SQL statements and dynamic SQL statements, the isolation-clause in a select-statement overrides both the special register (if set) and the bind option value. For more information, see “Select-statement”.

Depending on the type of lock, this limits or prevents access to the data by concurrent application processes. Declared temporary tables and their rows cannot be locked because they are only accessible to the application that declared them.

The database manager supports three general categories of locks:

Share Limits concurrent application processes to read-only operations on the data.

Update

Limits concurrent application processes to read-only operations on the data, if these processes have not declared that they might update the row. The database manager assumes that the process currently looking at a row may update it.

Exclusive

Prevents concurrent application processes from accessing the data in any

Isolation levels

way. Does not apply to application processes with an isolation level of *uncommitted read*, which can read but not modify the data.

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 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)

This level ensures that:

- Any row read during a unit of work is not changed by other application processes until the unit of work is complete. The rows are read in the same unit of work as the corresponding OPEN statement. Use of the optional WITH RELEASE clause on the CLOSE statement means that any guarantees against non-repeatable reads and phantom reads no longer apply to any previously accessed rows if the cursor is reopened.
- Any row changed by another application process cannot be read until it is committed by that application process.

The Repeatable Read level does not allow phantom rows to be viewed (see Read Stability).

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

- Read Stability (RS)

Like the Repeatable Read level, the Read Stability level ensures that:

- Any row read during a unit of work is not changed by other application processes until the unit of work is complete. The rows are read in the same unit of work as the corresponding OPEN statement. Use of the optional WITH RELEASE clause on the CLOSE statement means that any guarantees against non-repeatable reads no longer apply to any previously accessed rows if the cursor is reopened.
- Any row changed by another application process cannot be read until it is committed by that application process.

Unlike Repeatable Read, Read Stability does not completely isolate the application process from the effects of concurrent application processes. At the RS level, application processes that issue the same query more than once may see additional rows caused by other application processes appending new information to the database. These additional rows are called *phantom rows*.

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

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 new 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 the RS isolation level acquires at least share locks on all the qualifying rows.

- Cursor Stability (CS)

Like the Repeatable Read level, the Cursor Stability level ensures that any row that was changed by another application process cannot be read until it is committed by that application process.

Unlike Repeatable Read, Cursor Stability 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 the CS isolation level acquires at least a share lock on the current row of every cursor.

- Uncommitted Read (UR)

For SELECT INTO, FETCH with a read-only cursor, fullselect in an INSERT, row fullselect in an UPDATE, or scalar fullselect (wherever it is used), the Uncommitted Read level allows:

- Any row read during a unit of work to be changed by other application processes.
- Any row changed by another application process to be read, even if the change has not been committed by that application process.

For other operations, rules associated with the CS level apply.

Comparison of isolation levels

The following table summarizes information about isolation levels.

	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? <i>See phenomenon P3 (phantom) below.</i>	Yes	Yes	Yes	No ³
Can “updated” rows be updated by other application processes? <i>See Note 1 below.</i>	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? <i>See phenomenon P2 (nonrepeatable read) below.</i>	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? <i>See phenomenon P1 (dirty-read) below.</i>	See Note 2 below.	See Note 2 below.	No	No

Isolation levels

UR CS RS RR

Note:

1. The isolation level offers no protection to the application if the application is both reading and writing a table. For example, an application opens a cursor on a table and then performs an insert, update, or delete operation on the same table. The application may see inconsistent data when more rows are fetched from the open cursor.
 2. If the cursor is not updatable, with CS the current row may be updated or deleted by other application processes in some cases. For example, buffering may cause the current row at the client to be different than what the current row actually is at the server.
 3. If your label-based access control (LBAC) credentials change between reads, the results of the second read might be different because you have access to different rows.
-

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 and performs a COMMIT. If UW1 then repeats the initial read with the same search condition, it obtains the original rows plus the inserted rows.
-

Table spaces

A *table space* is a storage structure containing tables, indexes, large objects, and long data. Table spaces reside in database partition groups. They allow you to assign the location of database and table data directly onto containers. (A container can be a directory name, a device name, or a file name.) This can provide improved performance and more flexible configuration.

Since table spaces reside in database partition groups, the table space selected to hold a table defines how the data for that table is distributed across the database partitions in a database partition group. A single table space can span several containers. It is possible for multiple containers (from one or more table spaces) to be created on the same physical disk (or drive). If you are using automatic storage table spaces, this is handled by the database manager. If you are *not* using automatic storage table spaces, for improved performance, each container should use a different disk.

Figure 7 on page 21 illustrates the relationship between tables and table spaces within a database, and the containers associated with that database.

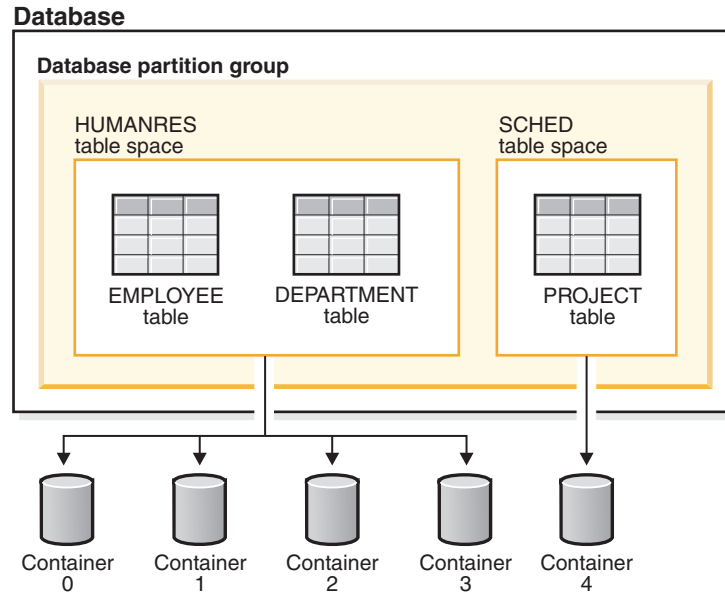


Figure 7. Table spaces and tables in a database

The EMPLOYEE and DEPARTMENT tables are in the HUMANRES table space, which spans containers 0, 1, 2 and 3. The PROJECT table is in the SCHED table space in container 4. This example shows each container existing on a separate disk.

The database manager attempts to balance the data load across containers. As a result, all containers are used to store data. The number of pages that the database manager writes to a container before using a different container is called the *extent size*. The database manager does not always start storing table data in the first container.

Figure 8 on page 22 shows the HUMANRES table space with an extent size of two 4 KB pages, and four containers, each with a small number of allocated extents. The DEPARTMENT and EMPLOYEE tables both have seven pages, and span all four containers.

Character conversion

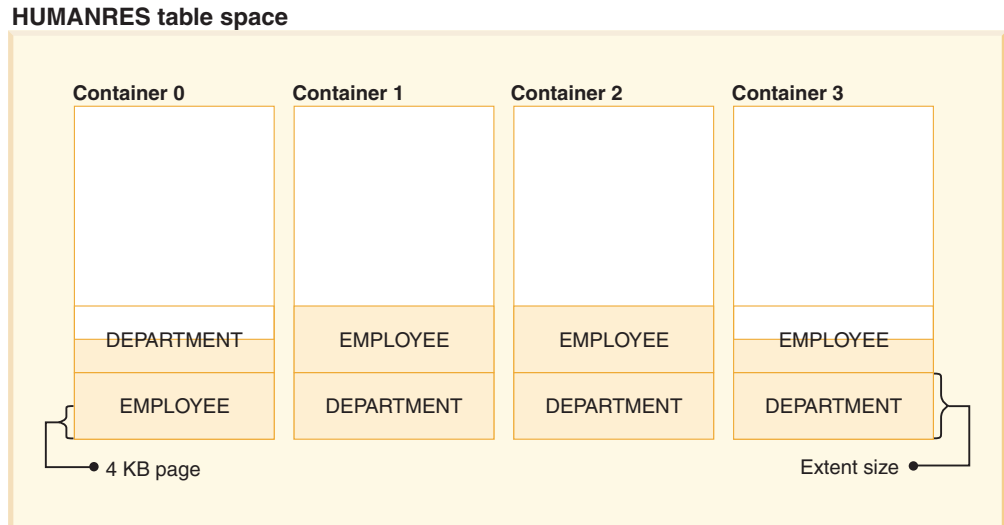


Figure 8. Containers and extents in a table space

Character conversion

A *string* is a sequence of bytes that may represent characters. All the characters within a string have a common coding representation. In some cases, it may be necessary to convert these characters to a different coding representation, a process known as *character conversion*. Character conversion, when required, is automatic, and when successful, it is transparent to the application.

Character conversion can occur when an SQL statement is executed remotely. Consider, for example, the following scenarios in which the coding representations may be different at the sending and receiving systems:

- The values of host variables are sent from the application requester to the application server.
- The values of result columns are sent from the application server to the application requester.

Following is a list of 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.

code point

A unique bit pattern that represents a character.

encoding scheme

A set of rules used to represent character data, for example:

- Single-Byte ASCII
- Single-Byte EBCDIC
- Double-Byte ASCII
- Mixed single- and double-byte ASCII

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

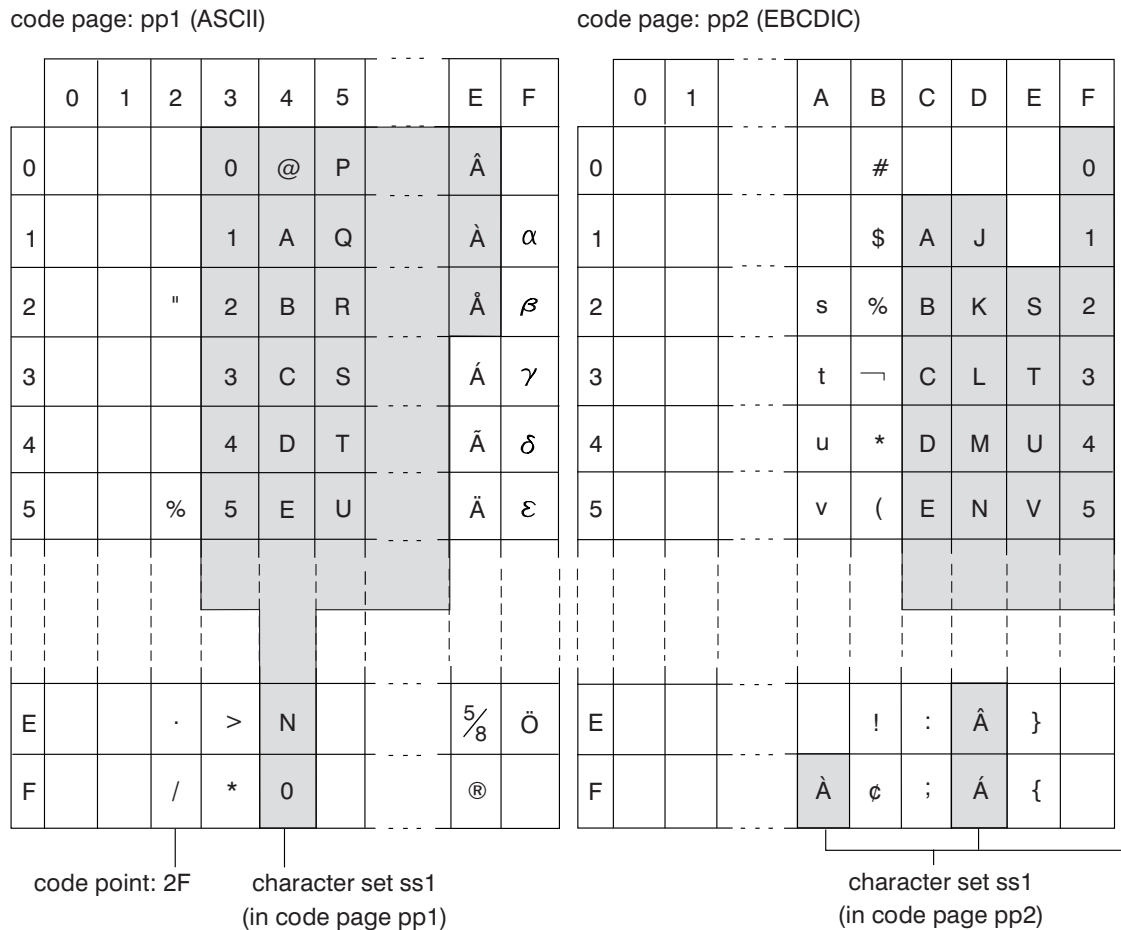


Figure 9. Mapping a Character Set in Different Code Pages

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

Database code page

The database code page is stored in the database configuration file. The value is specified when the database is created and cannot be altered.

Character conversion

Application code page

The code page under which the application runs. This is not necessarily the same code page under which the application was bound.

Section code page

The code page under which the SQL statement runs. Typically, the section code page is the database code page. However, the Unicode code page (UTF-8) is used as the section code page if:

- The statement references a table that is created with the Unicode encoding scheme in a non-Unicode database
- The statement references a table function that is defined with `PARAMETER CCSID UNICODE` in a non-Unicode database

Code Page 0

This represents a string that is derived from an expression that contains a `FOR BIT DATA` value or a `BLOB` value.

Character string code pages have the following attributes:

- Columns can be in the database code page, the Unicode code page (UTF-8), or code page 0 (if defined as `FOR BIT DATA` or `BLOB`).
- Constants and special registers (for example, `USER`, `CURRENT SERVER`) are in the section code page. Constants are converted, if necessary, from the application code page to the database code page, and then to the section code page when an SQL statement is bound to the database.
- Input host variables are in the application code page. As of Version 8, string data in input host variables is converted, if necessary, from the application code page to the section code page before being used. The exception occurs when a host variable is used in a context where it is to be interpreted as bit data; for example, when the host variable is to be assigned to a column that is defined as `FOR BIT DATA`.

A set of rules is used to determine code page attributes for operations that combine string objects, such as scalar operations, set operations, or concatenation. Code page attributes are used to determine requirements for code page conversion of strings at run time.

National language support and SQL statements

The coding of SQL statements is not language dependent. The SQL keywords must be typed as shown, although they may be typed in uppercase, lowercase, or mixed case. The names of database objects, host variables and program labels that occur in an SQL statement must be characters supported by your application code page.

The server does not convert file names. To code a file name, either use the ASCII invariant set, or provide the path in the hexadecimal values that are physically stored in the file system.

In a multi-byte environment, there are four characters which are considered special that do not belong to the invariant character set. These characters are:

- The double-byte percentage and double-byte underscore characters used in `LIKE` processing.
- The double-byte space character, used for, among other things, blank padding in graphic strings.

- The double-byte substitution character, used as a replacement during code page conversion when no mapping exists between a source code page and a target code page.

The code points for each of these characters, by code page, is as follows:

Table 1. Code Points for Special Double-Byte Characters

Code Page	Double-Byte Percentage	Double-Byte Underscore	Double-Byte Space	Double-Byte Substitution Character
932	X'8193'	X'8151'	X'8140'	X'FCFC'
938	X'8193'	X'8151'	X'8140'	X'FCFC'
942	X'8193'	X'8151'	X'8140'	X'FCFC'
943	X'8193'	X'8151'	X'8140'	X'FCFC'
948	X'8193'	X'8151'	X'8140'	X'FCFC'
949	X'A3A5'	X'A3DF'	X'A1A1'	X'AFFE'
950	X'A248'	X'A1C4'	X'A140'	X'C8FE'
954	X'A1F3'	X'A1B2'	X'A1A1'	X'F4FE'
964	X'A2E8'	X'A2A5'	X'A1A1'	X'FDFF'
970	X'A3A5'	X'A3DF'	X'A1A1'	X'AFFE'
1381	X'A3A5'	X'A3DF'	X'A1A1'	X'FEFE'
1383	X'A3A5'	X'A3DF'	X'A1A1'	X'A1A1'
13488	X'FF05'	X'FF3F'	X'3000'	X'FFFD'
1363	X'A3A5'	X'A3DF'	X'A1A1'	X'A1E0'
1386	X'A3A5'	X'A3DF'	X'A1A1'	X'FEFE'
5039	X'8193'	X'8151'	X'8140'	X'FCFC'

For Unicode databases, the GRAPHIC space is X'0020', which is different from the GRAPHIC space of X'3000' used for euc-Japan and euc-Taiwan databases. Both X'0020' and X'3000' are space characters in the Unicode standard. The difference in the GRAPHIC space code points should be taken into consideration when comparing data from these EUC databases to data from a Unicode database.

Connecting to distributed relational databases

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 a *database server* at the other end of the connection. Working together, the application requester and the database server handle communication and location considerations, so that the application can operate as if it were accessing a local database.

An application process must connect to a database manager's application server before SQL statements that reference tables or views can be executed. The CONNECT statement establishes a connection between an application process and its server.

Connecting to distributed relational databases

There are two types of CONNECT statements:

- CONNECT (Type 1) supports the single database per unit of work (Remote Unit of Work) semantics.
- CONNECT (Type 2) supports the multiple databases per unit of work (Application-Directed Distributed Unit of Work) semantics.

The DB2 call level interface (CLI) and embedded SQL support a connection mode called *concurrent transactions*, which allows multiple connections, each of which is an independent transaction. An application can have multiple concurrent connections to the same database.

The application server can be local to or remote from the environment in which the process is initiated. An application server is present, even if the environment is not using distributed relational databases. This environment includes a local directory that describes the application servers that can be identified in a CONNECT statement.

The application server runs the bound form of a static SQL statement that references tables or views. The bound statement is taken from a package that the database manager has previously created through a bind operation.

For the most part, an application connected to an application server can use statements and clauses that are supported by the application server's database manager. This is true even if an application is running through the application requester of a database manager that does *not* support some of those statements and clauses.

Event monitors

Event monitors are used to collect information about the database and any connected applications when specified events occur. Events represent transitions in database activity such as connections, deadlocks, statements, or transactions. You can define an event monitor by the type of event or events you want it to monitor. For example, a deadlock event monitor waits for a deadlock to occur; when one does, it collects information about the applications involved and the locks in contention.

By default, all databases have an event monitor defined named DB2DETAILDEADLOCK, which records detailed information about deadlock events. The DB2DETAILDEADLOCK event monitor starts automatically when the database starts.

Whereas the snapshot monitor is typically used for preventative maintenance and problem analysis, event monitors are used to alert administrators to immediate problems or to track impending ones.

To create an event monitor, use the CREATE EVENT MONITOR SQL statement. Event monitors collect event data only when they are active. To activate or deactivate an event monitor, use the SET EVENT MONITOR STATE SQL statement. The status of an event monitor (whether it is active or inactive) can be determined by the SQL function EVENT_MON_STATE.

When the CREATE EVENT MONITOR SQL statement is executed, the definition of the event monitor it creates is stored in the following database system catalog tables:

- SYSCAT.EVENTMONITORS: event monitors defined for the database.
- SYSCAT.EVENTS: events monitored for the database.
- SYSCAT.EVENTTABLES: target tables for table event monitors.

Each event monitor has its own private logical view of the instance's data in the monitor elements. If a particular event monitor is deactivated and then reactivated, its view of these counters is reset. Only the newly activated event monitor is affected; all other event monitors will continue to use their view of the counter values (plus any new additions).

Event monitor output can be directed to non-partitioned SQL tables, a file, or a named pipe.

Database partitioning across multiple database partitions

The database manager allows great flexibility in spreading data across multiple database partitions (nodes) of a partitioned database. Users can choose how to distribute their data by declaring distribution keys, and can determine which and how many database partitions their table data can be spread across by selecting the database partition group and table space in which the data should be stored.

In addition, a distribution map (which is updatable) specifies the mapping of distribution key values to database 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 database partitions if the application designer so chooses. Each local database partition may have local indexes on the data it stores to provide high performance local data access.

In a partitioned database, the distribution key is used to distribute table data across a set of database partitions. Index data is also partitioned with its corresponding tables, and stored locally at each database partition.

Before database partitions can be used to store data, they must be defined to the database manager. Database partitions are defined in a file called `db2nodes.cfg`.

The distribution key for a table in a table space on a partitioned database partition group is specified in the `CREATE TABLE` statement or the `ALTER TABLE` statement. If not specified, a distribution key for a table is created by default from the first column of the primary key. If no primary key is defined, the default distribution key is the first column defined in that table that has a data type other than a long or a LOB data type. Tables in partitioned databases must have at least one column that is neither a long nor a LOB data type. A table in a table space that is in a single partition database partition group will have a distribution key only if it is explicitly specified.

Rows are placed in a database partition as follows:

1. A hashing algorithm (database partitioning function) is applied to all of the columns of the distribution key, which results in the generation of a distribution map index value.
2. The database partition number at that index value in the distribution map identifies the database partition in which the row is to be stored.

Database partitioning across multiple database partitions

The database manager supports *partial declustering*, which means that a table can be distributed across a subset of database partitions in the system (that is, a database partition group). Tables do not have to be distributed across all of the database partitions in the system.

The database manager has the capability of recognizing when data being accessed for a join or a subquery is located at the same database partition in the same database partition group. This is known as *table collocation*. Rows in collocated tables with the same distribution key values are located on the same database partition. The database manager can choose to perform join or subquery processing at the database partition in which the data is stored. This can have significant performance advantages.

Collocated tables must:

- Be in the same database partition group, one that is not being redistributed. (During redistribution, tables in the database partition group may be using different distribution maps – they are not collocated.)
- Have distribution keys with the same number of columns.
- Have the corresponding columns of the distribution key be database partition-compatible.
- Be in a single partition database partition group defined on the same database partition.

Large object behavior in partitioned tables

A partitioned table uses a data organization scheme in which table data is divided across multiple storage objects, called data partitions or ranges, according to values in one or more table partitioning key columns of the table. Data from a given table is partitioned into multiple storage objects based on the specifications provided in the `PARTITION BY` clause of the `CREATE TABLE` statement. These storage objects can be in different table spaces, in the same table space, or a combination of both.

A large object for a partitioned table is, by default, stored in the same table space as its corresponding data object. This applies to partitioned tables that use only one table space or use multiple table spaces. When a partitioned table's data is stored in multiple table spaces, the large object data is also stored in multiple table spaces.

Use the `LONG IN` clause of the `CREATE TABLE` statement to override this default behavior. You can specify a list of table spaces for the table where long data is to be stored. If you choose to override the default behavior, the table space specified in the `LONG IN` clause must be a large table space. If you specify that long data be stored in a separate table space for one or more data partitions, you must do so for all the data partitions of the table. That is, you cannot have long data stored remotely for some data partitions and stored locally for others. Whether you are using the default behavior or the `LONG IN` clause to override the default behavior, a long object is created to correspond to each data partition. For SMS table spaces, the long data must reside in the same table space as the data object it belongs to. All the table spaces used to store long data objects corresponding to each data partition must have the same: pagesize, extentsize, storage mechanism (DMS or SMS), and type (regular or large). Remote large table spaces must be of type `LARGE` and cannot be SMS.

For example, the following `CREATE TABLE` statement creates objects for the `CLOB` data for each data partition in the same table space as the data:

```
CREATE TABLE document(id INT, contents CLOB)
PARTITION BY RANGE(id)
(STARTING FROM 1 ENDING AT 100 IN tbsp1,
 STARTING FROM 101 ENDING AT 200 IN tbsp2,
 STARTING FROM 201 ENDING AT 300 IN tbsp3,
 STARTING FROM 301 ENDING AT 400 IN tbsp4);
```

You can use LONG IN to place the CLOB data in one or more large table spaces, distinct from those the data is in.

```
CREATE TABLE document(id INT, contents CLOB)
PARTITION BY RANGE(id)
(STARTING FROM 1 ENDING AT 100 IN tbsp1 LONG IN large1,
 STARTING FROM 101 ENDING AT 200 IN tbsp2 LONG IN large1,
 STARTING FROM 201 ENDING AT 300 IN tbsp3 LONG IN large2,
 STARTING FROM 301 ENDING AT 400 IN tbsp4 LONG IN large2);
```

Note: Only a single LONG IN clause is allowed at the table level and for each data partition.

DB2 federated systems

Federated systems

A *federated system* is a special type of distributed database management system (DBMS). A federated system consists of a DB2 instance that operates as a federated server, a database that acts as the federated database, one or more data sources, and clients (users and applications) that access the database and data sources.

With a federated system, you can send distributed requests to multiple data sources within a single SQL statement. For example, you can join data that is located in a DB2 table, an Oracle table, and an XML tagged file in a single SQL statement. The following figure shows the components of a federated system and a sample of the data sources you can access.

Federated systems

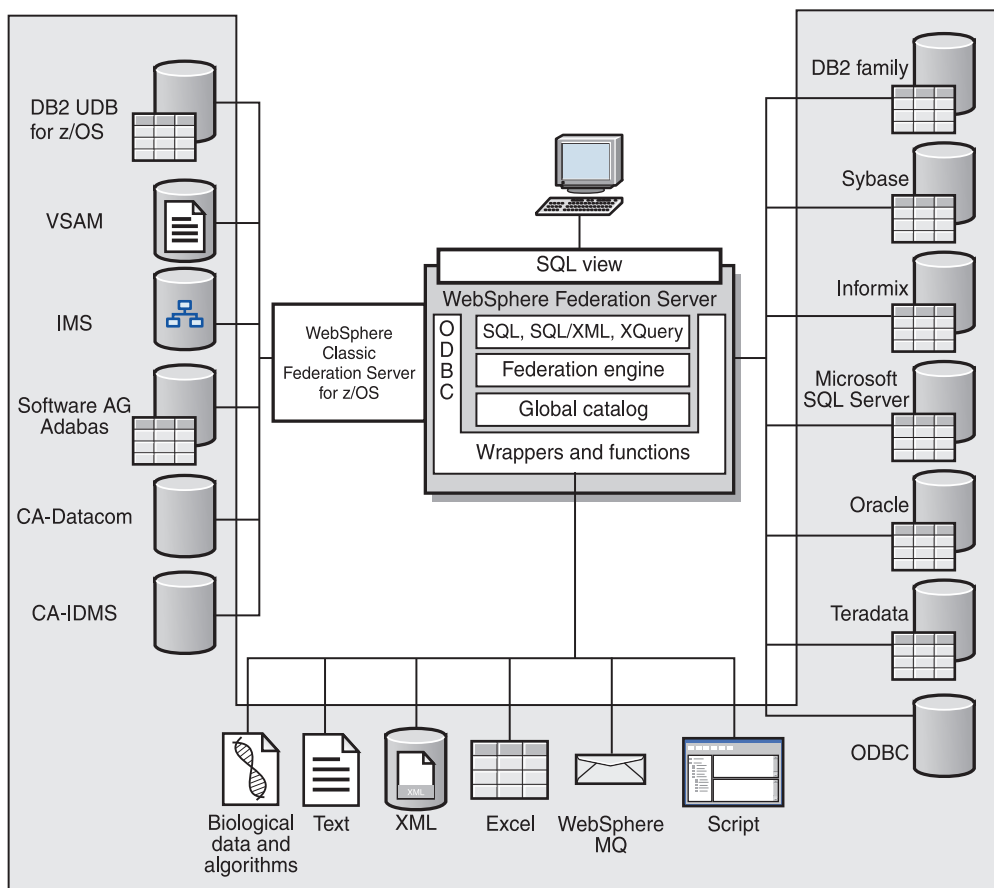


Figure 10. The components of a federated system

The power of a federated system is in its ability to:

- Correlate data from local tables and remote data sources, as if all the data is stored locally in the federated database
- Update data in relational data sources, as if the data is stored in the federated database
- Move data to and from relational data sources
- Take advantage of the data source processing strengths, by sending requests to the data sources for processing
- Compensate for SQL limitations at the data source by processing parts of a distributed request at the federated server

What is a data source?

In a federated system, a *data source* can be a relational database (such as Oracle or Sybase) or a nonrelational data source (such as a BLAST search algorithm or an XML tagged file).

Through some data sources you can access other data sources. For example, with the ODBC wrapper you can access WebSphere® Classic Federation Server for z/OS data sources such as DB2 UDB for z/OS, IMS™, CA-IDMS, CA-Datcom, Software AG Adabas, and VSAM.

The method, or protocol, used to access a data source depends on the type of data source. For example, DRDA® is used to access DB2 for z/OS data sources.

Data sources are 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 federated system does not monopolize or restrict access to the other data sources, beyond integrity and locking constraints.

The federated database

To end users and client applications, data sources appear as a single collective database in DB2. Users and applications interface with the *federated database* that is managed by the federated server.

The federated database contains a system catalog that stores information about data. The federated database system catalog contains entries that identify data sources and their characteristics. The federated server consults the information stored in the federated database system catalog and the data source wrapper to determine the best plan for processing SQL statements.

The federated system processes SQL statements as if the data from the data sources were ordinary relational tables or views within the federated database. As a result:

- The federated system can correlate relational data with data in nonrelational formats. This is true even when the data sources use different SQL dialects, or do not support SQL at all.
- The characteristics of the federated database take precedence when there are differences between the characteristics of the federated database and the characteristics of the data sources. Query results conform to DB2 semantics, even if data from other non-DB2 data sources is used to compute the query result.

Examples:

- The code page that the federated server uses is different than the code page used that the data source uses. In this case, character data from the data source is converted based on the code page used by the federated database, when that data is returned to a federated user.
- The collating sequence that the federated server uses is different than the collating sequence that the data source uses. In this case, any sort operations on character data are performed at the federated server instead of at the data source.

The SQL compiler

The DB2 SQL compiler gathers information to help it process queries.

To obtain data from data sources, users and applications submit queries in SQL to the federated database. When a query is submitted, the DB2 SQL compiler consults information in the global catalog and the data source wrapper to help it process the query. This includes information about connecting to the data source, server information, mappings, index information, and processing statistics.

Wrappers and wrapper modules

Wrappers are mechanisms by which the federated database interacts with data sources. The federated database uses routines stored in a library called a *wrapper module* to implement a wrapper.

These routines allow the federated database to perform operations such as connecting to a data source and retrieving data from it iteratively. Typically, the

Wrappers and wrapper modules

federated instance owner uses the CREATE WRAPPER statement to register a wrapper in the federated database. You can register a wrapper as fenced or trusted using the DB2_FENCED wrapper option.

You create one wrapper for each type of data source that you want to access. For example, you want to access three DB2 for z/OS database tables, one DB2 for System i table, two Informix® tables, and one Informix view. In this case, you need to create one wrapper for the DB2 data source objects and one wrapper for the Informix data source objects. After these wrappers are registered in the federated database, you can use these wrappers to access other objects from those data sources. For example, you can use the DRDA wrapper with all DB2 family data source objects—DB2 Database for Linux, UNIX, and Windows, DB2 for z/OS, DB2 for System i, and DB2 Server for VM and VSE.

You use the server definitions and nicknames to identify the specifics (name, location, and so forth) of each data source object.

A wrapper performs many tasks. Some of these tasks are:

- It connects to the data source. The wrapper uses the standard connection API of the data source.
- It submits queries to the data source.
 - For data sources that support SQL, the query is submitted in SQL.
 - For data sources that do not support SQL, the query is translated into the native query language of the source or into a series of source API calls.
- It receives results sets from the data source. The wrapper uses the data source standard APIs for receiving results set.
- It responds to federated database queries about the default data type mappings for a data source. The wrapper contains the default type mappings that are used when nicknames are created for a data source object. For relational wrappers, data type mappings that you create override the default data type mappings. User-defined data type mappings are stored in the global catalog.
- It responds to federated database queries about the default function mappings for a data source. The federated database needs data type mapping information for query planning purposes. The wrapper contains information that the federated database needs to determine if DB2 functions are mapped to functions of the data source, and how the functions are mapped. This information is used by the SQL Compiler to determine if the data source is able to perform the query operations. For relational wrappers, function mappings that you create override the default function type mappings. User-defined function mappings are stored in the global catalog.

Wrapper options are used to configure the wrapper or to define how WebSphere Federation Server uses the wrapper.

Server definitions and server options

After wrappers are created for the data sources, the federated instance owner defines the data sources to the federated database.

The instance owner supplies a name to identify the data source, and other information that pertains to the data source. This information includes:

- The type and version of the data source
- The database name for the data source (RDBMS only)
- Metadata that is specific to the data source

For example, a DB2 family data source can have multiple databases. The definition 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 knowing its name. The database name is not included in the federated server definition of an Oracle data source.

The name and other information that the instance owner supplies to the federated server are collectively called a *server definition*. Data sources answer requests for data and are servers in their own right.

The CREATE SERVER and ALTER SERVER statements are used to create and modify a server definition.

Some of the information within a server definition is stored as *server options*. When you create server definitions, it is important to understand the options that you can specify about the server.

Server options can be set to persist over successive connections to the data source, or set for the duration of a single connection.

User mappings

A *user mapping* is an association between an authorization ID on the federated server and the information that is required to connect to the remote data source.

To create a user mapping, you use the CREATE USER MAPPING statement. In the statement, you specify the local authorization ID, the local name of the remote data source server as specified in the server definition, and the remote ID and password.

For example, assume that you created a server definition for a remote server and specified 'argon' as the local name for the remote server. To give Mary access to the remote server, create this user mapping:

```
CREATE USER MAPPING FOR Mary
SERVER argon
OPTIONS (REMOTE_AUTHID 'remote_ID', REMOTE_PASSWORD 'remote_pw')
```

When Mary issues an SQL statement to connect to the remote server, the federated server performs these steps:

1. Retrieves Mary's user mapping
2. Decrypts the remote password 'remote_pw' that is associated with the remote server
3. Calls the wrapper to connect to the remote server
4. Passes the remote ID 'remote_ID' and the decrypted remote password to the wrapper
5. Creates a connection to the remote server for Mary

By default, the federated server stores user mapping in the SYSCAT.USEROPTIONS view in the global catalog and encrypts the remote passwords. As an alternative, you can use an external repository, for example a file or an LDAP server, to store user mappings. To provide the interface between the federated server and the external repository, you create a user mapping plug-in.

User mappings

No matter how you store user mappings, carefully restrict access to them. If user mappings are compromised, data in the remote databases may be vulnerable to unauthorized activity.

Nicknames and data source objects

A *nickname* is an identifier that you use to identify the data source object that you want to access. The object that a nickname identifies is referred to as *adata source object*.

A nickname is not an alternative name for a data source object in the same way that an alias is an alternative name. A nickname is the pointer by which the federated server references the object. Nicknames are typically defined with the CREATE NICKNAME statement along with specific nickname column options and nickname options.

When a client application or a user submits a distributed request to the federated server, the request does not need to specify the data sources. Instead, the request references the data source objects by their nicknames. The nicknames are mapped to specific objects at the data source. These mappings eliminate the need to qualify the nicknames by data source names. The location of the data source objects is transparent to the client application or the user.

Suppose that you define the nickname *DEPT* to represent an Informix database table called *NFX1.PERSON*. The statement `SELECT * FROM DEPT` is allowed from the federated server. However, the statement `SELECT * FROM NFX1.PERSON` is not allowed from the federated server (except in a pass-through session) unless there is a local table on the federated server named *NFX1.PERSON*.

When you create a nickname for a data source object, metadata about the object is added to the global catalog. The query optimizer uses this metadata, as well as information in the wrapper, to facilitate access to the data source object. For example, if a nickname is for a table that has an index, the global catalog contains information about the index, and the wrapper contains the mappings between the DB2 data types and the data source data types.

Nicknames for objects that use label-based access control (LBAC) are not cached. Therefore, data in the object remains secure. For example, if you use the Oracle (Net8) wrapper to create a nickname on a table that uses Oracle Label Security, the table is automatically identified as secure. The resulting nickname data cannot be cached. As a result, materialized query tables cannot be created on it. Using LBAC ensures that the information is viewed only by users who have the appropriate security privileges. For nicknames that were created before LBAC was supported, you must use the ALTER NICKNAME statement to disallow caching. LBAC is supported by both the DRDA (for data sources that use DB2 for Linux, UNIX, and Windows version 9.1 and later) and the Net8 wrapper.

Nickname column options

You can supply the global catalog with additional metadata information about the nicknamed object. This metadata describes values in certain columns of the data source object. You assign this metadata to parameters that are called *nickname column options*.

The nickname column options tell the wrapper to handle the data in a column differently than it normally would handle it. The SQL compiler and query optimizer use the metadata to develop better plans for accessing the data.

Nickname column options are used to provide other information to the wrapper as well. For example for XML data sources, a nickname column option is used to tell the wrapper the XPath expression to use when the wrapper parses the column out of the XML document.

With federation, the DB2 server treats the data source object that a nickname references as if it is a local DB2 table. As a result, you can set nickname column options for any data source object that you create a nickname for. Some nickname column options are designed for specific types of data sources and can be applied only to those data sources.

Suppose that a data source has a collating sequence that differs from the federated database collating sequence. The federated server typically would not sort any columns containing character data at the data source. It would return the data to the federated database and perform the sort locally. However, suppose that the column is a character data type (CHAR or VARCHAR) and contains only numeric characters ('0','1',..., '9'). You can indicate this by assigning a value of 'Y' to the NUMERIC_STRING nickname column option. This gives the DB2 query optimizer the option of performing the sort at the data source. If the sort is performed remotely, you can avoid the overhead of porting the data to the federated server and performing the sort locally.

You can define nickname column options for relational nicknames using the ALTER NICKNAME statements. You can define nickname column options for nonrelational nicknames using the CREATE NICKNAME and ALTER NICKNAME statements.

Data type mappings

The data types at the data source must map to corresponding DB2 data types so that the federated server can retrieve data from data sources.

Some examples of default data type mappings are:

- The Oracle type FLOAT maps to the DB2 type DOUBLE
- The Oracle type DATE maps to the DB2 type TIMESTAMP
- The DB2 for z/OS™ type DATE maps to the DB2 type DATE

For most data sources, the default type mappings are in the wrappers. The default type mappings for DB2 data sources are in the DRDA wrapper. The default type mappings for Informix are in the INFORMIX wrapper, and so forth.

For some nonrelational data sources, you must specify data type information in the CREATE NICKNAME statement. The corresponding DB2 data types must be specified for each column of the data source object when the nickname is created. Each column must be mapped to a particular field or column in the data source object.

For relational data sources, you can override the default data type mappings. For example, by default the Informix INTEGER data type maps to the DB2 INTEGER data type. You could override the default mappings and map Informix's INTEGER data type to DB2 DECIMAL(10,0) data type.

The federated server

The DB2 server in a federated system is referred to as the federated server. Any number of DB2 instances can be configured to function as federated servers. You can use existing DB2 instances as your federated servers, or you can create new ones specifically for the federated system.

The DB2 instance that manages the federated system is called a server because it responds to requests from end users and client applications. The federated server often sends parts of the requests it receives to the data sources for processing. A pushdown operation is an operation that is processed remotely. The DB2 instance that manages the federated system is referred to as the federated server, even though it acts as a client when it pushes down requests to the data sources.

Like any other application server, the federated server is a database manager instance. Application processes connect and submit requests to the database within the federated server. However, two main features distinguish it from other application servers:

- A federated server is configured to receive requests that might be partially or entirely intended for data sources. The federated server distributes these requests to the data sources.
- Like other application servers, a federated server uses DRDA communication protocols (over TCP/IP) to communicate with DB2 family instances. However, unlike other application servers, a federated server uses the native client of the data source to access the data source. For example, a federated server uses the Sybase Open Client to access Sybase data sources and an Microsoft SQL Server ODBC Driver to access Microsoft SQL Server data sources.

Supported data sources

There are many data sources that you can access using a federated system.

WebSphere Federation Server supports the data sources shown in the following tables. The first table lists the requirements for data client software. The client software must be acquired separately unless specified otherwise.

You must install the client software for the data sources that you want to access. The client software must be installed on the same system as WebSphere Federation Server. You also need the appropriate Java SDK to use some tools such as the DB2 Control Center and to create and run Java applications, including stored procedures and user-defined functions.

For the most up-to-date information, see the Data source requirements page on the Web.

Table 2. Supported data sources, client software requirements, and support from 32-bit operating systems.

			32-bit hardware architecture and operating system	
			X86-32	X86-32
Data source	Supported versions	Client software	Linux, RedHat Enterprise Linux (RHEL), SUSE	Windows
BioRS	5.2	None	Y	Y

Supported data sources

Table 2. Supported data sources, client software requirements, and support from 32-bit operating systems. (continued)

			32-bit hardware architecture and operating system	
			X86-32	X86-32
Data source	Supported versions	Client software	Linux, RedHat Enterprise Linux (RHEL), SUSE	Windows
Blast	2.2.1, 2.2.3, 2.2.4, 2.2.9	Blast daemon (supplied)	Y	Y
DB2 for Linux, UNIX, and Windows	8.1.x, 8.2.x, 9.1	None	Y	Y
DB2 for z/OS	7.x, 8.x	DB2® Connect™ V9.1	Y	Y
DB2 for iSeries®	5.2, 5.3, 5.4	DB2 Connect V9.1	Y	Y
DB2 Server for VSE and VM	7.2 , 7.4	DB2 Connect V9.1	Y	Y
Entrez		None	Y	Y
Flat files		None	Y	Y
HMMER	2.3, 2.3.2	HMMER daemon (supplied)	Y	Y
Informix	Informix XPS 8.40, 8.50, 8.51 and Informix IDS IDS 7.31, IDS 9.40, IDS 10.0	Informix Client SDK version 2.81.TC2 or later	Y	Y
Microsoft Excel	2000,2002,2003	Excel 2000,2002 or 2003		Y
Microsoft SQL Server	Microsoft SQL Server 2000/SP3, 2005	For Windows, Microsoft SQL Server Client ODBC 3.0 (or later) driver. For Unix, DataDirect ODBC 5.0 *!ENTITY!* 5.1	Y	Y
ODBC	3.0	ODBC drivers that comply with ODBC 3.0 **	Y	Y
OLE DB	2.7, 2.8	OLE DB 2.0, or later	Y	Y
Oracle	8.1.7, 9.2, 10g, 10gR2	Oracle NET client 9.0 - 9.2.0.5 & 10.0 - 10.1.x or NET8 client 8.1.7.x	Y	Y

Supported data sources

Table 2. Supported data sources, client software requirements, and support from 32-bit operating systems. (continued)

			32-bit hardware architecture and operating system	
			X86-32	X86-32
Data source	Supported versions	Client software	Linux, RedHat Enterprise Linux (RHEL), SUSE	Windows
Sybase	Sybase ASE 12.5, 15.0	Sybase Open Client 12.5 - 15.0	Y	Y
Teradata	2.4, 2.5, 2.6	For Windows Teradata client TTU 7.0 or later and the Teradata API library CLiv2 4.7.0 or later. For Unix Teradata Call-Level Interface Version 2 CLiv2 Release 04.06, 04.07		Y
WebSphere Business Integration for PeopleSoft 2.6	PeopleSoft 8.x	None	Y	Y
WebSphere Business Integration for SAP 2.6	SAP 3.x, 4.x	None	Y	Y
WebSphere Business Integration for Seibel 2.6	Seibel 7, 7.5, 2000	None	Y	Y
Web Services	WSDL 1.0, 1.1 SOAP 1.0, 1.1	None	Y	Y
XML	XML1.0, XML1.1	None	Y	Y

** ODBC can be used to access RedBrick 6.20cu5 and WebSphere Information Integrator Classic Federation V8.2.x , among other data sources.

Table 3. Support from 64-bit operating systems.

64-bit hardware architecture	X86-64	X86-64	Power	Itanium®	Power	Sparc	zSeries®
Operating system	Linux RHEL SUSE	Windows	AIX®	HP-UX	Linux RHEL SUSE	Solaris	Linux RHEL SUSE
Data source							
BioRS	Y	Y	Y	Y	Y	Y	Y
Blast	Y	Y	Y	Y	Y	Y	Y

Table 3. Support from 64-bit operating systems. (continued)

64-bit hardware architecture	X86-64	X86-64	Power	Itanium®	Power	Sparc	zSeries®
Operating system	Linux RHEL SUSE	Windows	AIX®	HP-UX	Linux RHEL SUSE	Solaris	Linux RHEL SUSE
Data source							
DB2 for Linux, UNIX, and Windows	Y	Y	Y	Y	Y	Y	Y
DB2 for z/OS	Y	Y	Y	Y	Y	Y	Y
DB2 for System i	Y	Y	Y	Y	Y	Y	Y
DB2 Server for VSE and VM	Y	Y	Y	Y	Y	Y	Y
Entrez	Y	Y	Y	Y	Y	Y	Y
HMMER	Y	Y	Y	Y	Y	Y	Y
Informix	Y		Y	Y	Y	Y	Y
Microsoft Excel							
Microsoft SQL Server	Y	Y	Y	Y		Y	
ODBC	Y	Y	Y***	Y		Y***	
OLE DB		Y		Y			
Oracle	Y	Y	Y	Y	Y	Y	Y
Script	Y	Y	Y	Y	Y	Y	Y
Sybase	Y		Y	Y	Y	Y	
Teradata			Y	Y		Y	
WebSphere Business Integration for PeopleSoft 2.6	Y	Y	Y	Y	Y	Y	Y
WebSphere Business Integration for SAP 2.6	Y	Y	Y	Y	Y	Y	Y

Supported data sources

Table 3. Support from 64-bit operating systems. (continued)

64-bit hardware architecture	X86-64	X86-64	Power	Itanium®	Power	Sparc	zSeries®
Operating system	Linux RHEL SUSE	Windows	AIX®	HP-UX	Linux RHEL SUSE	Solaris	Linux RHEL SUSE
Data source							
WebSphere Business Integration for Seibel 2.6	Y	Y	Y	Y	Y	Y	Y
Web Services	Y	Y	Y	Y	Y	Y	Y
XML	Y	Y	Y	Y	Y	Y	Y

*** ODBC can be used to access RedBrick 6.20cu5 and WebSphere Information Integrator Classic Federation using 32-bit and 64-bit clients.

The federated database system catalog

The federated database system catalog contains information about the objects in the federated database and information about objects at the data sources.

The catalog in a federated database is called the global catalog because it contains information about the entire federated system. DB2 query optimizer uses the information in the global catalog and the data source wrapper to plan the best way to process SQL statements. The information stored in the global catalog includes remote and local information, such as column names, column data types, column default values, index information, and statistics information.

Remote catalog information is the information or name used by the data source. Local catalog information is the information or name used by the federated database. For example, suppose a remote table includes a column with the name of *EMPNO*. The global catalog would store the remote column name as *EMPNO*. Unless you designate a different name, the local column name will be stored as *EMPNO*. You can change the local column name to *Employee_Number*. Users submitting queries which include this column will use *Employee_Number* in their queries instead of *EMPNO*. You use the ALTER NICKNAME statement to change the local name of the data source columns.

For relational and nonrelational data sources, the information stored in the global catalog includes both remote and local information.

To see the data source table information that is stored in the global catalog, query the SYSCAT.TABLES, SYSCAT.NICKNAMES, SYSCAT.TABOPTIONS, SYSCAT.INDEXES, SYSCAT.INDEXOPTIONS, SYSCAT.COLUMNS, and SYSCAT.COLOPTIONS catalog views in the federated database.

The global catalog also includes other information about the data sources. For example, the global catalog includes information that the federated server uses to connect to the data source and map the federated user authorizations to the data

source user authorizations. The global catalog contains attributes about the data source that you explicitly set, such as server options.

The query optimizer

As part of the SQL compiler process, the query optimizer analyzes a query. The compiler develops alternative strategies, called access plans, for processing the query.

Access plans might call for the query to be:

- Processed by the data sources
- Processed by the federated server
- Processed partly by the data sources and partly by the federated server

The query optimizer evaluates the access plans primarily on the basis of information about the data source capabilities and the data. The wrapper and the global catalog contain this information. The query optimizer decomposes the query into segments that are called query fragments. Typically it is more efficient to pushdown a query fragment to a data source, if the data source can process the fragment. However, the query optimizer takes into account other factors such as:

- The amount of data that needs to be processed
- The processing speed of the data source
- The amount of data that the fragment will return
- The communication bandwidth
- Whether there is a usable materialized query table on the federated server that represents the same query result

The query optimizer generates access plan alternatives for processing a query fragment. The plan alternatives perform varying amounts of work locally on the federated server and on the remote data sources. Because the query optimizer is cost-based, it assigns resource consumption costs to the access plan alternatives. The query optimizer then chooses the plan that will process the query with the least resource consumption cost.

If any of the fragments are to be processed by data sources, the federated database submits these fragments to the data sources. After the data sources process the fragments, the results are retrieved and returned to the federated database. If the federated database performed any part of the processing, it combines its results with the results retrieved from the data source. The federated database then returns all results to the client.

Collating sequences

The order in which character data is sorted in a database depends on the structure of the data and the collating sequence defined for the database.

Suppose that the data in a database is all uppercase letters and does not contain any numeric or special characters. A sort of the data should result in the same output, regardless of whether the data is sorted at the data source or at the federated database. The collating sequence used by each database should not impact the sort results. Likewise, if the data in the database is all lowercase letters or all numeric characters, a sort of the data should produce the same results regardless of where the sort actually is performed.

If the data consists of any of the following structures:

Collating sequences

- A combination of letters and numeric characters
- Both uppercase and lowercase letters
- Special characters such as @, #, €

Sorting this data can result in different outputs, if the federated database and the data source use different collating sequences.

In general terms, a collating sequence is a defined ordering for character data that determines whether a particular character sorts higher, lower, or the same as another character.

How collating sequences determine sort orders

A collating sequence determines the sort order of the characters in a coded character set.

A character set is the aggregate of characters that are used in a computer system or programming language. In a coded character set, each character is assigned to a different number within the range of 0 to 255 (or the hexadecimal equivalent thereof). The numbers are called code points; the assignments of numbers to characters in a set are collectively called a code page.

In addition to being assigned to a character, a code point can be mapped to the character's position in a sort order. In technical terms, then, a collating sequence is the collective mapping of a character set's code points to the sort order positions of the set's characters. A character's position is represented by a number; this number is called the weight of the character. In the simplest collating sequence, called an identity sequence, the weights are identical to the code points.

Example: Database ALPHA uses the default collating sequence of the EBCDIC code page. Database BETA uses the default collating sequence of the ASCII code page. Sort orders for character strings at these two databases would differ:

```
SELECT . . . .  
ORDER BY COL2
```

EBCDIC-Based Sort	ASCII-Based Sort
COL2	COL2
----	----
V1G	7AB
Y2W	V1G
7AB	Y2W

Example: Similarly, character comparisons in a database depend on the collating sequence defined for that database. Database ALPHA uses the default collating sequence of the EBCDIC code page. Database BETA uses the default collating sequence of the ASCII code page. Character comparisons at these two databases would yield different results:

```
SELECT . . . .  
WHERE COL2 > 'TT3'
```

EBCDIC-Based Results	ASCII-Based Results
COL2	COL2
----	----
TW4	TW4
X82	X82
39G	

Setting the local collating sequence to optimize queries

Administrators can create federated databases with a particular collating sequence that matches a data source collating sequence.

Then for each data source server definition, the `COLLATING_SEQUENCE` server option is set to 'Y'. This setting tells the federated database that the collating sequences of the federated database and the data source match.

You set the federated database collating sequence as part of the `CREATE DATABASE` command. Through this command, you can specify one of the following sequences:

- An identity sequence
- A system sequence (the sequence used by the operating system that supports the database)
- A customized sequence (a predefined sequence that DB2 supplies or that you define yourself)

Suppose that the data source is DB2 for z/OS. Sorts that are defined in an `ORDER BY` clause are implemented by a collating sequence based on an EBCDIC code page. To retrieve DB2 for z/OS data sorted in accordance with `ORDER BY` clauses, configure the federated database so that it uses the predefined collating sequence based on the appropriate EBCDIC code page.

Setting the local collating sequence to optimize queries

Chapter 2. Language elements

Characters

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

A *letter* is any of the 26 uppercase (A through Z) or 26 lowercase (a through z) letters, plus the three characters \$, #, and @, which are included for compatibility with host database products. For example, in code page 850, \$ is at X'24', # is at X'23', and @ is at X'40'. Letters also include the alphabets from the extended character sets. Extended character sets contain additional alphabetic characters; for example, those with diacritical marks (´ is an example of a diacritical mark). The available characters depend on the code page in use.

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

A *special character* is any of the characters listed below:

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

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

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

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

Tokens

Tokens are the basic syntactical units of SQL. A *token* is a sequence of one or more characters. A token cannot contain blank characters, unless it is a string constant or a delimited identifier, which may contain blanks.

Tokens are classified as ordinary or delimiter:

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

Examples

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

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

Examples

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

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

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

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

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

is equivalent, after folding, to:

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

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

For characters in Unicode:

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

Identifiers

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

- SQL identifiers

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

- An *ordinary identifier* is an uppercase letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character. Note that ordinary identifiers are converted to uppercase. An ordinary identifier should not be a reserved word.

Examples

```
WKLYSAL    WKLY_SAL
```

- A *delimited identifier* is a sequence of one or more characters enclosed by double 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

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

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

- Host identifiers

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

Naming conventions and implicit object name qualifications

The rules for forming the name of an object depend on the object type. 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, the 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. If the dynamic SQL statement is contained in a package that exhibits bind, define, or invoke behaviour, the CURRENT SCHEMA special register is not used for qualification. In a bind behaviour package, the package default qualifier is used as the value for implicit qualification of unqualified object references. In a define behaviour package, the authorization ID of the routine definer is used as the value for implicit qualification of unqualified object references within that routine. In an invoke behaviour package, the statement authorization ID in effect when the routine is invoked is used as the value for implicit qualification of unqualified object references within dynamic SQL statements within that routine. For more information, see “Dynamic SQL characteristics at run time” on page 54.

In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified database object names. By default, this value is set to the package authorization ID.

Identifiers

The following object names, when used in the context of an SQL procedure, are permitted to use only the characters allowed in an ordinary identifier, even if the names are delimited:

- condition-name
- label
- parameter-name
- procedure-name
- SQL-variable-name
- statement-name

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

alias-name

A schema-qualified name that designates an alias.

attribute-name

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

authorization-name

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

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

bufferpool-name

An identifier that designates a buffer pool.

column-name

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

component-name

An identifier that designates a security label component.

condition-name

An identifier that designates a condition in an SQL procedure.

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

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

db-partition-group-name

An identifier that designates a database partition group.

descriptor-name

A colon followed by a host identifier that designates an SQL descriptor area (SQLDA). For the description of a host identifier, see “References to host variables” on page 63. Note that a descriptor name never includes an indicator variable.

distinct-type-name

A qualified or unqualified name that designates a distinct type. 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. An unqualified function name in an SQL statement is implicitly qualified by the database manager, depending on context.

global-variable-name

A qualified or unqualified name that designates a global variable. An unqualified global variable name in an SQL statement is implicitly qualified by the database manager, depending on context.

group-name

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

host-variable

A sequence of tokens that designates a host variable. A host variable includes at least one host identifier, explained in “References to host variables” on page 63.

index-name

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

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

method-name

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

nickname

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

package-name

A schema-qualified name that designates a package. If a package has a version ID that is not the empty string, the package name also includes the version ID at the end of the name, in the form: schema-id.package-id.version-id.

parameter-name

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

Identifiers

partition-name

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

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 in which the table or view resides. The parts of this name are data-source-name, remote-schema-name, and remote-table-name.

remote-schema-name

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

remote-table-name

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

remote-type-name

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

savepoint-name

An identifier that designates a savepoint.

schema-name

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

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

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

security-label-name

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

security-policy-name

An identifier that designates a security policy.

sequence-name

An identifier that designates a sequence.

server-name

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

specific-name

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

SQL-variable-name

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

statement-name

An identifier that designates a prepared SQL statement.

supertype-name

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

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. 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.

XML-schema-name

A qualified or unqualified name that designates an XML schema.

xsobject-name

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

Aliases

A table alias can be thought of as an alternative name for a table or a 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 a view name can be used. An alias can be created even if the object does not exist (although it must exist by the time a statement referring to it is compiled). It can refer to another alias if no circular or repetitive references are made along the chain of aliases. An alias can only refer to a 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 the alias name PERSONNEL has been created, subsequent statements such as CREATE TABLE PERSONNEL... will return an error.

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

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 by the time that the SQL statement is compiled, is replaced at statement compilation time by the qualified base table or view name. For example, if PBIIRD.SALES is an alias for DSPN014.DIST4_SALES_148, then at compilation time:

```
SELECT * FROM PBIIRD.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 instead 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 applications running under other relational database management systems, SYNONYM can be used in place of ALIAS in the CREATE ALIAS and DROP ALIAS statements.

Authorization IDs and authorization names

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

Authorization IDs are used by the database manager to provide:

- Authorization checking of SQL statements

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

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

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

For more information, see "Dynamic SQL characteristics at run time" on page 54.

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

Examples:

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

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

SMITH is the authorization ID of the statement. Therefore, in a dynamic SQL statement, the default value of the CURRENT SCHEMA special register is SMITH, and in static SQL, the default value of the QUALIFIER precompile/bind option is SMITH. The authority to execute the statement is checked against SMITH, and SMITH is the *table-name* implicit qualifier based on qualification rules described in "Naming conventions and implicit object name qualifications" on page 47.

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

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

```
DROP TABLE TDEPT
```

Removes the SMITH.TDEPT table.

```
DROP TABLE SMITH.TDEPT
```

Removes the SMITH.TDEPT table.

```
DROP TABLE KEENE.TDEPT
```

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

```
CREATE SCHEMA PAYROLL AUTHORIZATION KEENE
```

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

Dynamic SQL characteristics at run time

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

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

Table 4. How DYNAMICRULES and the runtime environment determine dynamic SQL statement behavior

DYNAMICRULES value	Behavior of dynamic SQL statements	
	Standalone program environment	Routine environment
BIND	Bind behavior	Bind behavior
RUN	Run behavior	Run behavior
DEFINEBIND	Bind behavior	Define behavior
DEFINERUN	Run behavior	Define behavior
INVOKEBIND	Bind behavior	Invoke behavior
INVOKERUN	Run behavior	Invoke behavior

Run behavior

DB2 uses the authorization ID of the user (the ID that initially connected to DB2) executing the package as the value to be used for authorization checking of dynamic SQL statements and for the initial value used for implicit qualification of unqualified object references within dynamic SQL statements.

Bind behavior

At run time, DB2 uses all the rules that apply to static SQL for authorization and qualification. It takes the authorization ID of the package owner as the value to be used for authorization checking of dynamic SQL statements, and the package default qualifier for implicit qualification of unqualified object references within dynamic SQL statements.

Define behavior

Define behavior applies only if the dynamic SQL statement is in a package

that is run within a routine context, and the package was bound with DYNAMICRULES DEFINEBIND or DYNAMICRULES DEFINERUN. DB2 uses the authorization ID of the routine definer (not the routine's package binder) as the value to be used for authorization checking of dynamic SQL statements, and for implicit qualification of unqualified object references within dynamic SQL statements within that routine.

Invoke behavior

Invoke behavior applies only if the dynamic SQL statement is in a package that is run within a routine context, and the package was bound with DYNAMICRULES INVOKEBIND or DYNAMICRULES INVOKERUN. DB2 uses the statement authorization ID in effect when the routine is invoked as the value to be used for authorization checking of dynamic SQL, and for implicit qualification of unqualified object references within dynamic SQL statements within that routine. This is summarized by the following table.

Invoking Environment	ID Used
any static SQL	implicit or explicit value of the OWNER of the package the SQL invoking the routine came from
used in definition of view or trigger	definer of the view or trigger
dynamic SQL from a bind behavior package	implicit or explicit value of the OWNER of the package the SQL invoking the routine came from
dynamic SQL from a run behavior package	ID used to make the initial connection to DB2
dynamic SQL from a define behavior package	definer of the routine that uses the package that the SQL invoking the routine came from
dynamic SQL from an invoke behavior package	the current authorization ID invoking the routine

Restricted statements when run behavior does not apply

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

Considerations regarding the DYNAMICRULES option

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

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

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

authorization ID of the package owner. Similarly, when a package exhibits define behavior, the definer of the routine should not have any authorities granted that the user of the package should not receive.

Authorization IDs and statement preparation

If the VALIDATE BIND option is specified at bind time, the privileges required to manipulate tables and views must also exist at bind time. If these privileges or the referenced objects do not exist, and the SQLERROR NOPACKAGE option is in effect, the bind operation will be unsuccessful. If the SQLERROR CONTINUE option is specified, the bind operation will be successful, and any statements in error will be flagged. Any attempt to execute such a statement will result in an error.

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

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

Column names

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

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

Qualified column names

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

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

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

Where a qualifier is optional, it can serve two purposes. They are described under "Column name qualifiers to avoid ambiguity" on page 59 and "Column name qualifiers in correlated references" on page 61.

Correlation names

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

```
FROM X.MYTABLE Z
```

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

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

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

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

Example

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

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

Identifiers

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. Qualifiers for column names are also useful in SQL procedures to distinguish column names from SQL variable names used in SQL statements.

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

Table designators

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

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

Table designators in the FROM clause are established as follows:

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

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

For example, assume that the current schema is CORPDATA.

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

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

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

is also valid, because the first select list column references the unqualified exposed table designator CORPDATA.EMPLOYEE, which is in the FROM clause, and the

second select list column references the correlation name REGEMP of the table object REGION.EMPLOYEE, which is also in the FROM clause.

Now assume that the current schema is REGION.

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

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

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

Avoiding undefined or ambiguous references

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

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

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

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

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

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

is a valid statement.

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

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

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

Column name qualifiers in correlated references

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

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

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

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

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

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

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

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

Identifiers

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

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

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

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

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

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

References to variables

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

host variable

Host variables are defined by statements of a host language. For more information about how to refer to host variables, see “References to host variables” on page 63.

transition variable

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

SQL variable

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

global variable

Global variables are defined by the CREATE VARIABLE statement. For

more information about global variables, see “CREATE VARIABLE” and “References to SQL parameters, SQL variables, and global variables” in the *SQL Reference, Volume 2* .

SQL parameter

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

parameter marker

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

References to host variables

A *host variable* is either:

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

or:

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

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

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

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

The meta-variable *host-variable*, as used in the syntax diagrams, shows a reference to a host variable. A host-variable 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:

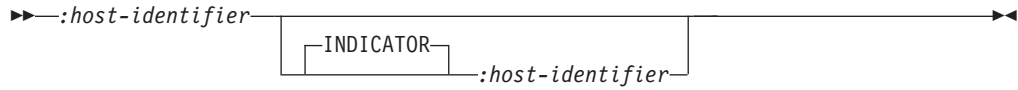
```
INSERT INTO DEPARTMENT
VALUES (:hv_deptno, :hv_deptname, :hv_mgrno, :hv_admrdept)
```

This example shows a dynamic SQL statement using parameter markers:

Identifiers

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

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



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

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

The second host-identifier designates its *indicator variable*. 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. If the database is configured with DFT_SQLMATHWARN yes (or was during binding of a static SQL statement), HV2 could be -2. If HV2 is -2, a value for HV1 could not be returned because of an error converting to the numeric type of HV1, or an error evaluating an arithmetic expression that is used to determine the value for HV1. When accessing a database with a client version earlier than DB2® Universal Database™ Version 5, HV2 will be -1 for arithmetic exceptions. If the value returned is not null, that value is assigned to HV1 and HV2 is set to zero (unless the assignment to HV1 requires string truncation of a non-LOB string; in which case HV2 is set to the original length of the string). If an assignment requires truncation of the seconds part of a time, HV2 is set to the number of seconds.

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

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

Example

Using the PROJECT table, set the host variable PNAME (VARCHAR(26)) to the project name (PROJNAME), the host variable STAFF (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”), and LOB file reference variables (see “References to BLOB, CLOB, and DBCLOB file reference variables” on page 66) can be defined in all host languages. Where LOBs are allowed, the term *host-variable* in a syntax diagram can refer to a regular host variable, a locator variable, or a file reference variable. Since these are not native data types, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable. In the case of REXX, LOBs are mapped to strings.

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

References to locator variables

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

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

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

As with all other host variables, a large object locator variable may have an associated indicator variable. Indicator variables for large object locator host

variables behave in the same way as indicator variables for other data types. When a null value is returned from the database, the indicator variable is set and the locator host variable is unchanged. This means a locator can never point to a null value.

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

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

References to 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). The direction is one of:

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

File name

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

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

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

File Name Length

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

File Options

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

- Input (from client to server)

SQL_FILE_READ

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

- Output (from server to client)

SQL_FILE_CREATE

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

SQL_FILE_OVERWRITE (Overwrite)

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

SQL_FILE_APPEND

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

Data Length

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

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

Example of an output file reference variable (in C)

Given a declare section coded as:

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

Following preprocessing this would be:

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

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

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

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

Example of an input file reference variable (in C)

Given the same declare section as 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);
```

References to structured type host variables

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

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

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

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

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

Example

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

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

Data types

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

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

DB2 supports a number of built-in data types. It also provides support for user-defined data types. Figure 11 on page 70 shows the supported built-in data types.

Data types

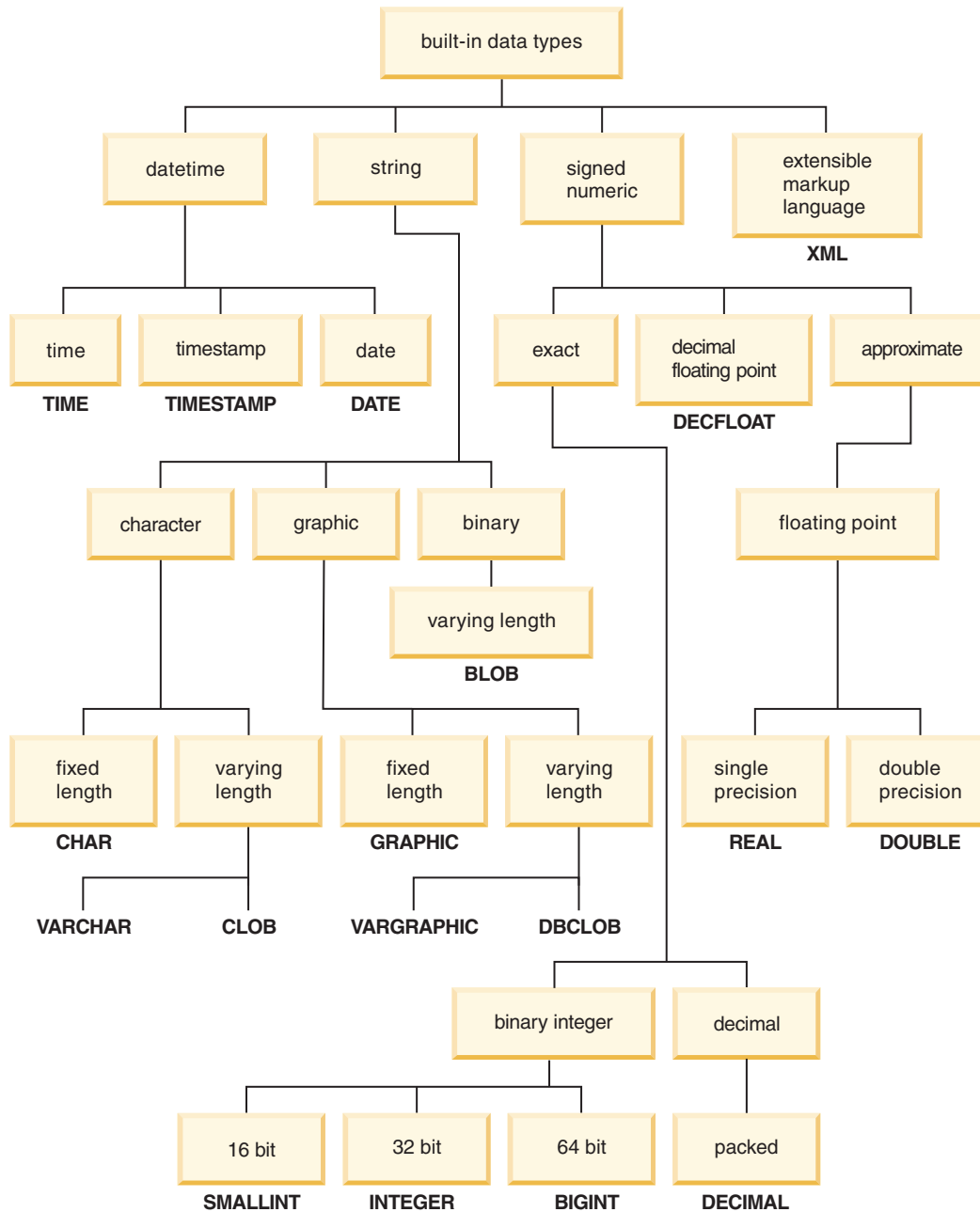


Figure 11. The DB2 Built-in Data Types

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.

Data type list

Numbers

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

The numeric data types are categorized as follows:

- Exact numerics: integer and decimal
- Decimal floating-point
- Approximate numerics: floating-point

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

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

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

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

See also the data type section in the description of the CREATE TABLE statement.

Small integer (SMALLINT)

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

Large integer (INTEGER)

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

Big integer (BIGINT)

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

Decimal (DECIMAL or NUMERIC)

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

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

Single-precision floating-point (REAL)

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

Double-precision floating-point (DOUBLE or FLOAT)

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

Decimal floating-point (DECFLOAT)

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

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

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

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

Subnormal numbers and underflow

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

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

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

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

Character strings

A *character string* is a sequence of 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 string (CHAR)

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

Varying-length character strings

There are two types of varying-length character string:

- A VARCHAR value can be up to 32 672 bytes long.
- A CLOB (character large object) value 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.

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

- A SELECT list preceded by the DISTINCT clause
- A GROUP BY clause
- An ORDER BY clause
- A subselect of a set operator other than UNION ALL
- A basic, quantified, BETWEEN, 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 explicitly 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 found in C are handled differently, depending on the standards level of the precompile option.

Each character string is further defined as one of:

Bit data

Data that is not associated with a code page.

Single-byte character set (SBCS) data

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

Mixed data

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

Character strings

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

String units in built-in functions

The ability to specify string units for certain built-in functions allows you to process string data in a more "character-based manner" than a "byte-based manner". The *string unit* determines the length in which an operation is to occur. You can specify CODEUNITS16, CODEUNITS32, or OCTETS as the string unit for an operation.

CODEUNITS16

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

CODEUNITS32

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

OCTETS

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

The calculated length of a string computed using OCTETS (bytes) might differ from that computed using CODEUNITS16 or CODEUNITS32. When using OCTETS, the length of the string is determined by simply counting the number of bytes in the string. When using CODEUNITS16 or CODEUNITS32, the length of the string is determined by counting the number of 16-bit or 32-bit code units necessary to represent the string in UTF-16 or UTF-32, respectively. The length determined using CODEUNITS16 and CODEUNITS32 will be identical unless the data contains supplementary characters (see "Difference between CODEUNITS16 and CODEUNITS32" on page 75).

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

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

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

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

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

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

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

```

Format      Representation of the name 'Jürgen'
-----
UTF-8      X'4AC3BC7267656E'
UTF-16BE   X'004A00FC007200670065006E'
UTF-32BE   X'0000004A000000FC0000007200000067000000650000006E'
    
```

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

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

Specifying string units for a built-in function does not affect the data type or the code page of the result of the function. If necessary, DB2 converts the data to Unicode for evaluation when CODEUNITS16 or CODEUNITS32 is specified.

When OCTETS is specified for the LOCATE or POSITION function, and the code pages of the string arguments differ, DB2 converts the data to the code page of the *source-string* argument. In this case, the result of the function is in the code page of the *source-string* argument. When OCTETS is specified for functions that take a single string argument, the data is evaluated in the code page of the string argument, and the result of the function is in the code page of the string argument.

Difference between CODEUNITS16 and CODEUNITS32

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

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

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

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

Character strings

Query	Returns
<code>SELECT CHARACTER_LENGTH(C1, CODEUNITS16) FROM T1</code>	2
<code>SELECT CHARACTER_LENGTH(C1, CODEUNITS32) FROM T1</code>	1
<code>SELECT CHARACTER_LENGTH(C1, OCTETS) FROM T1</code>	4

Graphic strings

A *graphic string* is a sequence of bytes that 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 checked to ensure that their values contain only double-byte character code points. (The exception to this rule is an application precompiled with the WCHARTYPE CONVERT option. In this case, validation does occur.) Rather, the database manager assumes that double-byte character data is contained in graphic data fields. The database manager *does* check that a graphic string value is an even number of bytes long.

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

Fixed-length graphic strings (GRAPHIC)

All values in a fixed-length graphic string column have the same length, which is determined by the length attribute of the column. The length attribute must be between 1 and 127, inclusive.

Varying-length graphic strings

There are two types of varying-length graphic string:

- A VARGRAPHIC value can be up to 16 336 double-byte characters long.
- A DBCLOB (double-byte character large object) value can be up to 1 073 741 823 double-byte 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 code page associated with it.

Special restrictions apply to an expression that results in a varying-length graphic string whose maximum length is greater than 127 bytes. These restrictions are the same as those specified in “Varying-length character strings” on page 73.

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

Binary strings

A *binary string* is a sequence of bytes. Unlike character strings, which usually contain text data, binary strings are used to hold non-traditional data such as pictures, voice, or mixed media. Character strings of the FOR 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 FOR BIT DATA character string to a

binary string. Binary strings are not associated with a code page. They have the same restrictions as character strings (for details, see “Varying-length character strings” on page 73).

Binary large object (BLOB)

A *binary large object* is a varying-length binary string that can be up to 2 gigabytes (2 147 483 647 bytes) long. BLOBs can hold structured data for exploitation by user-defined types and user-defined functions. Like FOR BIT DATA character strings, BLOB strings are not associated with a code page.

Large objects (LOBs)

The term *large object* and the generic acronym LOB refer to the BLOB, CLOB, or DBCLOB data type. LOB values are subject to restrictions that apply to LONG VARCHAR values, as described in “Varying-length character strings” on page 73. These restrictions apply even if the length attribute of the LOB string is 254 bytes or less.

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

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

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

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

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

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

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

Large objects (LOBs)

it is actually assigned to some location — either a user buffer in the form of a host variable, or another record in the database.

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

Datetime values

The datetime data types include DATE, TIME, and TIMESTAMP. 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. The range of the other parts is 0 to 59. If the hour is 24, the minute and second specifications are zero.

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

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

Timestamp

A *timestamp* is a seven-part value (year, month, day, hour, minute, second, and microsecond) designating 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 byte 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 user. Date, time, and timestamp values can, however, also be represented by strings. This is useful because there are no constants or variables whose data types are DATE, TIME, or TIMESTAMP. Before it can be retrieved, a datetime value must be assigned to a string variable. The CHAR function or the GRAPHIC function (for Unicode databases only) can be used to change a datetime value to a string representation. The string representation is normally the default format of datetime values associated with the territory code of the application, unless overridden by specification of the DATETIME option when the program is precompiled or bound to the database.

No matter what its length, a large object string, a LONG VARCHAR value, or a LONG VARGRAPHIC value cannot be used to represent a datetime value (SQLSTATE 42884).

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

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

Date strings

A string representation of a date is a 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 the following table. Each format is identified by name and associated abbreviation.

Table 5. 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	LOC	Depends on the territory code of the application	—

Time strings

A string representation of a time is a string that starts with a digit and has a length of at least 4 characters. Trailing blanks can be included; a leading zero can be omitted from the hour part of the time, and seconds can 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 the following table. Each format is identified by name and associated abbreviation.

Datetime values

Table 6. Formats for String Representations of Times

Format Name	Abbreviation	Time Format	Example
International Standards Organization	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	LOC	Depends on the territory code of the application	—

Note:

1. In ISO, EUR, or JIS format, .ss (or :ss) is optional.
2. The International Standards Organization changed the time format so that it is identical to the Japanese Industrial Standard Christian Era format. Therefore, use the JIS format if an application requires the current International Standards Organization format.
3. In the USA time string format, the minutes specification can be omitted, indicating an implicit specification of 00 minutes. Thus, 1 PM is equivalent to 1:00 PM.
4. In the USA time string format, the hour must not be greater than 12 and cannot be 0, except in the special case of 00:00 AM. There is a single space before 'AM' or 'PM'. 'AM' and 'PM' can be represented in lowercase or uppercase characters.

Using the JIS 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 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, but as an input value only. The ODBC string representation of a timestamp has the form *yyyy-mm-dd hh:mm:ss.nnnnnn*.

XML values

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

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

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

User-defined types

There are four types of user-defined data type:

- Distinct type
- Structured type
- Reference type
- Array type

Each of these types is described in the following sections.

Distinct type

A *distinct type* is a user-defined data type that shares its internal representation with an existing 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 TYPE AUDIO AS BLOB (1M)
```

Although AUDIO has the same representation as the built-in data type BLOB, it is considered to be a separate type; this allows the creation of functions written specifically for AUDIO, and assures that these functions will not be applied to values of any other data type (pictures, text, and so on).

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

User-defined types

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, because 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, or LOB types 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. This can be done by creating user-defined functions that are sourced on functions defined on the source type of the distinct type. The comparison operators are automatically generated for user-defined distinct types, except those using LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, or DBCLOB 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 type

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

A structured type may be used as the type of a table, view, or column. When used as a type for a table or view, that table or view is known as a *typed table* or *typed view*, respectively. For typed tables and typed views, the names and data types of the attributes of the structured type become the names and data types of the columns of this typed table or typed view. Rows of the typed table or typed view can be thought of as a representation of instances of the structured type. When used as a data type for a column, the column contains values of that structured type (or values of any of that type's subtypes, as defined below). Methods are used to retrieve or manipulate attributes of a structured column object.

Terminology: A *supertype* is a structured type for which other structured types, called *subtypes*, have been defined. A subtype inherits all the attributes and methods of its supertype and may have additional attributes and methods 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.

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

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

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

- *Indirectly uses*: A type **A** is said to indirectly use a type **B**, if one of the following is true:
 1. type **A** directly uses type **B**
 2. type **A** directly uses some type **C**, and type **C** indirectly uses type **B**

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

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

Reference type

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

The target of a reference is always a row in a typed table or a typed view. When a reference type is used, it 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*.

Array type

An *array* is a structure that contains an ordered collection of data elements in which each element can be referenced by its ordinal position in the collection. If N is the *cardinality* (number of elements) in an array, the ordinal position associated with each element is an integer value greater than or equal to 1 and less than or equal to N . All elements in an array have the same data type.

An *array type* is a data type that is defined as an array of another data type. Every array type has a maximum cardinality, which is specified on the CREATE TYPE statement. If A is an array type with maximum cardinality M , the cardinality of a value of type A can be any value between 0 and M , inclusive. Unlike the maximum cardinality of arrays in programming languages such as C, the maximum cardinality of SQL arrays is not related to their physical representation. Instead, the maximum cardinality is used by the system at run time to ensure that subscripts are within bounds. The amount of memory required to represent an array value is usually proportional to its cardinality, and not to the maximum cardinality of its type.

When an array is being referenced, all of the values in the array are stored in main memory. Therefore, arrays that contain a large amount of data will consume large amounts of main memory.

Promotion of data types

Data types can be classified into groups of related data types. Within such groups, a precedence order exists where one data type is considered to precede another data type. This precedence is used to allow the *promotion* of one data type to a data type later in the precedence ordering. For example, the data type CHAR can be promoted to VARCHAR; INTEGER can be promoted to DOUBLE-PRECISION; but CLOB is NOT promotable to VARCHAR.

Promotion of data types is used when:

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

Table 7 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 7. 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, DECFLOAT
INTEGER	INTEGER, BIGINT, decimal, real, double, DECFLOAT
BIGINT	BIGINT, decimal, real, double, DECFLOAT
decimal	decimal, real, double, DECFLOAT
real	real, double, DECFLOAT
double	double, DECFLOAT
DECFLOAT	DECFLOAT
DATE	DATE
TIME	TIME
TIMESTAMP	TIMESTAMP
udt	udt (same name) or a supertype of udt
REF(T)	REF(S) (provided that S is a supertype of T)

Table 7. Data Type Precedence Table (continued)

Data Type	Data Type Precedence List (in best-to-worst order)
Note:	
1.	The lowercase types above are defined as follows: <ul style="list-style-type: none"> • decimal = DECIMAL(p,s) or NUMERIC(p,s) • real = REAL or FLOAT(<i>n</i>), where <i>n</i> is not greater than 24 • double = DOUBLE, DOUBLE-PRECISION, FLOAT or FLOAT(<i>n</i>), where <i>n</i> is greater than 24 • udt = a user-defined type Shorter and longer form synonyms of the listed data types are considered to be the same as the listed form.
2.	For a Unicode database, the following are considered to be equivalent data types: <ul style="list-style-type: none"> • CHAR and GRAPHIC • VARCHAR and VARCHAR • LONG VARCHAR and LONG VARCHAR • CLOB and DBCLOB When resolving a function within a Unicode database, if a user-defined function and a built-in function are both applicable for a given function invocation, then generally the built-in function will be invoked. The UDF will be invoked only if its schema precedes SYSIBM in the CURRENT PATH special register and if its argument data types match all the function invocation argument data types, regardless of Unicode data type equivalence.

Casting between data types

There are many occasions where a value with a given data type needs to be *cast* to a different data type or to the same data type with a different length, precision, or scale. Data type promotion is one example where the promotion of one data type to another data type requires that the value be cast to the new data type. A data type that can be cast to another data type is *castable* from the source data type to the target data type.

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

The supported casts between built-in data types are shown in Table 8 on page 87. 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 operation. A 'Y' indicates that the CAST specification can be used for the combination of source and target data types. Cases in which only the XMLCAST specification can be used are noted.

In a Unicode database, if a truncation occurs when a character or graphic string is cast to another data type, a warning returns if any nonblank characters are truncated. This truncation behavior is unlike the assignment of character or graphic strings to a target when an error occurs if any nonblank characters are truncated.

Casting between data types

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

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

FOR BIT DATA character types cannot be cast to CLOB.

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

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

The following casts involving reference types are supported:

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

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

Table 8. Supported Casts between Built-in Data Types

Source Data Type	Target Data Type																			
	S M A L L I N T	I N T E G E R	B I G I N T	D E C I M A L	D E C I M A L	D E C I M A L	D E C I M A L	D E C I M A L	D E C I M A L	D E C I M A L	D E C I M A L	D E C I M A L	D E C I M A L	D E C I M A L	D E C I M A L	D E C I M A L	D E C I M A L	D E C I M A L	D E C I M A L	
SMALLINT	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	Y ³
INTEGER	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	Y ³
BIGINT	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	Y ³
DECIMAL	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	Y ³
REAL	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	Y ³
DOUBLE	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	Y ³
DECFLOAT	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
CHAR	Y	Y	Y	Y	-	-	Y	Y	Y	Y	Y	Y	Y ¹	Y ¹	Y ¹	Y	Y	Y	Y	Y ⁴
CHAR FOR BIT DATA	Y	Y	Y	Y	-	-	Y	Y	Y	Y	Y	-	-	-	-	Y	Y	Y	Y	Y ³
VARCHAR	Y	Y	Y	Y	-	-	Y	Y	Y	Y	Y	Y	Y ¹	Y ¹	Y ¹	Y	Y	Y	Y	Y ⁴
VARCHAR FOR BIT DATA	Y	Y	Y	Y	-	-	Y	Y	Y	Y	Y	-	-	-	-	Y	Y	Y	Y	Y ³
CLOB	-	-	-	-	-	-	-	Y	-	Y	-	Y	Y ¹	Y ¹	Y ¹	Y	-	-	-	Y ⁴
GRAPHIC	Y ¹	Y ¹	Y ¹	Y ¹	-	-	Y ¹	Y ¹	-	Y ¹	-	Y ¹	Y	Y	Y	Y	Y ¹	Y ¹	Y ¹	Y ³
VARGRAPHIC	Y ¹	Y ¹	Y ¹	Y ¹	-	-	Y ¹	Y ¹	-	Y ¹	-	Y ¹	Y	Y	Y	Y	Y ¹	Y ¹	Y ¹	Y ³
DBCLOB	-	-	-	-	-	-	-	Y ¹	-	Y ¹	-	Y ¹	Y	Y	Y	Y	-	-	-	Y ³
BLOB	-	-	-	-	-	-	-	-	-	Y	Y	-	-	-	-	Y	-	-	-	Y ⁴
DATE	-	Y	Y	Y	-	-	-	Y	Y	Y	Y	-	Y ¹	Y ¹	-	-	Y	-	-	Y ³
TIME	-	Y	Y	Y	-	-	-	Y	Y	Y	Y	-	Y ¹	Y ¹	-	-	-	Y	-	Y ³
TIMESTAMP	-	-	Y	Y	-	-	-	Y	Y	Y	Y	-	Y ¹	Y ¹	-	-	Y	Y	Y	Y ³
XML	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y ⁵	Y

Casting between data types

Table 8. Supported Casts between Built-in Data Types (continued)

Source Data Type	Target Data Type															
	S	M	A	L	I	G	N	T	D	E	C	R	D ²	V	H	B
SMALLINT																
INTEGER																
BIGINT																
DECIMAL																
NUMERIC																
REAL																
DOUBLE																
DECFLOAT																

Notes

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

¹ Cast is only supported for Unicode databases.

² FOR BIT DATA

³ Cast can only be performed using XMLCAST.

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

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

Table Table 9 shows where to find information about the rules that apply when casting to the identified target data types.

Table 9. Rules for Casting to a Data Type

Target Data Type	Rules
SMALLINT	“SMALLINT scalar function” in <i>SQL Reference, Volume 1</i>
INTEGER	“INTEGER scalar function” in <i>SQL Reference, Volume 1</i>
BIGINT	“BIGINT scalar function” in <i>SQL Reference, Volume 1</i>
DECIMAL	“DECIMAL scalar function” in <i>SQL Reference, Volume 1</i>
NUMERIC	“NUMERIC scalar function” in <i>SQL Reference, Volume 1</i>
REAL	“REAL scalar function” in <i>SQL Reference, Volume 1</i>
DOUBLE	“DOUBLE scalar function” in <i>SQL Reference, Volume 1</i>
DECFLOAT	“DECFLOAT scalar function” in <i>SQL Reference, Volume 1</i>

Table 9. Rules for Casting to a Data Type (continued)

Target Data Type	Rules
CHAR	“CHAR scalar function” in <i>SQL Reference, Volume 1</i>
VARCHAR	“VARCHAR scalar function” in <i>SQL Reference, Volume 1</i>
CLOB	“CLOB scalar function” in <i>SQL Reference, Volume 1</i>
GRAPHIC	“GRAPHIC scalar function” in <i>SQL Reference, Volume 1</i>
VARGRAPHIC	“VARGRAPHIC scalar function” in <i>SQL Reference, Volume 1</i>
DBCLOB	“DBCLOB scalar function” in <i>SQL Reference, Volume 1</i>
BLOB	“BLOB scalar function” in <i>SQL Reference, Volume 1</i>
DATE	“DATE scalar function” in <i>SQL Reference, Volume 1</i>
TIME	“TIME scalar function” in <i>SQL Reference, Volume 1</i>
TIMESTAMP	If the source type is a character string, see “TIMESTAMP scalar function” in <i>SQL Reference, Volume 1</i> , where one operand is specified. If the source data type is a DATE, the timestamp is composed of the specified date and a time of 00:00:00. If the source data type is a TIME, the timestamp is composed of the CURRENT DATE and the specified time.

Casting non-XML values to XML values

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

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

Casting between data types

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

Source Data Type	Target Data Type	
	XML	Resulting XML Schema Type
DATE	Y	xs:date
TIME	Y	xs:time
TIMESTAMP	Y	xs:dateTime
BLOB	Y	xs:base64Binary
character type FOR BIT DATA	Y	xs:base64Binary
distinct type		use this chart with the source type of the distinct type

Note:

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

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

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

Casting XML values to non-XML values

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

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

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

Table 11. Supported Casts from XML Values to Non-XML Values

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

Table 11. Supported Casts from XML Values to Non-XML Values (continued)

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

Notes

¹ DB2 supports XML Schema 1.0, which does not provide a matching XML schema type for a DECFLOAT. Processing of the XQuery cast step of XMLCAST is handled as follows:

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

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

- XML values that are to be converted to string types must fit within the length limits of those DB2 types without truncation of any characters or bytes. The name used for the derived XML schema type is the uppercase SQL type name followed by an underscore character and the maximum length of the string; for example, VARCHAR_20 if the XMLCAST target data type is VARCHAR(20).
- XML values that are to be converted to DECIMAL values must fit within the precision of the specified DECIMAL values, and must not contain more non-zero digits after the decimal point than the scale. The name used for the derived XML schema type is DECIMAL_precision_scale, where *precision* is the precision of the target SQL data type, and *scale* is the scale of the target SQL data type; for example, DECIMAL_9_2 if the XMLCAST target data type is DECIMAL(9,2).
- XML values that are to be converted to TIME values cannot contain a seconds component with non-zero digits after the decimal point. The name used for the derived XML schema type is TIME.

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

Casting between data types

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

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

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

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

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

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

Assignments and comparisons

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

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

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

Assignments and comparisons involving both character and graphic data are only supported when one of the strings is a literal.

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

Table 12. Data Type Compatibility for Assignments and Comparisons

Operands	Binary Integer	Decimal Number	Floating-point	Decimal Floating-point	Character String	Graphic String	Date	Time	Time-stamp	Binary String	UDT
Binary Integer	Yes	Yes	Yes	Yes	No	No	No	No	No	No	²
Decimal Number	Yes	Yes	Yes	Yes	No	No	No	No	No	No	²
Floating-point	Yes	Yes	Yes	Yes	No	No	No	No	No	No	²
Decimal Floating-point	Yes	Yes	Yes	Yes	No	No	No	No	No	No	²
Character String	No	No	No	No	Yes	Yes ^{5,6}	¹	¹	¹	No ³	²
Graphic String	No	No	No	No	Yes ^{5,6}	Yes	¹	¹	¹	No	²
Date	No	No	No	No	¹	¹	Yes	No	No	No	²
Time	No	No	No	No	¹	¹	No	Yes	No	No	²
Timestamp	No	No	No	No	¹	¹	No	No	Yes	No	²
Binary String	No	No	No	No	No ³	No	No	No	No	Yes	²
UDT	²	²	²	²	²	²	²	²	²	²	Yes

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

- Datetime values can be assigned to string columns and to string variables.
- 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.

(Graphic string support is only available for Unicode databases.)

² A user-defined distinct type value is only comparable to a value defined with the same user-defined distinct type. In general, assignments are supported between a distinct type value and its source data type. A user-defined structured type is not comparable and can only be assigned to an operand of the same structured type or one of its supertypes. For additional information see “User-defined type assignments” on page 99.

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

⁴ For information on assignment and comparison of reference types, see “Reference type assignments” on page 100 and “Reference type comparisons” on page 105.

⁵ Only supported for Unicode databases.

⁶ Bit data and graphic strings are not compatible.

Numeric assignments

For numeric assignments, overflow is not allowed.

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

For decimal floating-point, truncation of the whole part of the number is not allowed and results in an error.

Assignments and comparisons

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

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

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

Assignments to integer

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

Assignments to decimal

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

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

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

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

Assignments to floating-point

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

variable using floating-point arithmetic. A decimal floating-point value is first converted to a string representation, and is then converted to a floating-point number.

Assignments to decimal floating-point

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

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

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

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

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

String assignments

There are two types of assignments:

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

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

Storage assignment

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

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

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

Assignments and comparisons

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

Retrieval assignment

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

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

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

Conversion rules for string assignments

A character string or graphic string assigned to a column 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).

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

MBCS considerations for character string assignments

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

- Blank padding is always done using the single-byte blank character (X'20').
- Blank truncation is always done based on the single-byte blank character (X'20'). The double-byte blank character is treated like any other character with respect to truncation.
- Assignment of a character string to a host variable 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. For non-Unicode databases, graphic string data types are compatible only with other graphic string data types, and never with numeric, character string, or datetime data types. For Unicode databases, graphic string data types are compatible with character string data types. However, graphic and character string data types cannot be used interchangeably in the SELECT INTO or the VALUES INTO statement.

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

If a graphic string value (the 'source' string) is assigned to a fixed length graphic string data type (the 'target', which can be a column 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 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.

Datetime assignments

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

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

Assignments and comparisons

When the target is a string variable that is not a host variable, the following rules apply:

- **For a DATE:** If the variable length is less than 10 characters, an error is returned.
- **For a TIME:** If the variable length is less than 8 characters, an error is returned.
- **For a TIMESTAMP:** If the host variable is less than 26 characters, an error is returned.

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

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

If ISO or JIS formats are used, and if the length of the host variable is less than 8 characters, the seconds part of the time is omitted from the result and assigned to the indicator variable, if provided. The SQLWARN1 field of the SQLCA is set to indicate the omission.

- **For a TIMESTAMP:** If the host variable is less than 19 characters, an error is returned. If the length is less than 26 characters, but greater than or equal to 19 characters, trailing digits of the microseconds part of the value are omitted. The SQLWARN1 field of the SQLCA is set to indicate the omission.

XML assignments

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

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

.NET), XML values can be retrieved into the character, graphic, or binary string types that are supported by the application programming interface. In all of these cases, the XML value is implicitly serialized to a string encoded in UTF-8 and, for character or graphic string variables, converted into the client code page.

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

User-defined type assignments

For distinct types and structured types, different rules are applied for assignments to host variables than are used for all other assignments.

Distinct Types: 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 based on a distinct type, the source data type must be castable to the target data type.

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

- A value of a structured 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 declared type of the host variable is the structured type or a supertype of the structured type.

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

For array types, different rules are applied for assignments to SQL variables and parameters. The validity of an assignment to an SQL variable or parameter is determined according to the following rules:

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

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

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

is equivalent to:

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

And the statement:

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

Assignments and comparisons

is equivalent to:

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

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.

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

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

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

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

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

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

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

The ordering among different special values is as follows: -NAN < -SNAN < -INFINITY < 0 < INFINITY < SNAN < NAN

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 with blanks sufficient to extend its length to that of the longer string. This logical extension is done for all character strings, including those tagged as FOR BIT DATA.

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

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

LOB strings 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.

Portions of 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      VARCHAR(8000)
```

then the following is valid:

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

Examples:

Assignments and comparisons

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

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

'a' < 'A' < 'á' < 'Á'

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

D4 < D3 < D2 < D1

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

'A' < 'a' < 'Á' < 'á'

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

D1 < D2 < D3 < D4

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

'A' < 'a' < 'Á' < 'á'

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

D1 < D2 < D3 < D4

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

D1 < D2 < D3 < D4

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

Conversion rules for comparison

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

Ordering of results

Results that require sorting are ordered based on the string comparison rules discussed in "String comparisons" on page 101. 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 DBCLOB.

For graphic strings, the collating sequence of the database is not used. Instead, graphic strings are always compared based on the numeric (binary) values of their corresponding bytes.

Using the previous example, if the literals were graphic strings, then:

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

and

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

When comparing graphic strings of unequal lengths, the comparison is made using a logical copy of the shorter string which is padded on the right with double-byte blank characters sufficient to extend its length to that of the longer string.

Assignments and comparisons

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'
```

User-defined type comparisons

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

Example:

Given the following YOUTH distinct type and CAMP_DB2_ROSTER table:

```
CREATE 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)
```

Values with a user-defined structured type cannot be compared with any other value (the NULL predicate and the TYPE predicate can be used).

Reference type comparisons

Reference type values can be compared only if their target types have a common supertype. The appropriate comparison function will only be found if the schema name of the common supertype is included in the SQL path. The comparison is performed using the representation type of the reference types. The scope of the reference is not considered in the comparison.

XML comparisons in a non-Unicode database

When performed in a non-Unicode database, comparisons between XML data and character or graphic string values require a code page conversion of one of the two sets of data being compared. Character or graphic values used in an SQL or XQuery statement, either as a query predicate or as a host variable with a character or graphic string data type, are converted to the database code page prior to comparison. If any characters included in this data have code points that are not part of the database code page, substitution characters are added in their place, potentially causing unexpected results for the query.

For example, a client with a UTF-8 code page is used to connect to a database server created with the Greek encoding ISO8859-7. The expression $\Sigma_G \Sigma_M$ is sent as the predicate of an XQuery statement, where Σ_G represents the Greek sigma character in Unicode (U+03A3) and Σ_M represents the mathematical symbol sigma in Unicode (U+2211). This expression is first converted to the database code page, so that both "Σ" characters are converted to the equivalent code point for sigma in the Greek database code page, 0xD3. We may denote this code point as Σ_A . The newly converted expression $\Sigma_A \Sigma_A$ is then converted again to UTF-8 for comparison with the target XML data. Since the distinction between these two code points was lost as a result of the code page conversion required to pass the predicate expression into the database, the two initially distinct values Σ_G and Σ_M are passed to the XML parser as the expression $\Sigma_G \Sigma_G$. This expression then fails to match when compared to the value $\Sigma_G \Sigma_M$ in an XML document.

One way to avoid the unexpected query results that may be caused by code page conversion issues is to ensure that all characters used in a query expression have matching code points in the database code page. Characters that do not have matching code points can be included through the use of a Unicode character entity reference. A character entity reference will always bypass code page conversion. For example, using the character entity reference `ࢣ` in place of the Σ_M character ensures that the correct Unicode code point is used for the comparison, regardless of the database code page.

Rules for result data types

The data types of a result are determined by rules which are applied to the operands in an operation. This 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 and the DECODE scalar function
- Arguments of the scalar function COALESCE (also NVL and VALUE)
- Arguments of the scalar functions GREATEST, LEAST, MAX, and MIN
- 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 LOB strings for the various operations.

The rules involving various data types follow. In some cases, a table is used to show the possible result data types. The LONG VARCHAR and LONG VARGRAPHIC data types continue to be supported but are deprecated and not recommended.

These tables identify the data type of the result, including the applicable length or precision and scale. The result type is determined by considering the operands. If there is more than one pair of operands, start by considering the first pair. This gives a result type which is considered with the next operand to determine the next result type, and so on. The last intermediate result type and the last operand determine the result type for the operation. Processing of operations is done from left to right so that the intermediate result types are important when operations are repeated. For example, consider a situation involving:

```
CHAR(2) UNION CHAR(4) UNION VARCHAR(3)
```

The first pair results in a type of CHAR(4). The result values always have 4 bytes. The final result type is VARCHAR(4). Values in the result from the first UNION operation will always have a length of 4.

Character strings

A character string value is compatible with another character string value. Character strings include data types CHAR, VARCHAR, and CLOB.

If one operand is...	And the other operand is...	The data type of the result is...
CHAR(x)	CHAR(y)	CHAR(z) where $z = \max(x,y)$
CHAR(x)	VARCHAR(y)	VARCHAR(z) where $z = \max(x,y)$
VARCHAR(x)	CHAR(y) or VARCHAR(y)	VARCHAR(z) where $z = \max(x,y)$
CLOB(x)	CHAR(y), VARCHAR(y), or CLOB(y)	CLOB(z) where $z = \max(x,y)$

The code page of the result character string will be derived based on the rules for string conversions.

Graphic strings

A graphic string value is compatible with another graphic string value. Graphic strings include data types GRAPHIC, VARGRAPHIC, and DBCLOB.

If one operand is...	And the other operand is...	The data type of the result is...
GRAPHIC(x)	GRAPHIC(y)	GRAPHIC(z) where $z = \max(x,y)$
VARGRAPHIC(x)	GRAPHIC(y) OR VARGRAPHIC(y)	VARGRAPHIC(z) where $z = \max(x,y)$
DBCLOB(x)	GRAPHIC(y), VARGRAPHIC(y), or DBCLOB(y)	DBCLOB(z) where $z = \max(x,y)$

The code page of the result graphic string will be derived based on the rules for string conversions.

Character and graphic strings in a Unicode database

In a Unicode database, a character string value is compatible with a graphic string value.

If one operand is...	And the other operand is...	The data type of the result is...
GRAPHIC(x)	CHAR(y) or GRAPHIC(y)	GRAPHIC(z) where $z = \max(x,y)$
VARGRAPHIC(x)	CHAR(y) or VARCHAR(y)	VARGRAPHIC(z) where $z = \max(x,y)$
VARCHAR(x)	GRAPHIC(y) or VARGRAPHIC	VARGRAPHIC(z) where $z = \max(x,y)$
DBCLOB(x)	CHAR(y) or VARCHAR(y) or CLOB(y)	DBCLOB(z) where $z = \max(x,y)$
CLOB(x)	GRAPHIC(y) or VARGRAPHIC(y)	DBCLOB(z) where $z = \max(x,y)$

Binary large object (BLOB)

A binary string (BLOB) value is compatible only with another binary string (BLOB) value. The BLOB scalar function can be used to cast from other types if they should be treated as BLOB types. The length of the result BLOB is the largest length of all the data types.

Numeric

Numeric types are compatible with other numeric types, character-string data types, and in a Unicode database, graphic-string data types. Numeric types include SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, DOUBLE, and DECFLOAT.

If one operand is...	And the other operand is...	The data type of the result is...
SMALLINT	SMALLINT	SMALLINT

Rules for result data types

If one operand is...	And the other operand is...	The data type of the result is...
INTEGER	SMALLINT	INTEGER
INTEGER	INTEGER	INTEGER
BIGINT	SMALLINT	BIGINT
BIGINT	INTEGER	BIGINT
BIGINT	BIGINT	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 $p = \max(x,z) + \max(w-x, y-z)^1$ $s = \max(x,z)$
REAL	REAL	REAL
REAL	SMALLINT, INTEGER, BIGINT, or DECIMAL	DOUBLE
DOUBLE	SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, or DOUBLE	DOUBLE
DECFLOAT(n)	SMALLINT, INTEGER, DECIMAL (<=16,s), REAL, or DOUBLE	DECFLOAT(n)
DECFLOAT(n)	BIGINT	DECFLOAT(34)
DECFLOAT(n)	DECIMAL (>16,s)	DECFLOAT(34)
DECFLOAT(n)	DECFLOAT(m)	DECFLOAT(MAX(n,m))

¹ Precision cannot exceed 31.

DATE

A date value is compatible with another date value, 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 value is compatible with another time value, 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 value is compatible with another timestamp value, or any CHAR or VARCHAR expression that contains a valid string representation of a timestamp.

The data type of the result is `TIMESTAMP`.

XML

An XML value is compatible with another XML value. The data type of the result is XML.

Distinct types

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

Reference types

A reference type value is compatible with another value of the same reference type provided that their target types have a common supertype. The data type of the result is a reference type having the common supertype as the target type. If all operands have the identical scope table, the result has that scope table. Otherwise the result is unscoped.

Structured types

A structured type value is compatible with another value of the same structured type provided that they have a common supertype. The static data type of the resulting structured type column is the structured type that is the least common supertype of either column.

For example, consider the following structured type hierarchy,



Structured types of the static type E and F are compatible with the resulting static type of B, which is the least common super type of E and F.

Nullable attribute of result

With the exception of `INTERSECT` and `EXCEPT`, the result allows nulls unless both operands do not allow nulls.

- For `INTERSECT`, if either operand does not allow nulls the result does not allow nulls (the intersection would never be null).
- For `EXCEPT`, if the first operand does not allow nulls the result does not allow nulls (the result can only be values from the first operand).

Rules for string conversions

The code page used to perform an operation is determined by rules which are applied to the operands in that operation. This section explains those rules.

These rules apply to:

Rules for string conversions

- Corresponding string columns in fullselects with set operations (UNION, INTERSECT and EXCEPT)
- Operands of concatenation
- Operands of predicates (with the exception of LIKE)
- Result expressions of a CASE expression and the DECODE scalar function
- Arguments of the scalar function COALESCE (also NVL and VALUE)
- Arguments of the scalar functions GREATEST, LEAST, MAX, and MIN
- The *source-string* and *insert-string* arguments of the scalar function OVERLAY (and INSERT)
- Expression values of the in list of an IN predicate
- Corresponding expressions of a multiple row VALUES clause.

In each case, the code page of the result is determined at bind time, and the execution of the operation may involve conversion of strings to the code page identified by that code page. A character that has no valid conversion is mapped to the substitution character for the character set and SQLWARN10 is set to 'W' in the SQLCA.

The code page of the result is determined by the code pages of the operands. The code pages of the first two operands determine an intermediate result code page, this code page and the code page of the next operand determine a new intermediate result code page (if applicable), and so on. The last intermediate result code page and the code page of the last operand determine the code page of the result string or column. For each pair of code pages, the result is determined by the sequential application of the following rules:

- If the code pages are equal, the result is that code page.
- If either code page is BIT DATA (code page 0), the result code page is BIT DATA.
- In a Unicode database, if one code page denotes data in an encoding scheme that is different from the other code page, the result is UCS-2 over UTF-8 (that is, the graphic data type over the character data type). (In a non-Unicode database, conversion between different encoding schemes is not supported.)
- For operands that are host variables (whose code page is not BIT DATA), the result code page is the database code page. Input data from such host variables is converted from the application code page to the database code page before being used.

Conversions to the code page of the result are performed, if necessary, for:

- An operand of the concatenation operator
- The selected argument of the COALESCE (also NVL and VALUE) scalar function
- The selected argument of the scalar functions GREATEST, LEAST, MAX, and MIN
- The *source-string* and *insert-string* arguments of the scalar function OVERLAY (and INSERT)
- The selected result expression of the CASE expression and the DECODE scalar function
- The expressions of the in list of the IN predicate
- The corresponding expressions of a multiple row VALUES clause
- The corresponding columns involved in set operations.

Character conversion is necessary if all of the following are true:

- The code pages are different
- Neither string is BIT DATA
- The string is neither null nor empty

Examples

Example 1: Given the following in a database created with code page 850:

Expression	Type	Code Page
COL_1	column	850
HV_2	host variable	437

When evaluating the predicate:

```
COL_1 CONCAT :HV_2
```

the result code page of the two operands is 850, because the host variable data will be converted to the database code page before being used.

Example 2: Using information from the previous example when evaluating the predicate:

```
COALESCE(COL_1, :HV_2:NULLIND,)
```

the result code page is 850; therefore, the result code page for the COALESCE scalar function will be code page 850.

String comparisons in a Unicode database

Pattern matching is one area where the behavior of existing MBCS databases is slightly different from the behavior of a Unicode database.

For MBCS databases in DB2 Database for Linux, UNIX, and Windows, the current behavior is as follows: If the match-expression contains MBCS data, the pattern can include both SBCS and non-SBCS characters. The special characters in the pattern are interpreted as follows:

- An SBCS halfwidth underscore refers to one SBCS character.
- A non-SBCS fullwidth underscore refers to one non-SBCS character.
- A percent (either SBCS halfwidth or non-SBCS fullwidth) refers to zero or more SBCS or non-SBCS characters.

In a Unicode database, there is really no distinction between "single-byte" and "non-single-byte" characters. Although the UTF-8 format is a "mixed-byte" encoding of Unicode characters, there is no real distinction between SBCS and non-SBCS characters in UTF-8. Every character is a Unicode character, regardless of the number of bytes in UTF-8 format. In a Unicode graphic column, every non-supplementary character, including the halfwidth underscore (U+005F) and halfwidth percent (U+0025), is two bytes in width. For Unicode databases, the special characters in the pattern are interpreted as follows:

- For character strings, a halfwidth underscore (X'5F') or a fullwidth underscore (X'EFBCBF') refers to one Unicode character. A halfwidth percent (X'25') or a fullwidth percent (X'EFBC85') refers to zero or more Unicode characters.
- For graphic strings, a halfwidth underscore (U+005F) or a fullwidth underscore (U+FF3F) refers to one Unicode character. A halfwidth percent (U+0025) or a fullwidth percent (U+FF05) refers to zero or more Unicode characters.

String comparisons in a Unicode database

Note: You need two underscores to match a Unicode supplementary graphic character because such a character is represented by two UCS-2 characters in a GRAPHIC column. Only one underscore is needed to match a Unicode supplementary character in a CHAR column.

For the optional "escape expression", which specifies a character to be used to modify the special meaning of the underscore and percent sign characters, the expression can be specified by any one of:

- A constant
- A special register
- A host variable
- A scalar function whose operands are any of the 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.
- For CHAR columns, the result of the expression must be one character or a binary string containing exactly one (1) byte (SQLSTATE 22019). For GRAPHIC columns, the result of the expression must be one character (SQLSTATE 22019).

Database partition-compatible data types

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

Table 13 on page 113 shows the compatibility of data types in database partitions.

Database partition compatibility has the following characteristics:

- Internal formats are used for DATE, TIME, and TIMESTAMP. They are not compatible with each other, and none are compatible with character or graphic data types.
- Partition compatibility is not affected by the nullability of a column.
- Partition compatibility is affected by collation. Locale-sensitive UCA-based collations require an exact match in collation, except that the strength (S) attribute of the collation is ignored. All other collations are considered equivalent for the purposes of determining partition compatibility.
- Character columns defined with FOR BIT DATA are only compatible with character columns without FOR BIT DATA when a collation other than a locale-sensitive UCA-based collation is used.
- NULL values of compatible data types are treated identically. Different results might be produced for NULL values of non-compatible data types.
- Base data type of the UDT is used to analyze database partition compatibility.
- Decimals of the same value in the distribution key are treated identically, even if their scale and precision differ.
- Trailing blanks in character strings (CHAR, VARCHAR, GRAPHIC or VARGRAPHIC) are ignored by the system-provided hashing function.

- When a locale-sensitive UCA-based collation is used, CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are compatible data types. When other collations are used, CHAR and VARCHAR are compatible types and GRAPHIC and VARGRAPHIC are compatible types, but CHAR and VARCHAR are not compatible types with GRAPHIC and VARGRAPHIC. CHAR or VARCHAR of different lengths are compatible data types.
- DECFLOAT values that are equal are treated identically even if their precision differs. DECFLOAT values that are numerically equal are treated identically even if they have a different number of significant digits.

Table 13. Database Partition Compatibilities

Operands	Binary Integer	Decimal Number	Floating-point	Decimal Floating-point	Character String	Graphic String	Date	Time	Time-stamp	Distinct Type	Structured Type
Binary Integer	Yes	No	No	No	No	No	No	No	No	¹	No
Decimal Number	No	Yes	No	No	No	No	No	No	No	¹	No
Floating-point	No	No	Yes	No	No	No	No	No	No	¹	No
Decimal Floating-point	No	No	No	Yes	No	No	No	No	No	¹	No
Character String ⁴	No	No	No	No	Yes ²	2, 3	No	No	No	¹	No
Graphic String ⁴	No	No	No	No	2, 3	Yes ²	No	No	No	¹	No
Date	No	No	No	No	No	No	Yes	No	No	¹	No
Time	No	No	No	No	No	No	No	Yes	No	¹	No
Timestamp	No	No	No	No	No	No	No	No	Yes	¹	No
Distinct Type	¹	¹	¹	¹	¹	¹	¹	¹	¹	¹	No
Structured Type ⁴	No	No	No	No	No	No	No	No	No	No	No

Note:

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

² Character and graphic string types are compatible when they have compatible collations.

³ Character and graphic string types are compatible when a locale-sensitive UCA-based collation is in effect. Otherwise, they are not compatible types.

⁴ User-defined structured types and data types LONG VARCHAR, LONG VARGRAPHIC, CLOB, DBCLOB, and BLOB are not applicable for database partition compatibility, because they are not supported in distribution 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 NOT NULL attribute.

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

Constants

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 if the constant has been cast to the user-defined type. For example, using the table and distinct type in "User-defined type comparisons" on page 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
```

Integer constants

An *integer constant* specifies an integer as a signed or unsigned number with a maximum of 19 digits that does not include a decimal point. The data type of an integer constant is large integer if its value is within the range of a large integer. The data type of an integer constant is big integer if its value is outside the range of large integer but within the range of a big integer. A constant that is defined outside the range of big integer values is considered a decimal constant.

Note that the smallest literal representation of a large integer constant is -2 147 483 647, and not -2 147 483 648, which is the limit for integer values. Similarly, the smallest literal representation of a big integer constant is -9 223 372 036 854 775 807, and not -9 223 372 036 854 775 808, which is the limit for big integer values.

Examples:

```
64      -15      +100     32767     720176     12345678901
```

In syntax diagrams, the term 'integer' is used for a large integer constant that must not include a sign.

Floating-point constants

A *floating-point constant* specifies a floating-point number as two numbers separated by an E. The first number may include a sign and a decimal point; the second number may include a sign but not a decimal point. The data type of a floating-point constant is double-precision. The value of the constant is the product of the first number and the power of 10 specified by the second number; it must be within the range of floating-point numbers. The number of bytes in the constant must not exceed 30.

Examples:

```
15E1     2.E5     2.2E-1   +5.E+2
```

Decimal constants

A *decimal constant* is a signed or unsigned number that consists of no more than 31 digits and either includes a decimal point or is not within the range of binary integers. It must be within the range of decimal numbers. The precision is the total

number of digits (including leading and trailing zeros); the scale is the number of digits to the right of the decimal point (including trailing zeros).

Examples:

```
25.5      1000.      -15.      +37589.3333333333
```

Decimal floating-point constants

There are no decimal floating-point constants except for the decimal floating-point special values, which are interpreted as `DECFLOAT(34)`.

These special values are: `INFINITY`, `NAN`, and `SNAN`. `INFINITY` represents infinity, a number whose magnitude is infinitely large. `INFINITY` can be preceded by an optional sign. `INF` can be specified in place of `INFINITY`. `NAN` represents Not a Number (NaN) and is sometimes called quiet NaN. It is a value that represents undefined results which does not cause a warning or exception. `SNAN` represents signaling NaN (sNaN). It is a value that represents undefined results which will cause a warning or exception if used in any operation that is defined in any numerical operation. Both `NAN` and `SNAN` can be preceded by an optional sign, but the sign is not significant. `SNAN` can be used in non-numerical operations without causing a warning or exception, for example in the `VALUES` list of an `INSERT` or as a constant compared in a predicate.

```
SNAN      -INFINITY
```

All non-special values are interpreted as integer, floating-point or decimal constants, in accordance with the rules specified above. To obtain a numeric decimal floating-point value, use the `DECFLOAT` cast function with a character string constant. It is not recommended to use floating-point constants as arguments to the `DECFLOAT` function, because floating-point is not exact and the resulting decimal floating-point value might be different than the decimal digit characters that make up the argument. Instead, use character constants as arguments to the `DECFLOAT` function.

For example, `DECFLOAT('6.0221415E23', 34)` returns the decimal floating-point value `6.0221415E+23`, but `DECFLOAT(6.0221415E23, 34)` returns the decimal floating-point value `6.0221415000000003E+23`.

Character string constants

A *character string constant* specifies a varying-length character string. There are three forms of a character string constant:

- A sequence of characters that starts and ends with a string delimiter, which is an apostrophe (`'`). The number of bytes between the string delimiters cannot be greater than 32 672. Two consecutive string delimiters are used to represent one string delimiter within the character string. Two consecutive string delimiters that are not contained within a string represent the empty string.
- `X` followed by a sequence of characters that starts and ends with a string delimiter. This form of a character string constant is also called a *hexadecimal constant*. The characters between the string delimiters must be an even number of hexadecimal digits. Blanks between the string delimiters are ignored. The number of hexadecimal digits must not exceed 32 672. A hexadecimal digit is a digit or any of the letters `A` through `F` (uppercase or lowercase). Under the conventions of hexadecimal notation, each pair of hexadecimal digits represents a character. This form of a character string constant allows you to specify characters that do not have a keyboard representation.

Constants

- U& followed by a sequence of characters that starts and ends with a string delimiter and that is optionally followed by the UESCAPE clause. This form of a character string constant is also called a *Unicode string constant*. The number of bytes between the string delimiters cannot be greater than 32 672. The Unicode string constant is converted from UTF-8 to the section code page during statement compilation. Two consecutive string delimiters are used to represent one string delimiter within the character string. Two consecutive Unicode escape characters are used to represent one Unicode escape character within the character string, but these characters count as one character when calculating the lengths of character constants. Two consecutive string delimiters that are not contained within a string represent the empty string. Because a character in UTF-8 can range from 1 to 4 bytes, a Unicode string constant of the maximum length might actually represent fewer than 32 672 characters.

A character can be expressed by either its typographical character (*glyph*) or its Unicode code point. The code point of a Unicode character ranges from X'000000' to X'10FFFF'. To express a Unicode character through its code point, use the Unicode escape character followed by 4 hexadecimal digits, or the Unicode escape character followed by a plus sign (+) and 6 hexadecimal digits. The default Unicode escape character is the reverse solidus (\), but a different character can be specified with the UESCAPE clause. The UESCAPE clause is specified as the UESCAPE keyword followed by a single character between string delimiters. The Unicode escape character cannot be a plus sign (+), a double quotation mark ("), a single quotation mark ('), a blank, or any of the characters 0 through 9 or A through F, in either uppercase or lowercase (SQLSTATE 42604). An example of the two ways in which the Latin capital letter A can be specified as a Unicode code point is \0041 and \+000041.

The constant value is always converted to the database code page when it is bound to the database. It is considered to be in the database code page. Therefore, if used in an expression that combines a constant with a FOR BIT DATA column, and whose result is FOR BIT DATA, the constant value will not be converted from its database code page representation when used.

Examples:

```
'12/14/1985' '32' 'DON''T CHANGE' ''  
X'FFFF' X'46 72 61 6E 6B'  
U&'\01416d\017A is a city in Poland' U&'c:\\temp' U&'@+01D11E' UESCAPE '@'
```

The rightmost string on the second line in the example represents the VARCHAR pattern of the ASCII string 'Frank'. The last line corresponds to: '01416d017A is a city in Poland', 'c:\\temp', and a single character representing the musical symbol G clef.

Graphic string constants

A *graphic string constant* specifies a varying-length graphic string consisting of a sequence of double-byte characters that starts and ends with a single-byte apostrophe ('), and that is preceded by a single-byte G or N. The characters between the apostrophes must represent an even number of bytes, and the length of the graphic string must not exceed 16 336 bytes.

Examples:

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

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

In a Unicode database, a hexadecimal graphic string constant that specifies a varying-length graphic string is also supported. The format of a hexadecimal graphic string constant is: GX followed by a sequence of characters that starts and ends with an apostrophe. The characters between the apostrophes must be an even multiple of four hexadecimal digits. The number of hexadecimal digits must not exceed 16 336; otherwise, an error is returned (SQLSTATE -54002). If a hexadecimal graphic string constant is improperly formed, an error is returned (SQLSTATE 42606). Each group of four digits represents a single graphic character. In a Unicode database, this would be a single UCS-2 graphic character.

Examples:

```
GX'FFFF'
```

represents the bit pattern '1111111111111111' in a Unicode database.

```
GX'005200690063006B'
```

represents the VARGRAPHIC pattern of the ASCII string 'Rick' in a Unicode database.

UCS-2 graphic string constants

In a Unicode database, a hexadecimal UCS-2 graphic string that specifies a varying-length UCS-2 graphic string constant is supported. The format of a hexadecimal UCS-2 graphic string constant is: UX followed by a sequence of characters that starts and ends with an apostrophe. The characters between the apostrophes must be an even multiple of four hexadecimal digits. The number of hexadecimal digits must not exceed 16 336; otherwise, an error is returned (SQLSTATE -54002). If a hexadecimal UCS-2 graphic string constant is improperly formed, an error is returned (SQLSTATE 42606). Each group of four digits represents a single UCS-2 graphic character.

Example:

```
UX'0042006F006200620079'
```

represents the VARGRAPHIC pattern of the ASCII string 'Bobby'.

Special registers

A *special register* is a storage area that is defined for an application process by the database manager. It is used to store information that can be referenced in SQL statements. A reference to a special register is a reference to a value provided by the current server. If the value is a string, its CCSID is a default CCSID of the current server. The special registers can be referenced as follows:

Special registers

CURRENT CLIENT_ACCTNG	
CLIENT ACCTNG	
CURRENT CLIENT_APPLNAME	
CLIENT APPLNAME	
CURRENT CLIENT_USERID	
CLIENT USERID	
CURRENT CLIENT_WRKSTNNAME	
CLIENT WRKSTNNAME	
CURRENT DATE	
CURRENT_DATE (1)	
CURRENT DBPARTITIONNUM	
CURRENT DECFLOAT ROUNDING MODE	
CURRENT DEFAULT TRANSFORM GROUP	
CURRENT DEGREE	
CURRENT EXPLAIN MODE	
CURRENT EXPLAIN SNAPSHOT	
CURRENT FEDERATED ASYNCHRONY	
CURRENT IMPLICIT XMLPARSE OPTION	
CURRENT ISOLATION	
CURRENT LOCK TIMEOUT	
CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	
CURRENT MDC ROLLOUT MODE	
CURRENT OPTIMIZATION PROFILE	
CURRENT PACKAGE PATH	
CURRENT PATH	
CURRENT_PATH (1)	
CURRENT QUERY OPTIMIZATION	
CURRENT REFRESH AGE	
CURRENT SCHEMA	
CURRENT_SCHEMA (1)	
CURRENT SERVER	
CURRENT_SERVER (1)	
CURRENT TIME	
CURRENT_TIME (1)	
CURRENT TIMESTAMP	
CURRENT_TIMESTAMP (1)	
CURRENT TIMEZONE	
CURRENT_TIMEZONE (1)	
CURRENT USER	
CURRENT_USER (1)	
SESSION_USER	
USER	
SYSTEM_USER	

Notes:

- 1 The SQL2003 Core standard uses the form with the underscore.

Some special registers can be updated using the SET statement. The following table shows which of the special registers can be updated.

Table 14. Special Registers

Special Register	Updatable
CURRENT CLIENT_ACCTNG	No
CURRENT CLIENT_APPLNAME	No
CURRENT CLIENT_USERID	No
CURRENT CLIENT_WRKSTNNAME	No
CURRENT DATE	No
CURRENT DBPARTITIONNUM	No
CURRENT DECFLOAT ROUNDING MODE	No
CURRENT DEFAULT TRANSFORM GROUP	Yes
CURRENT DEGREE	Yes
CURRENT EXPLAIN MODE	Yes
CURRENT EXPLAIN SNAPSHOT	Yes
CURRENT FEDERATED ASYNCHRONY	Yes
CURRENT IMPLICIT XMLPARSE OPTION	Yes
CURRENT ISOLATION	Yes
CURRENT LOCK TIMEOUT	Yes
CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	Yes
CURRENT MDC ROLLOUT MODE	Yes
CURRENT OPTIMIZATION PROFILE	Yes
CURRENT PACKAGE PATH	Yes
CURRENT PATH	Yes
CURRENT QUERY OPTIMIZATION	Yes
CURRENT REFRESH AGE	Yes
CURRENT SCHEMA	Yes
CURRENT SERVER	No
CURRENT TIME	No
CURRENT TIMESTAMP	No
CURRENT TIMEZONE	No
CURRENT USER	No
SESSION_USER	Yes
SYSTEM_USER	No
USER	Yes

When a special register is referenced in a routine, the value of the special register in the routine depends on whether the special register is updatable or not. For non-updatable special registers, the value is set to the default value for the special register. For updatable special registers, the initial value is inherited from the invoker of the routine and can be changed with a subsequent SET statement inside the routine.

CURRENT_CLIENT_ACCTNG

The CURRENT_CLIENT_ACCTNG (or CLIENT ACCTNG) special register contains the value of the accounting string from the client information specified for this connection. The data type of the register is VARCHAR(255). The default value of this register is an empty string.

The value of the accounting string can be changed by using the Set Client Information (sqleseti) API.

Note that the value provided via the sqleseti API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

Example: Get the current value of the accounting string for this connection.

```
VALUES (CURRENT_CLIENT_ACCTNG)
INTO :ACCT_STRING
```

CURRENT_CLIENT_APPLNAME

The CURRENT_CLIENT_APPLNAME (or CLIENT APPLNAME) special register contains the value of the application name from the client information specified for this connection. The data type of the register is VARCHAR(255). The default value of this register is an empty string.

The value of the application name can be changed by using the Set Client Information (sqleseti) API.

Note that the value provided via the sqleseti API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

Example: Select which departments are allowed to use the application being used in this connection.

```
SELECT DEPT
FROM DEPT_APPL_MAP
WHERE APPL_NAME = CURRENT_CLIENT_APPLNAME
```

CURRENT_CLIENT_USERID

The CURRENT_CLIENT_USERID (or CLIENT USERID) special register contains the value of the client user ID from the client information specified for this connection. The data type of the register is VARCHAR(255). The default value of this register is an empty string.

The value of the client user ID can be changed by using the Set Client Information (sqleseti) API.

Note that the value provided via the sqleseti API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

Example: Find out in which department the current client user ID works.

```
SELECT DEPT
FROM DEPT_USERID_MAP
WHERE USER_ID = CURRENT_CLIENT_USERID
```

CURRENT_CLIENT_WRKSTNNAME

The CURRENT_CLIENT_WRKSTNNAME (or CLIENT_WRKSTNNAME) special register contains the value of the workstation name from the client information specified for this connection. The data type of the register is VARCHAR(255). The default value of this register is an empty string.

The value of the workstation name can be changed by using the Set Client Information (sqleseti) API.

Note that the value provided via the sqleseti API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

Example: Get the workstation name being used for this connection.

```
VALUES (CURRENT_CLIENT_WRKSTNNAME)
INTO :WS_NAME
```

CURRENT_DATE

The CURRENT_DATE (or CURRENT_DATE) special register specifies a date that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server. If this special register is used more than once within a single SQL statement, or used with CURRENT_TIME or CURRENT_TIMESTAMP within a single statement, all values are based on a single clock reading.

When used in an SQL statement inside a routine, CURRENT_DATE is not inherited from the invoking statement.

In a federated system, CURRENT_DATE can be used in a query intended for data sources. When the query is processed, the date returned will be obtained from the CURRENT_DATE register at the federated server, not from the data sources.

Example: Run the following command from the DB2 CLP to obtain the current date.

```
db2 values CURRENT_DATE
```

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_DBPARTITIONNUM

The CURRENT_DBPARTITIONNUM special register specifies an INTEGER value that identifies the coordinator node number for the statement. For statements issued from an application, the coordinator is the database partition to which the application connects. For statements issued from a routine, the coordinator is the database partition from which the routine is invoked.

CURRENT DBPARTITIONNUM

When used in an SQL statement inside a routine, CURRENT DBPARTITIONNUM is never inherited from the invoking statement.

CURRENT DBPARTITIONNUM returns 0 if the database instance is not defined to support database partitioning. (In other words, if there is no db2nodes.cfg file. For partitioned databases, the db2nodes.cfg file exists and contains database partition definitions.)

CURRENT DBPARTITIONNUM can be changed through the CONNECT statement, but only under certain conditions.

For compatibility with versions earlier than Version 8, the keyword NODE can be substituted for DBPARTITIONNUM.

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

```
VALUES CURRENT DBPARTITIONNUM
INTO :APPL_NODE
```

CURRENT DECFLOAT ROUNDING MODE

The CURRENT DECFLOAT ROUNDING MODE special register specifies the rounding mode that is used for DECFLOAT values.

The data type is VARCHAR(128). The following rounding modes are supported:

- ROUND_CEILING rounds the value towards positive infinity. If all of the discarded digits are zero or if the sign is negative, the result is unchanged (except for the removal of the discarded digits). Otherwise, the result coefficient is incremented by 1.
- ROUND_DOWN rounds the value towards 0 (truncation). The discarded digits are ignored.
- ROUND_FLOOR rounds the value towards negative infinity. If all of the discarded digits are zero or if the sign is positive, the result is unchanged (except for the removal of the discarded digits). Otherwise, the sign is negative and the result coefficient is incremented by 1.
- ROUND_HALF_EVEN rounds the value to the nearest value. If the values are equidistant, rounds the value so that the final digit is even. If the discarded digits represent more than half of the value of a number in the next left position, the result coefficient is incremented by 1. If they represent less than half, the result coefficient is not adjusted (that is, the discarded digits are ignored). Otherwise, the result coefficient is unaltered if its rightmost digit is even, or incremented by 1 if its rightmost digit is odd (to make an even digit).
- ROUND_HALF_UP rounds the value to the nearest value. If the values are equidistant, rounds the value up. If the discarded digits represent half or more than half of the value of a number in the next left position, the result coefficient is incremented by 1. Otherwise, the discarded digits are ignored.

The value of the DECFLOAT rounding mode on a client can be confirmed to match that of the server by invoking the SET CURRENT DECFLOAT ROUNDING MODE statement. However, this statement cannot be used to change the rounding mode of the server. The initial value of CURRENT DECFLOAT ROUNDING MODE is determined by the **decflt_rounding** database configuration parameter and can only be changed by changing the value of this database configuration parameter.

CURRENT DEFAULT TRANSFORM GROUP

The CURRENT DEFAULT TRANSFORM GROUP special register specifies a VARCHAR(18) value that identifies the name of the transform group used by dynamic SQL statements for exchanging user-defined structured type values with host programs. This special register does not specify the transform groups used in static SQL statements, or in the exchange of parameters and results with external functions or methods.

Its value can be set by the SET CURRENT DEFAULT TRANSFORM GROUP statement. If no value is set, the initial value of the special register is the empty string (a VARCHAR with a length of zero).

In a dynamic SQL statement (that is, one which interacts with host variables), the name of the transform group used for exchanging values is the same as the value of this special register, unless this register contains the empty string. If the register contains the empty string (no value was set by using the SET CURRENT DEFAULT TRANSFORM GROUP statement), the DB2_PROGRAM transform group is used for the transform. If the DB2_PROGRAM transform group is not defined for the structured type subject, an error is raised at run time (SQLSTATE 42741).

Examples:

Set the default transform group to MYSTRUCT1. The TO SQL and FROM SQL functions defined in the MYSTRUCT1 transform are used to exchange user-defined structured type variables with the host program.

```
SET CURRENT DEFAULT TRANSFORM GROUP = MYSTRUCT1
```

Retrieve the name of the default transform group assigned to this special register.

```
VALUES (CURRENT DEFAULT TRANSFORM GROUP)
```

CURRENT DEGREE

The CURRENT DEGREE special register specifies the degree of intra-partition parallelism for the execution of dynamic SQL statements. (For static SQL, the DEGREE bind option provides the same control.) The data type of the register is CHAR(5). Valid values are ANY or the string representation of an integer between 1 and 32 767, inclusive.

If the value of CURRENT DEGREE represented as an integer is 1 when an SQL statement is dynamically prepared, the execution of that statement will not use 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:

CURRENT DEGREE

- The value of the maximum query degree (**max_querydegree**) configuration parameter
- The application runtime degree
- The SQL statement compilation degree.

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.

The value can be changed by invoking the SET CURRENT DEGREE statement.

The initial value of CURRENT DEGREE is determined by the **dft_degree** database configuration parameter.

CURRENT EXPLAIN MODE

The CURRENT EXPLAIN MODE special register holds a VARCHAR(254) value which controls the behavior of the Explain facility with respect to eligible dynamic SQL statements. This facility generates and inserts Explain information into the Explain tables. This information does not include the Explain snapshot. Possible values are YES, EXPLAIN, NO, REOPT, RECOMMEND INDEXES, and EVALUATE INDEXES. (For static SQL, the EXPLAIN bind option provides the same control. In the case of the PREP and BIND commands, the EXPLAIN option values are: YES, NO, and ALL.)

YES Enables the Explain facility and causes Explain information for a dynamic SQL statement to be captured when the statement is compiled.

EXPLAIN

Enables the facility, but dynamic statements are not executed.

NO Disables the Explain facility.

REOPT

Enables the Explain facility and causes Explain information for a dynamic (or incremental-bind) SQL statement to be captured only when the statement is reoptimized using real values for the input variables (host variables, special registers, global variables, or parameter markers).

RECOMMEND INDEXES

Recommends a set of indexes for each dynamic query. Populates the ADVISE_INDEX table with the set of indexes.

EVALUATE INDEXES

Explains dynamic queries as though the recommended indexes existed. The indexes are picked up from the ADVISE_INDEX table.

The initial value is NO. The value can be changed by invoking the SET CURRENT EXPLAIN MODE statement.

The CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values interact when the Explain facility is invoked. The CURRENT EXPLAIN MODE special register also interacts with the EXPLAIN bind option. RECOMMEND INDEXES and EVALUATE INDEXES can only be set for the CURRENT EXPLAIN MODE register, and must be set using the SET CURRENT EXPLAIN MODE statement.

Example: Set the host variable EXPL_MODE (VARCHAR(254)) to the value currently in the CURRENT EXPLAIN MODE special register.

```
VALUES CURRENT EXPLAIN MODE
INTO :EXPL_MODE
```

CURRENT EXPLAIN SNAPSHOT

The CURRENT EXPLAIN SNAPSHOT special register holds a CHAR(8) value that controls the behavior of the Explain snapshot facility. This facility generates compressed information, including access plan information, operator costs, and bind-time statistics.

Only the following statements consider the value of this register: CALL, Compound SQL (Dynamic), DELETE, INSERT, MERGE, REFRESH, SELECT, SELECT INTO, SET INTEGRITY, UPDATE, VALUES, or VALUES INTO. Possible values are YES, EXPLAIN, NO, and REOPT. (For static SQL, the EXPLSNAP bind option provides the same control. In the case of the PREP and BIND commands, the EXPLSNAP option values are: YES, NO, and ALL.)

YES Enables the Explain snapshot facility and takes a snapshot of the internal representation of a dynamic SQL statement as the statement is compiled.

EXPLAIN

Enables the Explain snapshot facility, but dynamic statements are not executed.

NO Disables the Explain snapshot facility.

REOPT

Enables the Explain facility and causes Explain information for a dynamic (or incremental-bind) SQL statement to be captured only when the statement is reoptimized using real values for the input variables (host variables, special registers, global variables, or parameter markers).

The initial value is NO. The value can be changed by invoking the SET CURRENT EXPLAIN SNAPSHOT statement.

The CURRENT EXPLAIN SNAPSHOT and CURRENT EXPLAIN MODE special register values interact when the Explain facility is invoked. The CURRENT EXPLAIN SNAPSHOT special register also interacts with the EXPLSNAP bind option.

Example: Set the host variable EXPL_SNAP (char(8)) to the value currently in the CURRENT EXPLAIN SNAPSHOT special register.

```
VALUES CURRENT EXPLAIN SNAPSHOT
INTO :EXPL_SNAP
```

CURRENT FEDERATED ASYNCHRONY

The CURRENT FEDERATED ASYNCHRONY special register specifies the degree of asynchrony for the execution of dynamic SQL statements. (The FEDERATED_ASYNCHRONY bind option provides the same control for static SQL.) The data type of the register is INTEGER. Valid values are ANY (representing -1) or an integer between 0 and 32 767, inclusive. If, when an SQL statement is dynamically prepared, the value of CURRENT FEDERATED ASYNCHRONY is:

- 0, the execution of that statement will not use asynchrony

CURRENT FEDERATED ASYNCHRONY

- Greater than 0 and less than or equal to 32 767, the execution of that statement can involve asynchrony using the specified degree
- ANY (representing -1), the execution of that statement can involve asynchrony using a degree that is determined by the database manager

The value of the CURRENT FEDERATED ASYNCHRONY special register can be changed by invoking the SET CURRENT FEDERATED ASYNCHRONY statement.

The initial value of the CURRENT FEDERATED ASYNCHRONY special register is determined by the **federated_async** database manager configuration parameter if the dynamic statement is issued through the command line processor (CLP). The initial value is determined by the FEDERATED_ASYNCHRONY bind option if the dynamic statement is part of an application that is being bound.

Example: Set the host variable FEDASYNC (INTEGER) to the value of the CURRENT FEDERATED ASYNCHRONY special register.

```
VALUES CURRENT FEDERATED ASYNCHRONY INTO :FEDASYNC
```

CURRENT IMPLICIT XMLPARSE OPTION

The CURRENT IMPLICIT XMLPARSE OPTION special register specifies the whitespace handling options that are to be used when serialized XML data is implicitly parsed by the DB2 server, without validation. An implicit non-validating parse operation occurs when an SQL statement is processing an XML host variable or an implicitly or explicitly typed XML parameter marker that is not an argument of the XMLVALIDATE function. The data type of the register is VARCHAR(19).

The value of the CURRENT IMPLICIT XMLPARSE OPTION special register can be changed by invoking the SET CURRENT IMPLICIT XMLPARSE OPTION statement. Its initial value is 'STRIP WHITESPACE'.

Examples:

Retrieve the value of the CURRENT IMPLICIT XMLPARSE OPTION special register into a host variable named CURXMLPARSEOPT:

```
EXEC SQL VALUES (CURRENT IMPLICIT XMLPARSE OPTION) INTO :CURXMLPARSEOPT;
```

Set the CURRENT IMPLICIT XMLPARSE OPTION special register to 'PRESERVE WHITESPACE'.

```
SET CURRENT IMPLICIT XMLPARSE OPTION = 'PRESERVE WHITESPACE'
```

Whitespace is then preserved when the following SQL statement executes:

```
INSERT INTO T1 (XMLCOL1) VALUES (?)
```

CURRENT ISOLATION

The CURRENT ISOLATION special register holds a CHAR(2) value that identifies the isolation level (in relation to other concurrent sessions) for any dynamic SQL statements issued within the current session.

The possible values are:

(blanks)

Not set; use the isolation attribute of the package.

UR	Uncommitted Read
CS	Cursor Stability
RR	Repeatable Read
RS	Read Stability

The value of the CURRENT ISOLATION special register can be changed by the SET CURRENT ISOLATION statement.

Until a SET CURRENT ISOLATION statement is issued within a session, or after RESET has been specified for SET CURRENT ISOLATION, the CURRENT ISOLATION special register is set to blanks and is not applied to dynamic SQL statements; the isolation level used is taken from the isolation attribute of the package which issued the dynamic SQL statement. Once a SET CURRENT ISOLATION statement has been issued, the CURRENT ISOLATION special register provides the isolation level for any subsequent dynamic SQL statement compiled within the session, regardless of the settings for the package issuing the statement. This will remain in effect until the session ends or until a SET CURRENT ISOLATION statement is issued with the RESET option.

Example: Set the host variable ISOLATION_MODE (CHAR(2)) to the value currently stored in the CURRENT ISOLATION special register.

```
VALUES CURRENT ISOLATION
INTO :ISOLATION_MODE
```

CURRENT LOCK TIMEOUT

The CURRENT LOCK TIMEOUT special register specifies the number of seconds to wait for a lock before returning an error indicating that a lock cannot be obtained. This special register impacts row, table, index key, MDC block, and XML path (XPath) locks. The data type of the register is INTEGER.

Valid values for the CURRENT LOCK TIMEOUT special register are integers between -1 and 32767, inclusive. This special register can also be set to the null value. A value of -1 specifies that timeouts are not to take place, and that the application is to wait until the lock is released or a deadlock is detected. A value of 0 specifies that the application is not to wait for a lock; if a lock cannot be obtained, an error is to be returned immediately.

The value of the CURRENT LOCK TIMEOUT special register can be changed by invoking the SET CURRENT LOCK TIMEOUT statement. Its initial value is null; in this case, the current value of the **locktimeout** database configuration parameter is used when waiting for a lock, and this value will be returned for the special register.

CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION

The CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register specifies a VARCHAR(254) value that identifies the types of tables that can be considered when optimizing the processing of dynamic SQL queries. Materialized query tables are never considered by static embedded SQL queries.

The initial value of CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION is SYSTEM. Its value can be changed by the SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION statement.

CURRENT MDC ROLLOUT MODE

CURRENT MDC ROLLOUT MODE

The CURRENT MDC ROLLOUT MODE special register specifies the behavior on multidimensional clustering (MDC) tables of DELETE statements that qualify for rollout processing.

The default value of this register is determined by the DB2_MDC_ROLLOUT registry variable. The value can be changed by invoking the SET CURRENT MDC ROLLOUT MODE statement. When the CURRENT MDC ROLLOUT MODE special register is set to a particular value, the execution behavior of subsequent DELETE statements that qualify for rollout is impacted. The DELETE statement does not need to be recompiled for the behavior to change.

CURRENT OPTIMIZATION PROFILE

The CURRENT OPTIMIZATION PROFILE special register specifies the qualified name of the optimization profile to be used by DML statements that are dynamically prepared for optimization.

The initial value is the null value. The value can be changed by invoking the SET CURRENT OPTIMIZATION PROFILE statement. An optimization profile that is not qualified with a schema name will be implicitly qualified with the value of the CURRENT DEFAULT SCHEMA special register.

Example 1: Set the optimization profile to 'JON.SALES'.

```
SET CURRENT OPTIMIZATION PROFILE = JON.SALES
```

Example 2: Get the current value of the optimization profile name for this connection.

```
VALUES (CURRENT OPTIMIZATION PROFILE) INTO :PROFILE
```

CURRENT PACKAGE PATH

The CURRENT PACKAGE PATH special register specifies a VARCHAR(4096) value that identifies the path to be used when resolving references to packages that are needed when executing SQL statements.

The value can be an empty or a blank string, or a list of one or more schema names that are delimited with double quotation marks and separated by commas. Any double quotation marks appearing as part of the string will need to be represented as two double quotation marks, as is common practice with delimited identifiers. The delimiters and commas contribute to the length of the special register.

This special register applies to both static and dynamic statements.

The initial value of CURRENT PACKAGE PATH in a user-defined function, method, or procedure is inherited from the invoking application. In other contexts, the initial value of CURRENT PACKAGE PATH is an empty string. The value is a list of schemas only if the application process has explicitly specified a list of schemas by means of the SET CURRENT PACKAGE PATH statement.

Examples:

An application will be using multiple SQLJ packages (in schemas SQLJ1 and SQLJ2) and a JDBC package (in schema DB2JAVA). Set the CURRENT PACKAGE PATH special register to check SQLJ1, SQLJ2, and DB2JAVA, in that order.

```
SET CURRENT PACKAGE PATH = "SQLJ1", "SQLJ2", "DB2JAVA"
```

Set the host variable HVPKLIST to the value currently stored in the CURRENT PACKAGE PATH special register.

```
VALUES CURRENT PACKAGE PATH INTO :HVPKLIST
```

CURRENT PATH

The CURRENT PATH (or CURRENT_PATH) special register specifies a VARCHAR(2048) value that identifies the SQL path to be used when resolving function references and data type references in dynamically prepared SQL statements. CURRENT FUNCTION PATH is a synonym for CURRENT PATH. 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 FUNCPATH bind option provides an SQL path that is used for function and data type resolution.

The CURRENT PATH special register contains a list of one or more schema names that are enclosed by double quotation marks and separated by commas. For example, an SQL path specifying that the database manager is to look first in the FERMAT schema, then in the XGRAPHIC schema, and finally in the SYSIBM schema, is returned in the CURRENT PATH special register as:

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

The default value is "SYSIBM", "SYSFUN", "SYSPROC", "SYSIBMADM", X, where X is the value of the USER special register, delimited by double quotation marks. The value can be changed by invoking the SET CURRENT PATH statement. The schema SYSIBM does not need to be specified. If it is not included in the SQL path, it is implicitly assumed to be the first schema. SYSIBM does not take up any of the 2048 bytes if it is implicitly assumed.

A data type that is not qualified with a schema name will be implicitly qualified with the first schema in the SQL path that contains a data type with the same unqualified name. There are exceptions to this rule, as outlined in the descriptions of the following statements: CREATE TYPE (Distinct), CREATE FUNCTION, COMMENT, and DROP.

Example: Using the SYSCAT.ROUTINES catalog view, find all user-defined routines that can be invoked without qualifying the routine name, because the CURRENT PATH special register contains the schema name.

```
SELECT ROUTINENAME, ROUTINESCHEMA FROM SYSCAT.ROUTINES  
WHERE POSITION (ROUTINESCHEMA, CURRENT PATH, CODEUNITS16) <> 0
```

CURRENT QUERY OPTIMIZATION

The CURRENT QUERY OPTIMIZATION special register specifies an INTEGER value that controls the class of query optimization performed by the database manager when binding dynamic SQL statements. The QUERYOPT bind option controls the class of query optimization for static SQL statements. The possible values range from 0 to 9. For example, if the query optimization class is set to 0 (minimal optimization), then the value in the special register is 0. The default value

CURRENT QUERY OPTIMIZATION

is determined by the `dft_queryopt` database configuration parameter. The value can be changed by invoking the `SET CURRENT QUERY OPTIMIZATION` statement.

Example: Using the `SYSCAT.PACKAGES` catalog view, find all plans that were bound with the same setting as the current value of the `CURRENT QUERY OPTIMIZATION` special register.

```
SELECT PKGNAME, PKGSHEMA 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)`. It is the maximum duration since a particular timestamped event occurred to a cached data object (for example, a `REFRESH TABLE` statement processed on a system-maintained `REFRESH DEFERRED` materialized query table), such that the cached data object can be used to optimize the processing of a query. If `CURRENT REFRESH AGE` has a value of 99 999 999 999 999, and the query optimization class is 5 or more, the types of tables specified in `CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION` are considered when optimizing the processing of a dynamic SQL query.

The value of `CURRENT REFRESH AGE` must be 0 or 99 999 999 999 999. The initial value is 0. The value can be changed by invoking the `SET CURRENT REFRESH AGE` statement.

CURRENT SCHEMA

The `CURRENT SCHEMA` (or `CURRENT_SCHEMA`) special register specifies a `VARCHAR(128)` value that identifies the schema name used to qualify database object references, where applicable, in dynamically prepared SQL statements. For compatibility with DB2 for z/OS, `CURRENT SQLID` (or `CURRENT_SQLID`) can be specified in place of `CURRENT SCHEMA`.

The initial value of `CURRENT SCHEMA` is the authorization ID of the current session user. The value can be changed by invoking the `SET SCHEMA` statement.

The `QUALIFIER bind` option controls the schema name used to qualify database object references, where applicable, for static SQL statements.

Example: Set the schema for object qualification to 'D123'.

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

CURRENT SERVER

The `CURRENT SERVER` (or `CURRENT_SERVER`) special register specifies a `VARCHAR(18)` value that identifies the current application server. The register contains the actual name of the application server, not an alias.

`CURRENT SERVER` can be changed through the `CONNECT` statement, but only under certain conditions.

When used in an SQL statement inside a routine, `CURRENT SERVER` is not inherited from the invoking statement.

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

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

CURRENT TIME

The CURRENT TIME (or CURRENT_TIME) special register specifies a time that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server. If this special register is used more than once within a single SQL statement, or used with CURRENT DATE or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.

When used in an SQL statement inside a routine, CURRENT TIME is not inherited from the invoking statement.

In a federated system, CURRENT TIME can be used in a query intended for data sources. When the query is processed, the time returned will be obtained from the CURRENT TIME register at the federated server, not from the data sources.

Example: Run the following command from the DB2 CLP to obtain the current time.

```
db2 values CURRENT TIME
```

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 (or CURRENT_TIMESTAMP) special register specifies a timestamp that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server. If this special register is used more than once within a single SQL statement, or used with CURRENT DATE or CURRENT TIME within a single statement, all values are based on a single clock reading. It is possible for separate CURRENT TIMESTAMP special register requests to return the same value; if unique values are required, consider using the GENERATE_UNIQUE function, a sequence, or an identity column.

When used in an SQL statement inside a routine, CURRENT TIMESTAMP is not inherited from the invoking statement.

In a federated system, CURRENT TIMESTAMP can be used in a query intended for data sources. When the query is processed, the timestamp returned will be obtained from the CURRENT TIMESTAMP register at the federated server, not from the data sources.

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

CURRENT TIMEZONE

The CURRENT TIMEZONE (or CURRENT_TIMEZONE) special register specifies the difference between UTC (Coordinated Universal Time, formerly known as GMT) and local time at the application server. The difference is represented by a time duration (a decimal number in which the first two digits are the number of hours, the next two digits are the number of minutes, and the last two digits are the number of seconds). The number of hours is between -24 and 24 exclusive. Subtracting CURRENT TIMEZONE from a local time converts that local time to UTC. The time is calculated from the operating system time at the moment the SQL statement is executed. (The CURRENT TIMEZONE value is determined from C runtime functions.)

The CURRENT TIMEZONE special register can be used wherever an expression of the DECIMAL(6,0) data type is used; for example, in time and timestamp arithmetic.

When used in an SQL statement inside a routine, CURRENT TIMEZONE is not inherited from the invoking statement.

Example: Insert a record into the IN_TRAY table, using a UTC timestamp for the RECEIVED column.

```
INSERT INTO IN_TRAY VALUES (  
    CURRENT_TIMESTAMP - CURRENT_TIMEZONE,  
    :source,  
    :subject,  
    :notetext )
```

CURRENT USER

The CURRENT USER (or CURRENT_USER) special register specifies the authorization ID that is to be used for statement authorization. For static SQL statements, the value represents the authorization ID that is used when the package is bound. For dynamic SQL statements, the value is the same as the value of the SESSION_USER special register for packages bound with the DYNAMICRULES(RUN) bind option. The data type of the register is VARCHAR(128).

Example: Select table names whose schema matches the value of the CURRENT USER special register.

```
SELECT TABNAME FROM SYSCAT.TABLES  
WHERE TABSCHEMA = CURRENT_USER AND TYPE = 'T'
```

If this statement is executed as a static SQL statement, it returns the tables whose schema name matches the binder of the package that includes the statement. If this statement is executed as a dynamic SQL statement, it returns the tables whose schema name matches the current value of the SESSION_USER special register.

SESSION_USER

The SESSION_USER special register specifies the authorization ID that is to be used for the current session. The value of this register is used for authorization checking of dynamic SQL statements when DYNAMICRULES run behavior is in effect for the package. The data type of the register is VARCHAR(128).

The initial value of SESSION_USER for a new connection is the same as the value of the SYSTEM_USER special register. Its value can be changed by invoking the SET SESSION AUTHORIZATION statement.

SESSION_USER is a synonym for the USER special register.

Example: Determine what routines can be executed using dynamic SQL. Assume DYNAMICRULES run behavior is in effect for the package that will issue the dynamic SQL statement that invokes the routine.

```
SELECT SCHEMA, SPECIFICNAME FROM SYSCAT.ROUTINEAUTH
WHERE GRANTEE = SESSION_USER
AND EXECUTEAUTH IN ('Y', 'G')
```

SYSTEM_USER

The SYSTEM_USER special register specifies the authorization ID of the user that connected to the database. The value of this register can only be changed by connecting as a user with a different authorization ID. The data type of the register is VARCHAR(128).

See “Example” in the description of the SET SESSION AUTHORIZATION statement.

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).

When used in an SQL statement inside a routine, USER is not inherited from the invoking statement.

Example: Select all notes from the IN_TRAY table that were placed there by the user.

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

Global variables

Global variables are named memory variables that you can access and modify through SQL statements.

Global variables enable you to share relational data between SQL statements without the need for application logic to support this data transfer. You can control access to global variables through the GRANT (Global Variable Privileges) and REVOKE (Global Variable Privileges) statements.

DB2 supports created session global variables. A *session global variable* is associated with a specific session, and contains a value that is unique to that session. A created session global variable is available to any active SQL statement running against the database on which the variable was defined. A session global variable can be associated with more than one session, but its value will be specific to each session. Created session global variables and the privileges that are associated with them are defined in the system catalog.

Global variables

Global variable names are qualified two-part names. When a global variable is referenced without the schema name, the SQL path is used for name resolution. For static SQL statements, the SQL path is specified using the FUNCSPATH bind option. For dynamic SQL statements, the SQL path is the value of the CURRENT PATH special register.

For static SQL statements in packages, global variables are resolved during a bind operation. In views, triggers, and SQL routines they are resolved when the database object is created. If resolution is performed again on any global variable, it could change the behavior if, for example, a new global variable has been added with the same name in a different schema that is also on the SQL path. To avoid this, the database manager applies conservative binding semantics wherever necessary. That means that only global variables in the SQL path that were defined before the last explicit bind timestamp are considered for global variable name resolution.

When a global variable is referenced within an SQL statement or within a trigger, view, or routine, a dependency on the fully qualified global variable name is recorded for the statement or object. Also, if applicable, the authorization ID being used for the statement is checked for the READ privilege on the global variable.

Global variables can be referenced within any expression that does not need to be deterministic. Deterministic expressions are required in the following situations, which preclude the use of global variables:

- Check constraints
- Definitions of generated columns
- Refresh immediate materialized query tables (MQTs)

The value of a global variable can be changed using the SET, SELECT INTO, or VALUES INTO statement. It can also be changed if it is an argument of an OUT or INOUT parameter in a CALL statement.

Functions

A *function* is an operation denoted by a function name followed by one or more operands that are enclosed in parentheses. For example, the `TIMESTAMP` function can be passed input data values of type `DATE` and `TIME`, and the result is a `TIMESTAMP`. Functions can be either built-in or user-defined.

- *Built-in functions* are provided with the database manager. They return a single result value and are identified as part of the `SYSIBM` schema. Such functions include column functions (for example, `AVG`), operator functions (for example, `+`), and casting functions (for example, `DECIMAL`).
- *User-defined functions* are functions that are registered to a database in `SYSCAT.ROUTINES` (using the `CREATE FUNCTION` statement). User-defined functions are never part of the `SYSIBM` schema. One set of such functions is provided with the database manager in a schema called `SYSFUN`.

User-defined functions extend the capabilities of the database system by adding function definitions (provided by users or third party vendors) that can be applied in the database engine itself. Extending database functions lets the database exploit the same functions in the engine that an application uses, providing more synergy between application and database.

External, SQL, and sourced user-defined functions

A user-defined function can be an external function, an SQL function, or a sourced function. An *external function* is defined to the database with a reference to an object code library, and a function within that library that will be executed when the function is invoked. External functions cannot be column functions. An *SQL function* is defined to the database using only the SQL RETURN statement. It can return either a scalar value, a row, or a table. SQL functions cannot 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 useful for supporting existing functions with user-defined types.

Scalar, column, row, and table user-defined functions

Each user-defined function is also categorized as a scalar, column, or table function. A *scalar function* is a function that returns a single-valued answer each time it is called. For example, the built-in function SUBSTR() is a scalar function. Scalar UDFs can be either external or sourced.

A *column function* is one which conceptually is passed a set of like values (a column) and returns a single-valued answer. 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 *row function* is a function that returns one row of values. It may only be used as a transform function, mapping attribute values of a structured type into values in a row. A row function must be defined as an SQL function.

A *table function* is a function that returns a table to the SQL statement which references it. It may only be referenced in the FROM clause of a SELECT statement. Such a function can be used to apply SQL language processing power to data that is not DB2 data, or to convert such data into a DB2 table. It could, for example, take a file and convert it into 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 be defined as an external function or as an SQL function. (A table function cannot be a sourced function.)

Function signatures

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; for example, TEST.RISK(INTEGER). 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. These functions must have different parameter types. A function name can also be overloaded in an SQL path,

Functions

in which case there is more than one function by that name in the path. These functions do not necessarily have different parameter types.

A function can be invoked by referring (in an allowable context) to its 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 SQL 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, the SQL path is specified using the FUNCSPATH bind option. For dynamic SQL statements, the SQL path is the value of the CURRENT PATH special register.

Access to functions is controlled through the EXECUTE privilege. GRANT and REVOKE statements are used to specify who can or cannot execute a specific function or a set of functions. The EXECUTE privilege (or DBADM authority) is needed to invoke a function. The definer of the function automatically receives the EXECUTE privilege. The definer of an external function or an SQL function having the WITH GRANT option on all underlying objects also receives the WITH GRANT option with the EXECUTE privilege on the function. The definer (or SYSADM or DBADM) must then grant it to the user who wants to invoke the function from any SQL statement, reference the function in any DDL statement (such as CREATE VIEW, CREATE TRIGGER, or when defining a constraint), or create another function sourced on this function. If the EXECUTE privilege is not granted to a user, the function will not be considered by the function resolution algorithm, even if it is a better match. Built-in functions (SYSIBM functions) and SYSFUN functions have the EXECUTE privilege implicitly granted to PUBLIC.

Function resolution

After 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.

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 (a built-in function) or by a user (a user-defined function), its parameters (zero or more) are specified, and the order of their definitions defines their positions and their semantics. Therefore, every parameter is a particular positional input to a function. 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, EXECUTE privilege on the function, 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:

- 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)
```


an 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, because SMALLINT can be promoted to INTEGER and is a better match than DOUBLE which is further down the precedence list.

When considering arguments that are structured types, the precedence list includes the supertypes of the static type of the argument. The best fit is the function defined with the supertype parameter that is closest in the structured type hierarchy to the static type of the function argument.

- No 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.

- 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 an SQL path of:

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

the following function reference:

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

will choose TEST.RANDOM, because 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.

Determining 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 types of the results of the functions, or the type of function (column, scalar, or table) under consideration do not enter into this determination.

Function resolution is performed using the following steps:

1. First, find all functions from the catalog (SYSCAT.ROUTINES), and built-in functions, such that all of the following are true:
 - For invocations where the schema name was specified (a qualified reference), the schema name and the function name match the invocation name.
 - For invocations where the schema name was not specified (an unqualified reference), the function name matches the invocation name and has a schema name that matches one of the schemas in the SQL path.
 - The invoker has the EXECUTE privilege on the function.

Functions

- The number of defined parameters matches the invocation.
 - Each invocation argument matches the function's corresponding defined parameter in data type, or is "promotable" to it.
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 for which there exists a function with a parameter of that data type. Length, precision, scale, and the FOR BIT DATA attribute are not considered in this comparison. For example, a DECIMAL(9,1) argument is considered an exact match for a DECIMAL(6,5) parameter, a DECFLOAT(34) argument is considered an exact match for a DECFLOAT(16) parameter, and a VARCHAR(19) argument is an exact match for a VARCHAR(6) parameter.

The best match for a user-defined structured-type argument is itself; the next best match is its immediate supertype, and so on for each supertype of the argument. Note that only the static type (declared type) of the structured-type argument is considered, not the dynamic type (most specific type).
 3. If more than one candidate function remains after Step 2, all remaining candidate functions must have identical signatures but be 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).

SQL path considerations for built-in functions

Built-in functions reside in a special schema called SYSIBM. Additional functions are available in the SYSFUN and SYSPROC schemas, but are not considered built-in functions because they are developed as user-defined functions and have no special processing considerations. Users cannot define additional functions in the SYSIBM, SYSFUN, or SYSPROC schemas (or in any other 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 functions must always be considered during function resolution. Therefore, omission of SYSIBM from the path results in the assumption (for function and data type resolution) that SYSIBM is the first schema on the path.

For example, if a user's SQL path is defined as:

```
"SHAREFUN", "SYSIBM", "SYSFUN"
```

and there is a LENGTH function defined in schema SHAREFUN with the same number and types of arguments as SYSIBM.LENGTH, then an unqualified reference to LENGTH in this user's SQL statement will result in selecting SHAREFUN.LENGTH. However, if the user's SQL path is defined as:

```
"SHAREFUN", "SYSFUN"
```

and the same SHAREFUN.LENGTH function exists, then an unqualified reference to LENGTH in this user's SQL statement will result in selecting SYSIBM.LENGTH, because SYSIBM implicitly appears first in the path.

To minimize potential problems in this area:

- Never use the names of built-in functions for user-defined functions.

- If, for some reason, it is necessary to create a user-defined function with the same name as a built-in function, be sure to qualify any references to it.

Note: An unqualified reference to a function named *function-name* invokes the built-in function *function-name* that is in the schema SYSIBM. This function cannot be explicitly qualified with the schema name.

Example of function resolution

Following is an example of successful function resolution. (Note that not all required keywords are shown.)

There are seven ACT functions, in three different schemas, registered as:

```
CREATE FUNCTION AUGUSTUS.ACT (CHAR(5), INT, DOUBLE) SPECIFIC ACT_1 ...
CREATE FUNCTION AUGUSTUS.ACT (INT, INT, DOUBLE) SPECIFIC ACT_2 ...
CREATE FUNCTION AUGUSTUS.ACT (INT, INT, DOUBLE, INT) SPECIFIC ACT_3 ...
CREATE FUNCTION JULIUS.ACT (INT, DOUBLE, DOUBLE) SPECIFIC ACT_4 ...
CREATE FUNCTION JULIUS.ACT (INT, INT, DOUBLE) SPECIFIC ACT_5 ...
CREATE FUNCTION JULIUS.ACT (SMALLINT, INT, DOUBLE) SPECIFIC ACT_6 ...
CREATE FUNCTION JULIUS.ACT (INT, INT, DECFLOAT) SPECIFIC ACT_7 ...
CREATE FUNCTION NERO.ACT (INT, INT, DEC(7,2)) SPECIFIC ACT_8 ...
```

The function reference is as follows (where I1 and I2 are INTEGER columns, and D is a DECIMAL column):

```
SELECT ... ACT(I1, I2, D) ...
```

Assume that the application making this reference has an SQL path established as:

```
"JULIUS", "AUGUSTUS", "CAESAR"
```

Following through the algorithm...

- The function with specific name ACT_8 is eliminated as a candidate, because the schema NERO is not included in the SQL path.
- The function with specific name ACT_3 is eliminated as a candidate, because it has the wrong number of parameters. ACT_1 and ACT_6 are eliminated because, in both cases, the first argument cannot be promoted to the data type of the first parameter.
- Because there is more than one candidate remaining, the arguments are considered in order.
- For the first argument, the remaining functions, ACT_2, ACT_4, ACT_5, and ACT_7 are an exact match with the argument type. No functions can be eliminated from consideration; therefore the next argument must be examined.
- For this second argument, ACT_2, ACT_5, and ACT_7 are exact matches, but ACT_4 is not, so it is eliminated from consideration. The next argument is examined to determine some differentiation among ACT_2, ACT_5, and ACT_7.
- For the third and last argument, neither ACT_2, ACT_5, nor ACT_7 match the argument type exactly. Although ACT_2 and ACT_5 are equally good, ACT_7 is not as good as the other two because the type DOUBLE is closer to DECIMAL than is DECFLOAT. ACT_7 is eliminated..
- There are two functions remaining, ACT_2 and ACT_5, with identical parameter signatures. The final tie-breaker is to see which function's schema comes first in the SQL path, and on this basis, ACT_5 is the function chosen.

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:

- The function was referenced in a FROM clause, but the function selected by the function resolution step was a scalar or column function.
- 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. This includes the case where precision, scale, or length differs between the argument and the parameter.

Conservative binding semantics

There are instances in which routines, data types, and global variables are resolved when a statement is processed, and the database manager must be able to repeat this resolution. This is true in:

- Static DML statements in packages
- Views
- Triggers
- Check constraints
- SQL routines

For static DML statements in packages, the routine, data type, or global variable references are resolved during a bind operation. Routine, data type, and global variable references in views, triggers, SQL routines, and check constraints are resolved when the database object is created.

If routine resolution is performed again on any routine references in these objects, it could change the behavior if:

- A new routine has been added with a signature that is a better match, but the actual executable performs different operations.
- The definer has been granted the execute privilege on a routine with a signature that is a better match, but the actual executable performs different operations.

Similarly, if resolution is performed again on any data type, global variable, or routine in these objects, it could change the behavior if a new data type, global variable, or routine has been added with the same name in a different schema that is also on the SQL path. To avoid this, the database manager applies *conservative binding semantics* wherever necessary. This ensures that routine, data type, and global variable references will be resolved using the same SQL path and the set of routines to which it previously resolved when it was bound. The creation timestamp of routines, data types, and global variables considered during resolution is not later than the time when the statement was bound. (Built-in functions added starting with Version 6.1 have a creation timestamp that is based on the time of database creation or migration.) In this way, only the routines, data types, and global variables that were considered during routine, data type, and global variable resolution when the statement was originally processed will be considered. Hence, newly created or authorized routines, data types, and global variables are not considered when conservative binding semantics are applied.

Consider a database with two functions that have the signatures SCHEMA1.BAR(INTEGER) and SCHEMA2.BAR(DOUBLE). Assume the SQL path contains both schemas SCHEMA1 and SCHEMA2 (although their order within the SQL path does not matter). USER1 has been granted the EXECUTE privilege on the function SCHEMA2.BAR(DOUBLE). Suppose USER1 creates a view that calls BAR(INT_VAL). This will resolve to the function SCHEMA2.BAR(DOUBLE). The view will always use SCHEMA2.BAR(DOUBLE), even if someone grants USER1 the EXECUTE privilege on SCHEMA1.BAR(INTEGER) after the view has been created.

For static DML in packages, the packages can rebind implicitly, or by explicitly issuing the REBIND command (or corresponding API), or the BIND command (or corresponding API). The implicit rebind is always performed to resolve routines, data types, and global variables with conservative binding semantics. The REBIND command provides the option to resolve with conservative binding semantics (RESOLVE CONSERVATIVE) or to resolve by considering any new routines, data types, or global variables (RESOLVE ANY, the default option).

Implicit rebind of a package always resolves the same routine. Even if EXECUTE privilege on a better-matched routine was granted, that routine will not be considered. Explicit rebind of a package can result in a different routine being selected. (But if RESOLVE CONSERVATIVE is specified, routine resolution will follow conservative binding semantics).

If a routine is specified during the creation of a view, trigger, constraint, or SQL routine body, the specific instance of the routine to be used is determined by routine resolution at the time the object is created. Subsequent granting of the EXECUTE privilege after the object has been created will not change the specific routine that the object uses.

Consider a database with two functions that have the signatures SCHEMA1.BAR(INTEGER) and SCHEMA2.BAR(DOUBLE). USER1 has been granted the EXECUTE privilege on the function SCHEMA2.BAR(DOUBLE). Suppose USER1 creates a view that calls BAR(INT_VAL). This will resolve to the function SCHEMA2.BAR(DOUBLE). The view will always use SCHEMA2.BAR(DOUBLE), even if someone grants USER1 the EXECUTE privilege on SCHEMA1.BAR(INTEGER) after the view has been created.

The same behavior occurs in other database objects. For example, if a package is implicitly rebound (perhaps after dropping an index), the package will refer to the

Functions

same specific routine both before and after the implicit rebind. An *explicit* rebind of a package, however, can result in a different routine being selected.

Methods

A database method of a structured type is a relationship between a set of input data values and a set of result values, where the first input value (or *subject argument*) has the same type, or is a subtype of the subject type (also called the *subject parameter*), of the method. For example, a method called CITY, of type ADDRESS, can be passed input data values of type VARCHAR, and the result is an ADDRESS (or a subtype of ADDRESS).

Methods are defined implicitly or explicitly, as part of the definition of a user-defined structured type.

Implicitly defined methods are created for every structured type. *Observer methods* are defined for each attribute of the structured type. Observer methods allow applications to get the value of an attribute for an instance of the type. *Mutator methods* are also defined for each attribute, allowing applications to mutate the type instance by changing the value for an attribute of a type instance. The CITY method described above is an example of a mutator method for the type ADDRESS.

Explicitly defined methods, or *user-defined methods*, are methods that are registered to a database in SYSCAT.ROUTINES, by using a combination of CREATE TYPE (or ALTER TYPE ADD METHOD) and CREATE METHOD statements. All methods defined for a structured type are defined in the same schema as the type.

User-defined methods for structured types extend the function of the database system by adding method definitions (provided by users or third party vendors) that can be applied to structured type instances in the database engine. Defining database methods lets the database exploit the same methods in the engine that an application uses, providing more synergy between application and database.

External and SQL user-defined methods

A user-defined method can be either external or based on an SQL expression. An external method is defined to the database with a reference to an object code library and a function within that library that will be executed when the method is invoked. A method based on an SQL expression returns the result of the SQL expression when the method is invoked. Such methods do not require any object code library, because they are written completely in SQL.

A user-defined method can return a single-valued answer each time it is called. This value can be a structured type. A method can be defined as *type preserving* (using SELF AS RESULT), to allow the dynamic type of the subject argument to be returned as the returned type of the method. All implicitly defined mutator methods are type preserving.

Method signatures

A method is identified by its subject type, a method name, the number of parameters, and the data types of its parameters. This is called a *method signature*, and it must be unique within the database.

There can be more than one method with the same name for a structured type, provided that:

- The number of parameters or the data types of the parameters are different, or
- The methods are part of the same method hierarchy (that is, the methods are in an overriding relationship or override the same original method), or
- The same function signature (using the subject type or any of its subtypes or supertypes as the first parameter) does not exist.

A method name that has multiple method instances is called an *overloaded method*. A method name can be overloaded within a type, in which case there is more than one method by that name for the type (all of which have different parameter types). A method name can also be overloaded in the subject type hierarchy, in which case there is more than one method by that name in the type hierarchy. These methods must have different parameter types.

A method can be invoked by referring (in an allowable context) to the method name, preceded by both a reference to a structured type instance (the subject argument), and the double dot operator. A list of arguments enclosed in parentheses must follow. Which method is actually invoked depends on the static type of the subject type, using the method resolution process described in the following section. Methods defined WITH FUNCTION ACCESS can also be invoked using function invocation, in which case the regular rules for function resolution apply.

If function resolution results in a method defined WITH FUNCTION ACCESS, all subsequent steps of method invocation are processed.

Access to methods is controlled through the EXECUTE privilege. GRANT and REVOKE statements are used to specify who can or cannot execute a specific method or a set of methods. The EXECUTE privilege (or DBADM authority) is needed to invoke a method. The definer of the method automatically receives the EXECUTE privilege. The definer of an external method or an SQL method having the WITH GRANT option on all underlying objects also receives the WITH GRANT option with the EXECUTE privilege on the method. The definer (or SYSADM or DBADM) must then grant it to the user who wants to invoke the method from any SQL statement, or reference the method in any DDL statement (such as CREATE VIEW, CREATE TRIGGER, or when defining a constraint). If the EXECUTE privilege is not granted to a user, the method will not be considered by the method resolution algorithm, even if it is a better match.

Method resolution

After method invocation, the database manager must decide which of the possible methods with the same name is the “best fit”. Functions (built-in or user-defined) are not considered during method resolution.

An *argument* is a value passed to a method upon invocation. When a method is invoked in SQL, it is passed the subject argument (of some structured type) and a list of zero or more arguments. They are positional in that the semantics of an argument are determined by its position in the argument list. A *parameter* is a formal definition of an input to a method. When a method is defined to the database, either implicitly (system-generated for a type) or by a user (a user-defined method), its parameters are specified (with the subject parameter as the first parameter), and the order of their definitions defines their positions and their semantics. Therefore, every parameter is a particular positional input to a

Methods

method. On invocation, an argument corresponds to a particular parameter by virtue of its position in the list of arguments.

The database manager uses the name of the method given in the invocation, EXECUTE privilege on the method, the number and data types of the arguments, all the methods with the same name for the subject argument's static type (and its supertypes), and the data types of their corresponding parameters as the basis for deciding whether or not to select a method. The following are the possible outcomes of the decision process:

- A particular method is deemed to be the best fit. For example, given the methods named RISK for the type SITE with signatures defined as:

```
PROXIMITY(INTEGER) FOR SITE  
PROXIMITY(DOUBLE) FOR SITE
```

the following method invocation (where ST is a SITE column, DB is a DOUBLE column):

```
SELECT ST..PROXIMITY(DB) ...
```

then, the second PROXIMITY will be chosen.

The following method invocation (where SI is a SMALLINT column):

```
SELECT ST..PROXIMITY(SI) ...
```

would choose the first PROXIMITY, because SMALLINT can be promoted to INTEGER and is a better match than DOUBLE, which is further down the precedence list.

When considering arguments that are structured types, the precedence list includes the supertypes of the static type of the argument. The best fit is the function defined with the supertype parameter that is closest in the structured type hierarchy to the static type of the function argument.

- No method is deemed to be an acceptable fit. For example, given the same two functions in the previous case and the following function reference (where C is a CHAR(5) column):

```
SELECT ST..PROXIMITY(C) ...
```

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

- A particular method is selected based on the methods in the type hierarchy and the number and data types of the arguments passed on invocation. For example, given methods named RISK for the types SITE and DRILLSITE (a subtype of SITE) with signatures defined as:

```
RISK(INTEGER) FOR DRILLSITE  
RISK(DOUBLE) FOR SITE
```

and the following method invocation (where DRST is a DRILLSITE column, DB is a DOUBLE column):

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

the second RISK will be chosen, because DRILLSITE can be promoted to SITE.

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

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

would choose the first RISK, because SMALLINT can be promoted to INTEGER, which is closer on the precedence list than DOUBLE, and DRILLSITE is a better match than SITE, which is a supertype.

Methods within the same type hierarchy cannot have the same signatures, considering parameters other than the subject parameter.

Determining the best fit

A comparison of the data types of the arguments with the defined data types of the parameters of the methods under consideration forms the basis for the decision of which method in a group of like-named methods is the “best fit”. Note that the data types of the results of the methods under consideration do not enter into this determination.

Method resolution is performed using the following steps:

1. First, find all methods from the catalog (SYSCAT.ROUTINES) such that all of the following are true:
 - The method name matches the invocation name, and the subject parameter is the same type or is a supertype of the static type of the subject argument.
 - The invoker has the EXECUTE privilege on the method.
 - The number of defined parameters matches the invocation.
 - Each invocation argument matches the method’s corresponding defined parameter in data type, or is “promotable” to it.
2. Next, consider each argument of the method invocation, from left to right. The leftmost argument (and thus the first argument) is the implicit SELF parameter. For example, a method defined for type ADDRESS_T has an implicit first parameter of type ADDRESS_T. For each argument, eliminate all functions that are not the best match for that argument. The best match for a given argument is the first data type appearing in the precedence list corresponding to the argument data type for which there exists a function with a parameter of that data type. Length, precision, scale, and the FOR BIT DATA attribute are not considered in this comparison. For example, a DECIMAL(9,1) argument is considered an exact match for a DECIMAL(6,5) parameter, a DECFLOAT(34) argument is considered an exact match for a DECFLOAT(16) parameter, and a VARCHAR(19) argument is an exact match for a VARCHAR(6) parameter. The best match for a user-defined structured-type argument is itself; the next best match is its immediate supertype, and so on for each supertype of the argument. Note that only the static type (declared type) of the structured-type argument is considered, not the dynamic type (most specific type).
3. At most, one candidate method remains after Step 2. This is the method that is chosen.
4. If there are no candidate methods remaining after step 2, an error is returned (SQLSTATE 42884).

Example of method resolution

Following is an example of successful method resolution.

There are seven FOO methods for three structured types defined in a hierarchy of GOVERNOR as a subtype of EMPEROR as a subtype of HEADOFSTATE, registered with the following signatures:

```
CREATE METHOD FOO (CHAR(5), INT, DOUBLE) FOR HEADOFSTATE SPECIFIC FOO_1 ...
CREATE METHOD FOO (INT, INT, DOUBLE) FOR HEADOFSTATE SPECIFIC FOO_2 ...
CREATE METHOD FOO (INT, INT, DOUBLE, INT) FOR HEADOFSTATE SPECIFIC FOO_3 ...
CREATE METHOD FOO (INT, DOUBLE, DOUBLE) FOR EMPEROR SPECIFIC FOO_4 ...
CREATE METHOD FOO (INT, INT, DOUBLE) FOR EMPEROR SPECIFIC FOO_5 ...
CREATE METHOD FOO (SMALLINT, INT, DOUBLE) FOR EMPEROR SPECIFIC FOO_6 ...
CREATE METHOD FOO (INT, INT, DEC(7,2)) FOR GOVERNOR SPECIFIC FOO_7 ...
```

Methods

The method reference is as follows (where I1 and I2 are INTEGER columns, D is a DECIMAL column and E is an EMPEROR column):

```
SELECT E..FOO(I1, I2, D) ...
```

Following through the algorithm...

- FOO_7 is eliminated as a candidate, because the type GOVERNOR is a subtype (not a supertype) of EMPEROR.
- FOO_3 is eliminated as a candidate, because it has the wrong number of parameters.
- FOO_1 and FOO_6 are eliminated because, in both cases, the first argument (not the subject argument) cannot be promoted to the data type of the first parameter. Because there is more than one candidate remaining, the arguments are considered in order.
- For the subject argument, FOO_2 is a supertype, while FOO_4 and FOO_5 match the subject argument.
- For the first argument, the remaining methods, FOO_4 and FOO_5, are an exact match with the argument type. No methods can be eliminated from consideration; therefore the next argument must be examined.
- For this second argument, FOO_5 is an exact match, but FOO_4 is not, so it is eliminated from consideration. This leaves FOO_5 as the method chosen.

Method invocation

Once the method is selected, there are still possible reasons why the use of the method may not be permitted.

Each method is defined to return a result with a specific data type. If this result data type is not compatible with the context in which the method is invoked, an error will occur. For example, assume that the following methods named STEP are defined, each with a different data type as the result:

```
STEP(SMALLINT) FOR TYPEA RETURNS CHAR(5)  
STEP(DOUBLE) FOR TYPEA RETURNS INTEGER
```

and the following method reference (where S is a SMALLINT column and TA is a column of TYPEA):

```
SELECT 3 + TA..STEP(S) ...
```

then, because there is an exact match on argument type, the first STEP is chosen. An error occurs on the statement, because the result type is CHAR(5) instead of a numeric type, as required for an argument of the addition operator.

Starting from the method that has been chosen, the algorithm described in “Dynamic dispatch of methods” is used to build the set of dispatchable methods at compile time. Exactly which method is invoked is described in “Dynamic dispatch of methods”.

Note that when the selected method is a type preserving method:

- the static result type following function resolution is the same as the static type of the subject argument of the method invocation
- the dynamic result type when the method is invoked is the same as the dynamic type of the subject argument of the method invocation.

This may be a subtype of the result type specified in the type preserving method definition, which in turn may be a supertype of the dynamic type that is actually returned when the method is processed.

In cases where the arguments of the method invocation were not an exact match to the data types of the parameters of the selected method, the arguments are converted to the data type of the parameter at execution using the same rules as assignment to columns. This includes the case where precision, scale, or length differs between the argument and the parameter, but excludes the case where the dynamic type of the argument is a subtype of the parameter's static type.

Dynamic dispatch of methods

Methods provide the functionality and encapsulate the data of a type. A method is defined for a type and can always be associated with this type. One of the method's parameters is the implicit SELF parameter. The SELF parameter is of the type for which the method has been declared. The argument that is passed as the SELF argument when the method is invoked in a DML statement is called *subject*.

When a method is chosen using method resolution (see "Method resolution" on page 143), or a method has been specified in a DDL statement, this method is known as the "most specific applicable authorized method". If the subject is of a structured type, that method could have one or more overriding methods. DB2 must then determine which of these methods to invoke, based on the dynamic type (most specific type) of the subject at run time. This determination is called "determining the most specific dispatchable method". That process is described here.

1. Find the original method in the method hierarchy that the most specific applicable authorized method is part of. This is called the *root method*.
2. Create the set of dispatchable methods, which includes the following:
 - The most specific applicable authorized method.
 - Any method that overrides the most specific applicable authorized method, and which is defined for a type that is a subtype of the subject of this invocation.
3. Determine the most specific dispatchable method, as follows:
 - a. Start with an arbitrary method that is an element of the set of dispatchable methods and that is a method of the dynamic type of the subject, or of one of its supertypes. This is the initial most specific dispatchable method.
 - b. Iterate through the elements of the set of dispatchable methods. For each method: If the method is defined for one of the proper subtypes of the type for which the most specific dispatchable method is defined, and if it is defined for one of the supertypes of the most specific type of the subject, then repeat step 2 with this method as the most specific dispatchable method; otherwise, continue iterating.
4. Invoke the most specific dispatchable method.

Example:

Given are three types, "Person", "Employee", and "Manager". There is an original method "income", defined for "Person", which computes a person's income. A person is by default unemployed (a child, a retiree, and so on). Therefore, "income" for type "Person" always returns zero. For type "Employee" and for type

Methods

"Manager", different algorithms have to be applied to calculate the income. Hence, the method "income" for type "Person" is overridden in "Employee" and "Manager".

Create and populate a table as follows:

```
CREATE TABLE aTable (id integer, personColumn Person);
INSERT INTO aTable VALUES (0, Person()), (1, Employee()), (2, Manager());
```

List all persons who have a minimum income of \$40000:

```
SELECT id, person, name
FROM aTable
WHERE person..income() >= 40000;
```

The method "income" for type "Person" is chosen, using method resolution, to be the most specific applicable authorized method.

1. The root method is "income" for "Person" itself.
2. The second step of the algorithm above is carried out to construct the set of dispatchable methods:
 - The method "income" for type "Person" is included, because it is the most specific applicable authorized method.
 - The method "income" for type "Employee", and "income" for "Manager" is included, because both methods override the root method, and both "Employee" and "Manager" are subtypes of "Person".

Therefore, the set of dispatchable methods is: {"income" for "Person", "income" for "Employee", "income" for "Manager"}.

3. Determine the most specific dispatchable method:
 - For a subject whose most specific type is "Person":
 - a. Let the initial most specific dispatchable method be "income" for type "Person".
 - b. Because there is no other method in the set of dispatchable methods that is defined for a proper subtype of "Person" and for a supertype of the most specific type of the subject, "income" for type "Person" is the most specific dispatchable method.
 - For a subject whose most specific type is "Employee":
 - a. Let the initial most specific dispatchable method be "income" for type "Person".
 - b. Iterate through the set of dispatchable methods. Because method "income" for type "Employee" is defined for a proper subtype of "Person" and for a supertype of the most specific type of the subject (Note: A type is its own super- and subtype.), method "income" for type "Employee" is a better match for the most specific dispatchable method. Repeat this step with method "income" for type "Employee" as the most specific dispatchable method.
 - c. Because there is no other method in the set of dispatchable methods that is defined for a proper subtype of "Employee" and for a supertype of the most specific type of the subject, method "income" for type "Employee" is the most specific dispatchable method.
 - For a subject whose most specific type is "Manager":
 - a. Let the initial most specific dispatchable method be "income" for type "Person".
 - b. Iterate through the set of dispatchable methods. Because method "income" for type "Manager" is defined for a proper subtype of "Person"

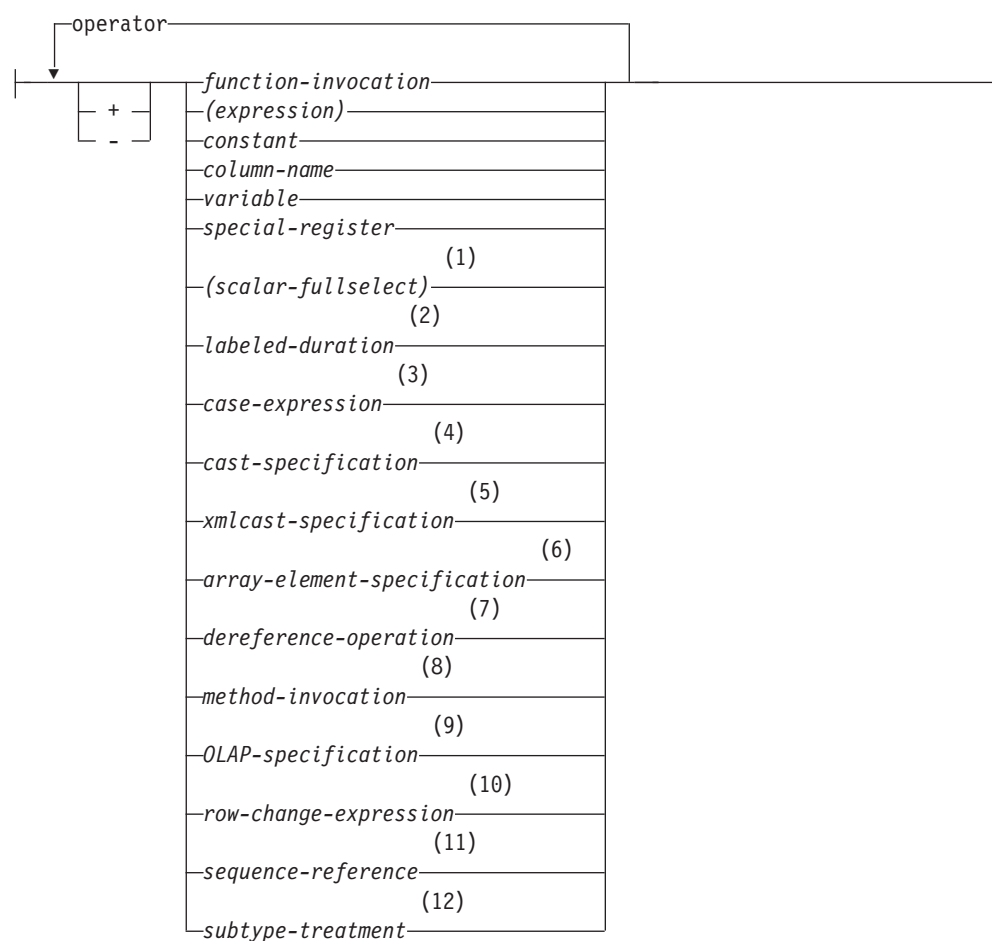
- and for a supertype of the most specific type of the subject (Note: A type is its own super- and subtype.), method "income" for type "Manager" is a better match for the most specific dispatchable method. Repeat this step with method "income" for type "Manager" as the most specific dispatchable method.
- c. Because there is no other method in the set of dispatchable methods that is defined for a proper subtype of "Manager" and for a supertype of the most specific type of the subject, method "income" for type "Manager" is the most specific dispatchable method.
4. Invoke the most specific dispatchable method.

Expressions

An expression specifies a value. It can be a simple value, consisting of only a constant or a column name, or it can be more complex. When repeatedly using similar complex expressions, an SQL function to encapsulate a common expression can be considered.

In a Unicode database, an expression that accepts a character or graphic string will accept any string types for which conversion is supported.

expression:



operator:

(13)

CONCAT
/
*
+
-

Notes:

- 1 See “Scalar fullselect” on page 158 for more information.
- 2 See “Durations” on page 158 for more information.
- 3 See “CASE expression” on page 163 for more information.
- 4 See “CAST specification” on page 165 for more information.
- 5 See “XMLCAST specification” on page 169 for more information.
- 6 See “ARRAY element specification” on page 170 for more information.
- 7 See “Dereference operation” on page 170 for more information.
- 8 See “Method invocation” on page 180 for more information.
- 9 See “OLAP specifications” on page 172 for more information.
- 10 See “ROW CHANGE expression” on page 185 for more information.
- 11 See “Sequence reference” on page 182 for more information.
- 12 See “Subtype treatment” on page 181 for more information.
- 13 || can be used as a synonym for CONCAT.

Expressions without operators

If no operators are used, the result of the expression is the specified value.

Examples:

```
SALARY:SALARY'SALARY'MAX(SALARY)
```

Expressions 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).

In a Unicode database, concatenation involving both character string operands and graphic string operands will first convert the character operands to graphic operands. Note that in a non-Unicode database, concatenation cannot involve both character and graphic operands.

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 15. 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
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.

Expressions

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 `FIRSTNAME` is Pierre and `LASTNAME` is Fermat, then the following:

```
FIRSTNAME 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` defined as `CHAR(10)`
- `COLB` defined as `VARCHAR(5)`

The parameter marker in the expression:

```
COLA CONCAT COLB CONCAT ?
```

is considered `VARCHAR(15)`, because `COLA CONCAT COLB` is evaluated first, giving a result that is the first operand of the second `CONCAT` operation.

User-defined types

A user-defined type cannot be used with the concatenation operator, even if it is a distinct type with a source data type that is a string type. To concatenate, create a function with the `CONCAT` operator as its source. For example, if there were distinct types `TITLE` and `TITLE_DESCRIPTION`, both of which had `VARCHAR(25)` data types, the following user-defined function, `ATTACH`, 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())
```

Expressions 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 can be applied to signed numeric types and datetime types (see “Datetime arithmetic in SQL” on page 159). For example, USER+2 is invalid. Sourced functions can be defined for arithmetic operations on distinct types with a source type that is a signed numeric type.

The prefix operator + (unary plus) does not change its operand. The prefix operator - (unary minus) reverses the sign of a nonzero non decimal floating-point operand. The prefix operator - (unary minus) reverses the sign of all decimal floating-point operands, including zero and special values; that is, signalling and non-signalling NaNs and plus and minus infinity. If the data type of A is small integer, 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 divide by zero or a numeric overflow occurs during the processing of a non-decimal floating-point expression, an error is returned (SQLSTATE 22003 or 22012). For decimal floating-point expressions, a warning is returned (SQLSTATEs 0168C, 0168D, 0168E, or 0168F) which depends on the nature of the arithmetic condition.

A database can be configured (using DFT_SQLMATHWARN set to yes) so that arithmetic errors return a null value for the non-decimal floating-point expression, the query returns a warning (SQLSTATE 01519 or 01564), and proceeds with processing the SQL statement.

For decimal floating-point expressions, DFT_SQLMATHWARN has no effect; arithmetic conditions return an appropriate value (possibly a decimal floating-point special value), the query returns a warning (SQLSTATEs 0168C, 0168D, 0168E, or 0168F), and proceeds with processing of the SQL statement. Special values returned include plus and minus infinity and not a number. Arithmetic expressions involving one or more decimal floating-point numbers never evaluate to a null value unless one or more of the arguments to the expression are null.

When arithmetic errors are treated as nulls, there are implications on the results of SQL statements. The following are some examples of these implications.

- An arithmetic error that occurs in the expression that is the argument of 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.

Expressions

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 that has been converted to a decimal number with precision p and scale 0; p is 19 for a big integer, 11 for a large integer, and 5 for a small integer.

Two decimal operands

If both operands are decimal, the operation is performed in decimal. The result of any decimal arithmetic operation is a decimal number with a precision and scale that are dependent on the operation and the precision and scale of the operands. If the operation is addition or subtraction and the operands do not have the same scale, the operation is performed with a temporary copy of one of the operands. The copy of the shorter operand is extended with trailing zeros so that its fractional part has the same number of digits as the longer operand.

The result of a decimal operation 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 $\min(31, \max(p-s, p'-s') + \max(s, s') + 1)$. 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 $31 - p + s'$. The scale must not be negative.

Note: The `min_dec_div_3` database configuration parameter alters the scale for decimal arithmetic operations involving division. If the parameter value is set to NO, the scale is calculated as $31 - p + s'$. If the parameter is set to YES, the scale is calculated as $\text{MAX}(3, 31 - p + s')$. This ensures that the result of decimal division always has a scale of at least 3 (precision is always 31).

Floating-point operands

If either operand of an arithmetic operator is floating-point, but not decimal floating-point, the operation is performed in floating-point. The operands are first converted to double-precision floating-point numbers, if necessary. Thus, if any element of an expression is a floating-point number, the result of the expression is a double-precision floating-point number.

An operation involving a floating-point number and an integer is performed with a temporary copy of the integer which has been converted to double-precision floating-point. An operation involving a floating-point number and a decimal number is performed with a temporary copy of the decimal number which has been converted to double-precision floating-point. The result of a floating-point operation must be within the range of floating-point numbers.

The order in which floating-point operands (or arguments to functions) are processed can slightly affect results because floating-point operands are approximate representations of real numbers. Since the order in which operands are processed may be implicitly modified by the optimizer (for example, the optimizer may decide what degree of parallelism to use and what access plan to use), an application that uses floating-point operands should not depend on the results being precisely the same each time an SQL statement is executed.

Decimal floating-point operands

If either operand of an arithmetic operator is decimal floating-point, the operation is performed in decimal floating-point.

Integer and decimal floating-point operands

If one operand is a small integer or large integer and the other is a `DECFLOAT(n)` number, the operation is performed in `DECFLOAT(n)` using a temporary copy of the integer that has been converted to a `DECFLOAT(n)` number. If one operand is a big integer, and the other is a decimal floating-point number, a temporary copy of the big integer is converted to a `DECFLOAT(34)` number. The rules for two-decimal floating-point operands then apply.

Decimal and decimal floating-point operands

If one operand is a decimal and the other is a decimal floating-point number, the operation is performed in decimal floating-point using a temporary copy of the decimal number that has been converted to a

decimal floating-point number based on the precision of the decimal number. If the decimal number has a precision less than 17, the decimal number is converted to a DECFLOAT(16) number; otherwise, the decimal number is converted to a DECFLOAT(34) number. The rules for two-decimal floating-point operands then apply.

Floating-point and decimal floating-point operands

If one operand is a floating-point number (REAL or DOUBLE) and the other is a DECFLOAT(n) number, the operation is performed in decimal floating-point using a temporary copy of the floating-point number that has been converted to a DECFLOAT(n) number.

Two decimal floating-point operands

If both operands are DECFLOAT(n), the operation is performed in DECFLOAT(n). If one operand is DECFLOAT(16) and the other is DECFLOAT(34), the operation is performed in DECFLOAT(34).

General arithmetic operation rules for decimal floating-point

The following general rules apply to all arithmetic operations on the decimal floating-point data type:

- Every operation on finite numbers is carried out as though an exact mathematical result is computed, using integer arithmetic on the coefficient, where possible.

If the coefficient of the theoretical exact result has no more than the number of digits that reflect its precision (16 or 34), it is used for the result without change (unless there is an underflow or overflow condition). If the coefficient has more than the number of digits that reflect its precision, it is rounded to exactly the number of digits that reflect its precision (16 or 34), and the exponent is increased by the number of digits that are removed.

The CURRENT DECFLOAT ROUNDING MODE special register determines the rounding mode.

If the value of the adjusted exponent of the result is less than E_{\min} , the calculated coefficient and exponent form the result, unless the value of the exponent is less than E_{tiny} , in which case the exponent is set to E_{tiny} , the coefficient is rounded (possibly to zero) to match the adjustment of the exponent, and the sign remains unchanged. If this rounding gives an inexact result, an underflow exception condition is returned.

If the value of the adjusted exponent of the result is larger than E_{\max} , an overflow exception condition is returned. In this case, the result is defined as an overflow exception condition and might be infinite. It has the same sign as the theoretical result.

- Arithmetic that uses the special value infinity follows the usual rules, where negative infinity is less than every finite number and positive infinity is greater than every finite number. Under these rules, an infinite result is always exact. Certain uses of infinity return an invalid operation condition. The following list shows the operations that can cause an invalid operation condition. The result of such an operation is NaN when one of the operands is infinity but the other operand is not NaN or sNaN.
 - Add +infinity to -infinity during an addition or subtraction operation
 - Multiply 0 by +infinity or -infinity
 - Divide either +infinity or -infinity by either +infinity or -infinity
 - Either argument of the QUANTIZE function is +infinity or -infinity
 - The second argument of the POWER function is +infinity or -infinity

- Signaling NaNs used as operands to arithmetic operations

The following rules apply to arithmetic operations and the NaN value:

- The result of any arithmetic operation that has a NaN (quiet or signalling) operand is NaN. The sign of the result is copied from the first operand that is a signalling NaN; if neither operand is signalling, the sign is copied from the first operand that is a NaN. Whenever a result is a NaN, the sign of the result depends only on the copied operand.
- The sign of the result of a multiplication or division operation is negative only if the operands have different signs and neither is a NaN.
- The sign of the result of an addition or subtraction operation is negative only if the result is less than zero and neither operand is a NaN, except for the following cases where the result is a negative 0:
 - A result is rounded to zero, and the value, before rounding, had a negative sign
 - -0 is added to 0
 - 0 is subtracted from -0
 - Operands with opposite signs are added, or operands with the same sign are subtracted; the result has a coefficient of 0, and the rounding mode is ROUND_FLOOR
 - Operands are multiplied or divided, the result has a coefficient of 0, and the signs of the operands are different
 - The first argument of the POWER function is -0, and the second argument is a positive odd number
 - The argument of the CEIL, FLOOR, or SQRT function is -0
 - The first argument of the ROUND or TRUNCATE function is -0

The following examples show special decimal floating-point values as operands:

```

INFINITY + 1           = INFINITY
INFINITY + INFINITY   = INFINITY
INFINITY + -INFINITY  = NAN           -- warning
NAN + 1               = NAN
NAN + INFINITY        = NAN
1 - INFINITY          = -INFINITY
INFINITY - INFINITY   = NAN           -- warning
-INFINITY - -INFINITY = NAN           -- warning
-0.0 - 0.0E1          = -0.0
-1.0 * 0.0E1          = -0.0
1.0E1 / 0             = INFINITY      -- warning
-1.0E5 / 0.0          = -INFINITY     -- warning
1.0E5 / -0            = -INFINITY     -- warning
INFINITY / -INFINITY  = NAN           -- warning
INFINITY / 0          = INFINITY
-INFINITY / 0         = -INFINITY
-INFINITY / -0        = INFINITY

```

User-defined types as operands

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)

```

Expressions

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)
```

Precedence of operations

Expressions within parentheses and dereference operations are evaluated first from left to right. (Parentheses are also used in subselect statements, search conditions, and functions. However, they should not be used to arbitrarily group sections within SQL statements.) When the order of evaluation is not specified by parentheses, 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.

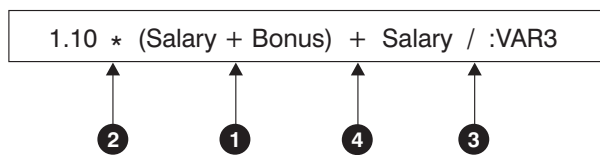


Figure 12. Precedence of Operations

Scalar fullselect

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. The authorization required for a scalar fullselect is the same as that required for an SQL query.

Datetime operations and durations

Datetime values can be incremented, decremented, and subtracted. These operations can involve decimal numbers called durations. The following sections describe duration types and detail the rules for datetime arithmetic.

Durations

A *duration* is a number representing an interval of time. There are four types of durations.

labeled-duration:

<i>function</i>	YEAR
<i>(expression)</i>	YEARS
<i>constant</i>	MONTH
<i>column-name</i>	MONTHS
<i>global-variable</i>	DAY
<i>host-variable</i>	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. (The singular form of these keywords is also acceptable: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and MICROSECOND.) The number specified is converted as if it were assigned to a DECIMAL(15,0) number. A labeled duration can only be used as an operand of an arithmetic operator in which the other operand is a value of data type DATE, TIME, or TIMESTAMP. Thus, the expression HIREDATE + 2 MONTHS + 14 DAYS is valid, whereas the expression HIREDATE + (2 MONTHS + 14 DAYS) is not. In both of these expressions, the labeled durations are 2 MONTHS and 14 DAYS.

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. (The period in the format indicates a DECIMAL data type.) The result of subtracting one date value from another, as in the expression HIREDATE - BRTHDATE, is a date duration.

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. (The period in the format indicates a DECIMAL data type.) The result of subtracting one time value from another is a time 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.nnnnnn*, where *yyyy*, *mm*, *dd*, *hh*, *mm*, *ss*, and *nnnnnn* 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.

Datetime operations and durations

- If one operand is a time, the other operand must be a time duration or a labeled duration of HOURS, MINUTES, or SECONDS.
- If one operand is a timestamp, the other operand must be a duration. Any type of duration is valid.
- Neither operand of the addition operator can be a parameter marker.

The rules for the use of the subtraction operator on datetime values are not the same as those for addition because a datetime value cannot be subtracted from a duration, and because the operation of subtracting two datetime values is not the same as the operation of subtracting a duration from a datetime value. The specific rules governing the use of the subtraction operator with datetime values follow.

- If the first operand is a 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.

- 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 $\text{result} = \text{DATE1} - \text{DATE2}$.

```
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 $\text{DATE}('3/15/2000') - '12/31/1999'$ is 00000215. (or, a duration of 0 years, 2 months, and 15 days).

- The result of adding a duration to a date, or of subtracting a duration from a date, is itself a date. (For the purposes of this operation, a month denotes the

equivalent of a calendar page. Adding months to a date, then, is like turning the pages of a calendar, starting with the page on which the date appears.) The result must fall between the dates January 1, 0001 and December 31, 9999 inclusive.

If a duration of years is added or subtracted, only the year portion of the date is affected. The month is unchanged, as is the day unless the result would be February 29 of a non-leap-year. In this case, the day is changed to 28, and a warning indicator in the SQLCA is set to indicate the adjustment.

Similarly, if a duration of months is added or subtracted, only months and, if necessary, years are affected. The day portion of the date is unchanged unless the result would be invalid (September 31, for example). In this case, the day is set to the last day of the month, and a warning indicator in the SQLCA is set to indicate the adjustment.

Adding or subtracting a duration of days will, of course, affect the day portion of the date, and potentially the month and year.

Date durations, whether positive or negative, may also be added to and subtracted from dates. As with labeled durations, the result is a valid date, and a warning indicator is set in the SQLCA whenever an end-of-month adjustment is necessary.

When a positive date duration is added to a date, or a negative date duration is subtracted from a date, the date is incremented by the specified number of years, months, and days, in that order. Thus, $DATE1 + X$, where X is a positive DECIMAL(8,0) number, is equivalent to the expression:

$$DATE1 + YEAR(X) YEARS + MONTH(X) MONTHS + DAY(X) DAYS.$$

When a positive date duration is subtracted from a date, or a negative date duration is added to a date, the date is decremented by the specified number of days, months, and years, in that order. Thus, $DATE1 - X$, where X is a positive DECIMAL(8,0) number, is equivalent to the expression:

$$DATE1 - DAY(X) DAYS - MONTH(X) MONTHS - YEAR(X) YEARS.$$

When adding durations to dates, adding one month to a given date gives the same date one month later unless that date does not exist in the later month. In that case, the date is set to that of the last day of the later month. For example, January 28 plus one month gives February 28; and one month added to January 29, 30, or 31 results in either February 28 or, for a leap year, February 29.

Note: If one or more months is added to a given date and then the same number of months is subtracted from the result, the final date is not necessarily the same as the original date.

Time arithmetic

Times can be subtracted, incremented, or decremented.

- The result of subtracting one time (TIME2) from another (TIME1) is a time duration that specifies the number of hours, minutes, and seconds between the two times. The data type of the result is DECIMAL(6,0).

If TIME1 is greater than or equal to TIME2, TIME2 is subtracted from TIME1.

If TIME1 is less than TIME2, however, TIME1 is subtracted from TIME2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation $result = TIME1 - TIME2$.

```
If SECOND(TIME2) <= SECOND(TIME1)
then SECOND(RESULT) = SECOND(TIME1) - SECOND(TIME2).
```

Datetime operations and durations

```
If SECOND(TIME2) > SECOND(TIME1)
then SECOND(RESULT) = 60 + SECOND(TIME1) - SECOND(TIME2).
MINUTE(TIME2) is then incremented by 1.
If MINUTE(TIME2) <= MINUTE(TIME1)
then MINUTE(RESULT) = MINUTE(TIME1) - MINUTE(TIME2).
If MINUTE(TIME1) > MINUTE(TIME1)
then MINUTE(RESULT) = 60 + MINUTE(TIME1) - MINUTE(TIME2).
HOUR(TIME2) is then incremented by 1.
HOUR(RESULT) = HOUR(TIME1) - HOUR(TIME2).
```

For example, the result of `TIME('11:02:26') - '00:32:56'` is 102930. (a duration of 10 hours, 29 minutes, and 30 seconds).

- The result of adding a duration to a time, or of subtracting a duration from a time, is itself a time. Any overflow or underflow of hours is discarded, thereby ensuring that the result is always a time. If a duration of hours is added or subtracted, only the hours portion of the time is affected. The minutes and seconds are unchanged.

Similarly, if a duration of minutes is added or subtracted, only minutes and, if necessary, hours are affected. The seconds portion of the time is unchanged.

Adding or subtracting a duration of seconds will, of course, affect the seconds portion of the time, and potentially the minutes and hours.

Time durations, whether positive or negative, also can be added to and subtracted from times. The result is a time that has been incremented or decremented by the specified number of hours, minutes, and seconds, in that order. `TIME1 + X`, where "X" is a `DECIMAL(6,0)` number, is equivalent to the expression:

```
TIME1 + HOUR(X) HOURS + MINUTE(X) MINUTES + SECOND(X) SECONDS
```

Note: Although the time `'24:00:00'` is accepted as a valid time, it is never returned as the result of time addition or subtraction, even if the duration operand is zero (for example, `time('24:00:00')±0 seconds = '00:00:00'`).

Timestamp arithmetic

Timestamps can be subtracted, incremented, or decremented.

- The result of subtracting one timestamp (TS2) from another (TS1) is a timestamp duration that specifies the number of years, months, days, hours, minutes, seconds, and 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).
```

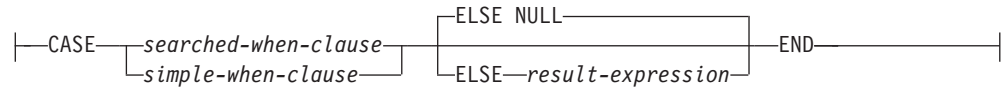
```
If HOUR(TS2) > HOUR(TS1)
then HOUR(RESULT) = 24 + HOUR(TS1) - HOUR(TS2)
and DAY(TS2) is incremented by 1.
```

The date part of the timestamps is subtracted as specified in the rules for subtracting dates.

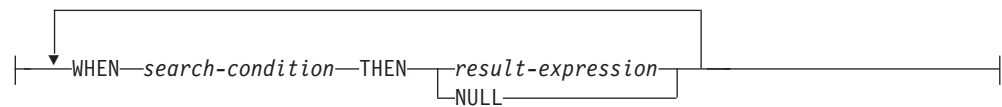
- The result of adding a duration to a timestamp, or of subtracting a duration from a timestamp is itself a timestamp. Date and time arithmetic is performed as previously defined, except that an overflow or underflow of hours is carried into the date part of the result, which must be within the range of valid dates. Microseconds overflow into seconds.

CASE expression

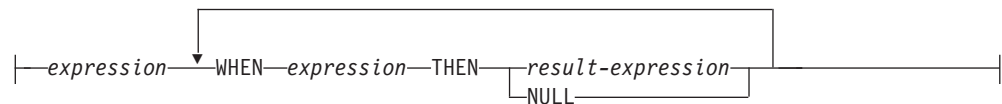
case-expression:



searched-when-clause:



simple-when-clause:



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).

CASE expression

A *result-expression* is an *expression* following the THEN or ELSE keywords. There must be at least one *result-expression* in the CASE expression (NULL cannot be specified for every case) (SQLSTATE 42625). All result expressions must have compatible data types (SQLSTATE 42804).

Examples

- If the first character of a department number is a division in the organization, then a CASE expression can be used to list the full name of the division to which each employee belongs:

```
SELECT EMPNO, LASTNAME,  
CASE SUBSTR(WORKDEPT,1,1)  
  WHEN 'A' THEN 'Administration'  
  WHEN 'B' THEN 'Human Resources'  
  WHEN 'C' THEN 'Accounting'  
  WHEN 'D' THEN 'Design'  
  WHEN 'E' THEN 'Operations'  
END  
FROM EMPLOYEE;
```

- The number of years of education are used in the EMPLOYEE table to give the education level. A CASE expression can be used to group these and to show the level of education.

```
SELECT EMPNO, FIRSTNAME, MIDINIT, LASTNAME,  
CASE  
  WHEN EDLEVEL < 15 THEN 'SECONDARY'  
  WHEN EDLEVEL < 19 THEN 'COLLEGE'  
  ELSE 'POST GRADUATE'  
END  
FROM EMPLOYEE
```

- Another interesting example of CASE statement usage is in protecting from division by 0 errors. For example, the following code finds the employees who earn more than 25% of their income from commission, but who are not fully paid on commission:

```
SELECT EMPNO, WORKDEPT, SALARY+COMM FROM EMPLOYEE  
WHERE (CASE WHEN SALARY=0 THEN NULL  
  ELSE COMM/SALARY  
END) > 0.25;
```

- The following CASE expressions are the same:

```
SELECT LASTNAME,  
CASE  
  WHEN LASTNAME = 'Haas' THEN 'President'  
  ...  
  
SELECT LASTNAME,  
CASE LASTNAME  
  WHEN 'Haas' THEN 'President'  
  ...
```

There are two scalar functions, NULLIF and COALESCE, that are specialized to handle a subset of the functionality provided by CASE. Table 16 shows the equivalent expressions using CASE or these functions.

Table 16. Equivalent CASE Expressions

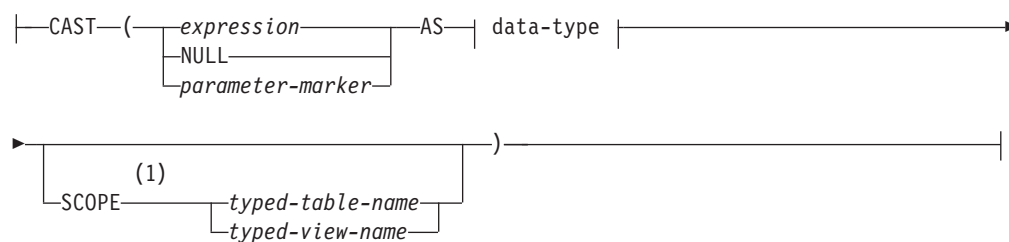
Expression	Equivalent Expression
CASE WHEN e1=e2 THEN NULL ELSE e1 END	NULLIF(e1,e2)

Table 16. Equivalent CASE Expressions (continued)

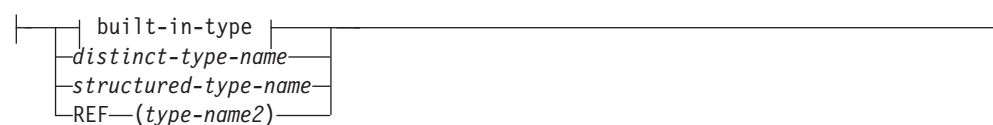
Expression	Equivalent Expression
CASE WHEN e1 IS NOT NULL THEN e1 ELSE e2 END	COALESCE(e1,e2)
CASE WHEN e1 IS NOT NULL THEN e1 ELSE COALESCE(e2,...,eN) END	COALESCE(e1,e2,...,eN)
CASE WHEN c1=var1 OR (c1 IS NULL AND var1 IS NULL) THEN 'a' WHEN c1=var2 OR (c1 IS NULL AND var2 IS NULL) THEN 'b' ELSE NULL END	DECODE(c1,var1, 'a', var2, 'b')

CAST specification

cast-specification:

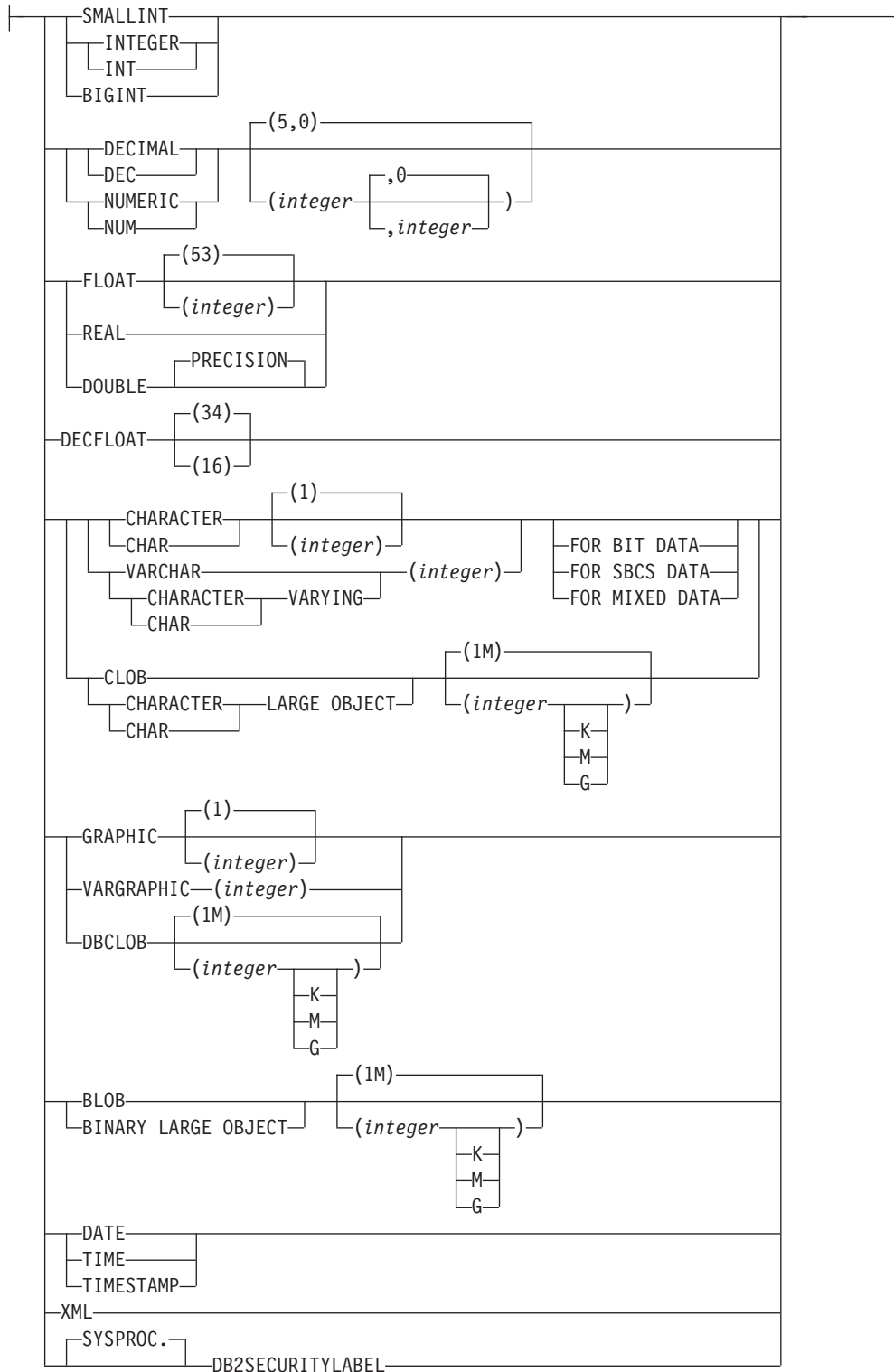


data-type:



built-in-type:

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*. If the cast is not supported, an error is returned (SQLSTATE 42846).

expression

If the cast operand is an expression (other than parameter marker or NULL), the result is the argument value converted to the specified target *data-type*.

When casting character strings (other than CLOBs) to a character string with a different length, a warning (SQLSTATE 01004) is returned if truncation of other than trailing blanks occurs. When casting graphic character strings (other than DBCLOBs) to a graphic character string with a different length, a warning (SQLSTATE 01004) is returned if truncation of other than trailing blanks occurs. For BLOB, CLOB and DBCLOB cast operands, the warning is issued if any characters are truncated.

When casting an array, the target data type must be a user-defined array data type (SQLSTATE 42821). The data type of the elements of the array must be the same as the data type of the elements of the target array data type (SQLSTATE 42846). The cardinality of the array must be less than or equal to the maximum cardinality of the target array data type (SQLSTATE 2202F).

NULL

If the cast operand is the keyword NULL, the result is a null value that has the specified *data-type*.

parameter-marker

A parameter marker (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 perform data type resolution. A data type that has associated attributes, such as length or precision and scale, should include these attributes when specifying *data-type*. (CHAR defaults to a length of 1, DECIMAL defaults to a precision of 5 and a scale of 0, and DECFLOAT defaults to a precision of 34 if not specified.) The FOR SBCS DATA clause or the FOR MIXED DATA clause (only one is supported depending on whether or not the database supports the graphic data type) can be used to cast a FOR BIT DATA string to the database code page. Restrictions on the supported data types are based on the specified cast operand.

- For a cast operand that is an *expression*, the supported target data types depend on the data type of the cast operand (source data type).
- 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 distinct type, the application using the parameter marker will use the source data type of the user-defined distinct type. If the data type is a user-defined structured type, the application using the parameter marker will use the input parameter type of the TO SQL transform function for the user-defined structured type.

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.

CAST specification

typed-table-name

The name of a typed table. The table must already exist (SQLSTATE 42704). The cast must be to *data-type* REF(*S*), where *S* is the type of *typed-table-name* (SQLSTATE 428DM).

typed-view-name

The name of a typed view. The view must exist or have the same name as the view being created that includes the cast as part of the view definition (SQLSTATE 42704). The cast must be to *data-type* REF(*S*), where *S* is the type of *typed-view-name* (SQLSTATE 428DM).

When numeric data is cast to character data, the result data type is a fixed-length character string. When character data is cast to numeric data, the result data type depends on the type of number specified. For example, if cast to integer, it becomes a large integer.

Examples

- An application is only interested in the integer portion of the SALARY (defined as decimal(9,2)) from the EMPLOYEE table. The following query, including the employee number and the integer value of SALARY, could be prepared.

```
SELECT EMPNO, CAST(SALARY AS INTEGER) FROM EMPLOYEE
```

- Assume the existence of a distinct type called T_AGE that is defined on SMALLINT and used to create column AGE in PERSONNEL table. Also assume the existence of a distinct type called R_YEAR that is defined on INTEGER and used to create column RETIRE_YEAR in PERSONNEL table. The following update statement could be prepared.

```
UPDATE PERSONNEL SET RETIRE_YEAR =?  
WHERE AGE = CAST( ? AS T_AGE)
```

The first parameter is an untyped parameter marker that would have a data type of R_YEAR, although the application will use an integer for this parameter marker. This does not require the explicit CAST specification because it is an assignment.

The second parameter marker is a typed parameter marker that is cast as a distinct type T_AGE. This satisfies the requirement that the comparison must be performed with compatible data types. The application will use the source data type (which is SMALLINT) for processing this parameter marker.

Successful processing of this statement assumes that the SQL path includes the schema name of the schema (or schemas) where the two distinct types are defined.

- An application supplies a value that is a series of bits, for example an audio stream, and it should not undergo code page conversion before being used in an SQL statement. The application could use the following CAST:

```
CAST( ? AS VARCHAR(10000) FOR BIT DATA)
```

- Assume that an array type and a table have been created as follows:

```
CREATE TYPE PHONELIST AS DECIMAL(10, 0) ARRAY[5]
```

```
CREATE TABLE EMP_PHONES  
(ID INTEGER,  
PHONENUMBER DECIMAL(10,0) )
```

The following procedure returns an array with the phone numbers for the employee with ID 1775. If there are more than five phone numbers for this employee, an error is returned (SQLSTATE 2202F).


```

CREATE PROCEDURE GET_PHONES(OUT EPHONES PHONELIST)
BEGIN
  SELECT CAST(ARRAY_AGG(PHONENUMBER) AS PHONELIST)
  INTO EPHONES
  FROM EMP_PHONES
  WHERE ID = 1775;
END

```

XMLCAST specification

xmlcast-specification:

```

XMLCAST—(—expression—AS—data-type—)
          |—NULL—|
          |—parameter-marker—|

```

The XMLCAST specification returns the cast operand (the first operand) cast to the type specified by the data type. XMLCAST supports casts involving XML values, including conversions between non-XML data types and the XML data type. If the cast is not supported, an error is returned (SQLSTATE 22003).

expression

If the cast operand is an expression (other than a parameter marker or NULL), the result is the argument value converted to the specified target data type. The expression or the target data type must be the XML data type (SQLSTATE 42846).

NULL

If the cast operand is the keyword NULL, the target data type must be the XML data type (SQLSTATE 42846). The result is a null XML value.

parameter-marker

If the cast operand is a parameter marker, the target data type must be XML (SQLSTATE 42846). A parameter marker (specified as a question mark character) is normally considered to be an expression, but is documented separately in this case because it has special meaning. If the cast operand is a parameter marker, the specified data type is considered to be a promise that the replacement will be assignable to the specified (XML) data type (using store assignment). Such a parameter marker is considered to be a typed parameter marker, which is treated like any other typed value for the purpose of function resolution, a describe operation on a select list, or column assignment.

data-type

The name of an existing SQL data type. If the name is not qualified, the SQL path is used to perform data type resolution. If a data type has associated attributes, such as length or precision and scale, these attributes should be included when specifying a value for *data-type*. CHAR defaults to a length of 1, and DECIMAL defaults to a precision of 5 and a scale of 0 if not specified. Restrictions on the supported data types are based on the specified cast operand.

- For a cast operand that is an expression, the supported target data types depend on the data type of the cast operand (source data type).
- For a cast operand that is the keyword NULL, the target data type must be XML.
- For a cast operand that is a parameter marker, the target data type must be XML.

XMLCAST specification

Note:

1. **Support in non-Unicode databases:** When XMLCAST is used to convert an XML value to an SQL data type, code page conversion is performed. The encoding of the cast expression is converted from UTF-8 to the database code page. Characters in the original expression that are not present in the database code page are replaced by substitution characters as a result of this conversion.
2. **Support in multiple database partition databases:** The XMLCAST specification is supported only in a database with a single database partition (SQLSTATE 42997).

Examples

- Create a null XML value.

```
XMLCAST(NULL AS XML)
```

- Convert a value extracted from an XMLQUERY expression into an INTEGER:

```
XMLCAST(XMLQUERY('$m/PRODUCT/QUANTITY'  
PASSING BY REF xmlcol AS "m" RETURNING SEQUENCE) AS INTEGER)
```

- Convert a value extracted from an XMLQUERY expression into a varying-length character string:

```
XMLCAST(XMLQUERY('$m/PRODUCT/ADD-TIMESTAMP'  
PASSING BY REF xmlcol AS "m" RETURNING SEQUENCE) AS VARCHAR(30))
```

- Convert a value extracted from an SQL scalar subquery into an XML value.

```
XMLCAST((SELECT quantity FROM product AS p  
WHERE p.id = 1077) AS XML)
```

ARRAY element specification

array-element-specification:

```
|-----array-variable-----|-----[expression]-----|  
|-----CAST(-----parameter-marker-----AS-----data-type-----)|
```

The ARRAY element specification returns the element from an array specified by *expression*. If any argument to *expression* is null, the null value is returned.

array-variable

Specifies a variable or parameter of type ARRAY in an SQL procedure (SQLSTATE 428H0).

CAST (*parameter-marker* AS *data-type*)

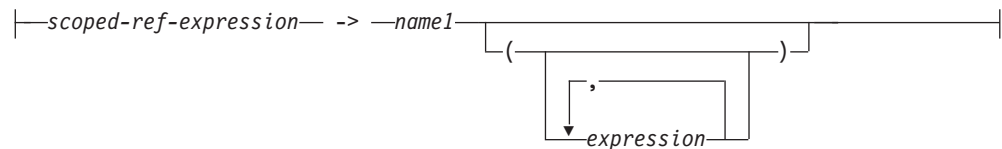
A parameter marker (specified as a question mark character) is normally considered to be an expression, but in this case it must explicitly be cast to a user-defined array data type.

[*expression*]

Specifies the subindex of the element that is to be extracted from the array. The subindex must be of an exact numeric type with scale zero (SQLSTATE 428H1); its value must be between 1 and the cardinality of the array (SQLSTATE 2202E).

Dereference operation

dereference-operation:



The scope of the scoped reference expression is a table or view called the *target* table or view. The scoped reference expression identifies a *target row*. The *target row* is the row in the target table or view (or in one of its subtables or subviews) whose object identifier (OID) column value matches the reference expression. The dereference operation can be used to access a column of the target row, or to invoke a method, using the target row as the subject of the method. The result of a dereference operation can always be null. The dereference operation takes precedence over all other operators.

scoped-ref-expression

An expression that is a reference type that has a scope (SQLSTATE 428DT). If the expression is a host variable, parameter marker or other unscoped reference type value, a CAST specification with a SCOPE clause is required to give the reference a scope.

name1

Specifies an unqualified identifier.

If no parentheses follow *name1*, and *name1* matches the name of an attribute of the target type, then the value of the dereference operation is the value of the named column in the target row. In this case, the data type of the column (made nullable) determines the result type of the dereference operation. If no target row exists whose object identifier matches the reference expression, then the result of the dereference operation is null. If the dereference operation is used in a select list and is not included as part of an expression, *name1* becomes the result column name.

If parentheses follow *name1*, or if *name1* does not match the name of an attribute of the target type, then the dereference operation is treated as a method invocation. The name of the invoked method is *name1*. The subject of the method is the target row, considered as an instance of its structured type. If no target row exists whose object identifier matches the reference expression, the subject of the method is a null value of the target type. The expressions inside parentheses, if any, provide the remaining parameters of the method invocation. The normal process is used for resolution of the method invocation. The result type of the selected method (made nullable) determines the result type of the dereference operation.

The authorization ID of the statement that uses a dereference operation must have SELECT privilege on the target table of the *scoped-ref-expression* (SQLSTATE 42501).

A dereference operation can never modify values in the database. If a dereference operation is used to invoke a mutator method, the mutator method modifies a copy of the target row and returns the copy, leaving the database unchanged.

Examples

- Assume the existence of an EMPLOYEE table that contains a column called DEPTREF which is a reference type scoped to a typed table based on a type that includes the attribute DEPTNAME. The values of DEPTREF in the table EMPLOYEE should correspond to the OID column values in the target table of DEPTREF column.

Dereference operation

```
SELECT EMPNO, DEPTREF->DEPTNAME  
FROM EMPLOYEE
```

- Using the same tables as in the previous example, use a dereference operation to invoke a method named BUDGET, with the target row as subject parameter, and '1997' as an additional parameter.

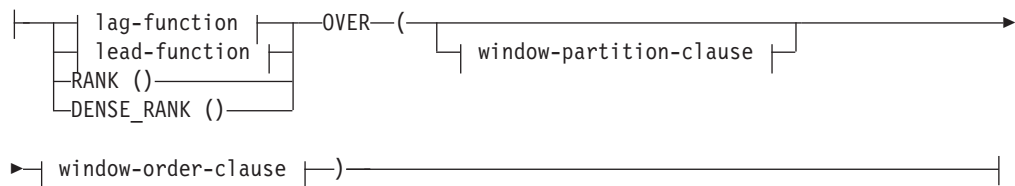
```
SELECT EMPNO, DEPTREF->BUDGET('1997') AS DEPTBUDGET97  
FROM EMPLOYEE
```

OLAP specifications

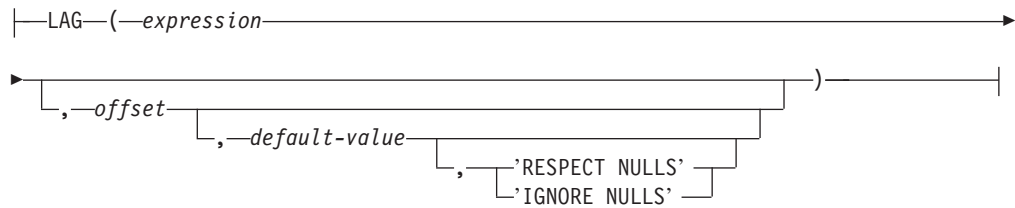
OLAP-specification:



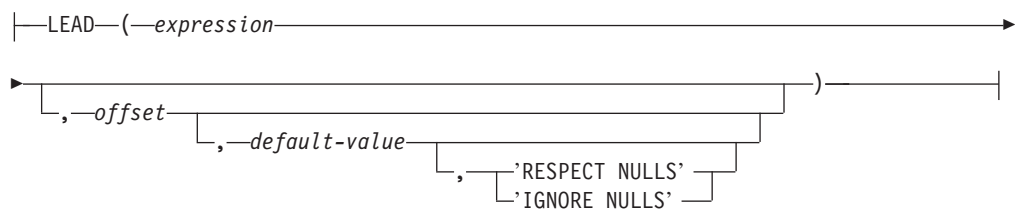
ordered-OLAP-specification:



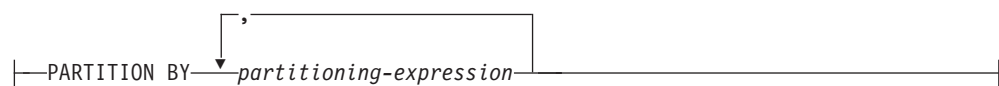
lag-function:



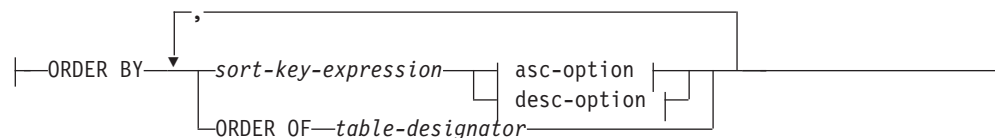
lead-function:



window-partition-clause:



window-order-clause:



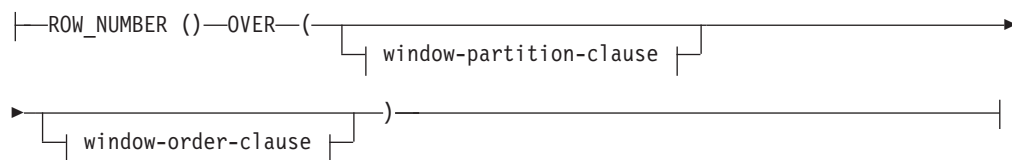
asc-option:



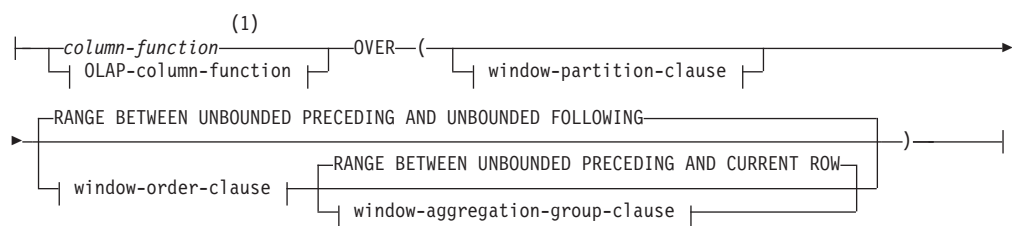
desc-option:



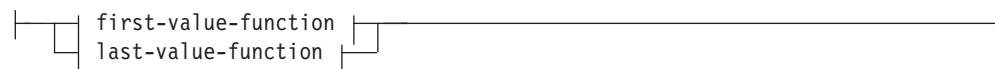
numbering-specification:



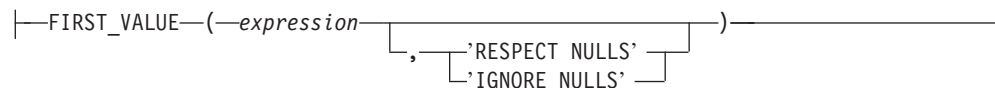
aggregation-specification:



OLAP-column-function:



first-value-function:



OLAP specifications

last-value-function:

| LAST_VALUE (*expression* | , | 'RESPECT NULLS' | 'IGNORE NULLS' |) |

window-aggregation-group-clause:

| ROWS | RANGE | group-start | group-between | group-end |

group-start:

| UNBOUNDED PRECEDING | *unsigned-constant* PRECEDING | CURRENT ROW |

group-between:

| BETWEEN | group-bound1 | AND | group-bound2 |

group-bound1:

| UNBOUNDED PRECEDING | *unsigned-constant* PRECEDING | *unsigned-constant* FOLLOWING | CURRENT ROW |

group-bound2:

| UNBOUNDED FOLLOWING | *unsigned-constant* PRECEDING | *unsigned-constant* FOLLOWING | CURRENT ROW |

group-end:

| UNBOUNDED FOLLOWING | *unsigned-constant* FOLLOWING |

Notes:

- 1 ARRAY_AGG is not supported as a column function in *aggregation-specification* (SQLSTATE 42887).

On-Line Analytical Processing (OLAP) functions provide the ability to return ranking, row numbering and existing column function information as a scalar value in a query result. An OLAP function can be included in expressions in a select-list or the ORDER BY clause of a select-statement (SQLSTATE 42903). An OLAP function cannot be used within an argument to an XMLQUERY or XMLEXISTS expression (SQLSTATE 42903). An OLAP function cannot be used as

an argument of a column function (SQLSTATE 42607). The query result to which the OLAP function is applied is the result table of the innermost subselect that includes the OLAP function.

When specifying an OLAP function, a window is specified that defines the rows over which the function is applied, and in what order. When used with a column function, the applicable rows can be further refined, relative to the current row, as either a range or a number of rows preceding and following the current row. For example, within a partition by month, an average can be calculated over the previous three month period.

The ranking function computes the ordinal rank of a row within the window. Rows that are not distinct with respect to the ordering within their window are assigned the same rank. The results of ranking may be defined with or without gaps in the numbers resulting from duplicate values.

If RANK is specified, the rank of a row is defined as 1 plus the number of rows that strictly precede the row. Thus, if two or more rows are not distinct with respect to the ordering, then there will be one or more gaps in the sequential rank numbering.

If DENSE_RANK (or DENSERANK) is specified, the rank of a row is defined as 1 plus the number of preceding rows that are distinct with respect to the ordering. Therefore, there will be no gaps in the sequential rank numbering.

The ROW_NUMBER (or ROWNUMBER) function computes the sequential row number of the row within the window defined by the ordering, starting with 1 for the first row. If the ORDER BY clause is not specified in the window, the row numbers are assigned to the rows in arbitrary order, as returned by the subselect (not according to any ORDER BY clause in the select-statement).

If the FETCH FIRST *n* ROWS ONLY clause is used along with the ROW_NUMBER function, the row numbers might not be displayed in order. The FETCH FIRST clause is applied after the result set (including any ROW_NUMBER assignments) is generated; therefore, if the row number order is not the same as the order of the result set, some assigned numbers might be missing from the sequence.

The data type of the result of RANK, DENSE_RANK or ROW_NUMBER is BIGINT. The result cannot be null.

The LAG function returns the expression value for the row at *offset* rows before the current row. The *offset* must be a positive integer (SQLSTATE 42815). An *offset* value of 0 means the current row. If a window-partition-clause is specified, *offset* means *offset* rows before the current row and within the current partition. If *offset* is not specified, the value 1 is used. If *default-value* (which can be an expression) is specified, it will be returned if the offset goes beyond the scope of the current partition. Otherwise, the null value is returned. If 'IGNORE NULLS' is specified, all rows where the expression value for the row is the null value are not considered in the calculation. If 'IGNORE NULLS' is specified and all rows are null, *default-value* (or the null value if *default-value* was not specified) is returned.

The LEAD function returns the expression value for the row at *offset* rows after the current row. The *offset* must be a positive integer (SQLSTATE 42815). An *offset* value of 0 means the current row. If a window-partition-clause is specified, *offset* means *offset* rows after the current row and within the current partition. If *offset* is not specified, the value 1 is used. If *default-value* (which can be an expression) is

OLAP specifications

specified, it will be returned if the offset goes beyond the scope of the current partition. Otherwise, the null value is returned. If 'IGNORE NULLS' is specified, all rows where the expression value for the row is the null value are not considered in the calculation. If 'IGNORE NULLS' is specified and all rows are null, *default-value* (or the null value if *default-value* was not specified) is returned.

The FIRST_VALUE function returns the expression value for the first row in an OLAP window. If 'IGNORE NULLS' is specified, all rows where the expression value for the row is the null value are not considered in the calculation. If 'IGNORE NULLS' is specified and all values in the OLAP window are null, FIRST_VALUE returns the null value.

The LAST_VALUE function returns the expression value for the last row in an OLAP window. If 'IGNORE NULLS' is specified, all rows where the expression value for the row is the null value are not considered in the calculation. If 'IGNORE NULLS' is specified and all values in the OLAP window are null, LAST_VALUE returns the null value.

The data type of the result of FIRST_VALUE, LAG, LAST_VALUE, and LEAD is the data type of the expression. The result can be null.

PARTITION BY (*partitioning-expression*,...)

Defines the partition within which the function is applied. A *partitioning-expression* is an expression that is used in defining the partitioning of the result set. Each *column-name* that is referenced in a *partitioning-expression* must unambiguously reference a result set column of the OLAP function subselect statement (SQLSTATE 42702 or 42703). A *partitioning-expression* cannot include a scalar fullselect or an XMLQUERY or XMLEXISTS expression (SQLSTATE 42822), or any function or query that is not deterministic or that has an external action (SQLSTATE 42845).

window-order-clause

ORDER BY (*sort-key-expression*,...)

Defines the ordering of rows within a partition that determines the value of the OLAP function or the meaning of the ROW values in the window-aggregation-group-clause (it does not define the ordering of the query result set).

sort-key-expression

An expression used in defining the ordering of the rows within a window partition. Each column name referenced in a *sort-key-expression* must unambiguously reference a column of the result set of the subselect, including the OLAP function (SQLSTATE 42702 or 42703). A *sort-key-expression* cannot include a scalar fullselect or an XMLQUERY or XMLEXISTS expression (SQLSTATE 42822), or any function or query that is not deterministic or that has an external action (SQLSTATE 42845). This clause is required for the RANK and DENSE_RANK functions (SQLSTATE 42601).

ASC

Uses the values of the sort-key-expression in ascending order.

DESC

Uses the values of the sort-key-expression in descending order.

NULLS FIRST

The window ordering considers null values *before* all non-null values in the sort order.

NULLS LAST

The window ordering considers null values *after* all non-null values in the sort order.

ORDER OF *table-designator*

Specifies that the same ordering used in *table-designator* should be applied to the result table of the subselect. There must be a table reference matching *table-designator* in the FROM clause of the subselect that specifies this clause (SQLSTATE 42703). The subselect (or fullselect) corresponding to the specified *table-designator* must include an ORDER BY clause that is dependant on the data (SQLSTATE 428FI). The ordering that is applied is the same as if the columns of the ORDER BY clause in the nested subselect (or fullselect) were included in the outer subselect (or fullselect), and these columns were specified in place of the ORDER OF clause.

window-aggregation-group-clause

The aggregation group of a row R is a set of rows defined in relation to R (in the ordering of the rows of R's partition). This clause specifies the aggregation group. If this clause is not specified and a window-order-clause is also not specified, the aggregation group consists of all the rows of the window partition. This default can be specified explicitly using RANGE (as shown) or ROWS.

If window-order-clause is specified, the default behavior is different when window-aggregation-group-clause is not specified. The window aggregation group consists of all rows of the partition of R that precede R or that are peers of R in the window ordering of the window partition defined by the window-order-clause.

ROWS

Indicates the aggregation group is defined by counting rows.

RANGE

Indicates the aggregation group is defined by an offset from a sort key.

group-start

Specifies the starting point for the aggregation group. The aggregation group end is the current row. Specification of the group-start clause is equivalent to a group-between clause of the form "BETWEEN group-start AND CURRENT ROW".

group-between

Specifies the aggregation group start and end based on either ROWS or RANGE.

group-end

Specifies the ending point for the aggregation group. The aggregation group start is the current row. Specification of the group-end clause is equivalent to a group-between clause of the form "BETWEEN CURRENT ROW AND group-end".

UNBOUNDED PRECEDING

Includes the entire partition preceding the current row. This can be specified with either ROWS or RANGE. Also, this can be specified with multiple sort-key-expressions in the window-order-clause.

UNBOUNDED FOLLOWING

Includes the entire partition following the current row. This can be specified with either ROWS or RANGE. Also, this can be specified with multiple sort-key-expressions in the window-order-clause.

CURRENT ROW

Specifies the start or end of the aggregation group based on the current row. If ROWS is specified, the current row is the aggregation group boundary. If RANGE is specified, the aggregation group boundary includes the set of rows with the same values for the *sort-key-expressions* as the current row. This clause cannot be specified in *group-bound2* if *group-bound1* specifies *value* FOLLOWING.

value PRECEDING

Specifies either the range or number of rows preceding the current row. If ROWS is specified, then *value* is a positive integer indicating a number of rows. If RANGE is specified, then the data type of *value* must be comparable to the type of the sort-key-expression of the window-order-clause. There can only be one sort-key-expression, and the data type of the sort-key-expression must allow subtraction. This clause cannot be specified in *group-bound2* if *group-bound1* is CURRENT ROW or *value* FOLLOWING.

value FOLLOWING

Specifies either the range or number of rows following the current row. If ROWS is specified, then *value* is a positive integer indicating a number of rows. If RANGE is specified, then the data type of *value* must be comparable to the type of the sort-key-expression of the window-order-clause. There can only be one sort-key-expression, and the data type of the sort-key-expression must allow addition.

Examples

- Display the ranking of employees, in order by surname, according to their total salary (based on salary plus bonus) that have a total salary more than \$30,000.

```
SELECT EMPNO, LASTNAME, FIRSTNME, SALARY+BONUS AS TOTAL_SALARY,  
       RANK() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY  
FROM EMPLOYEE WHERE SALARY+BONUS > 30000  
ORDER BY LASTNAME
```

Note that if the result is to be ordered by the ranking, then replace ORDER BY LASTNAME with:

```
ORDER BY RANK_SALARY
```

or

```
ORDER BY RANK() OVER (ORDER BY SALARY+BONUS DESC)
```

- Rank the departments according to their average total salary.

```
SELECT WORKDEPT, AVG(SALARY+BONUS) AS AVG_TOTAL_SALARY,  
       RANK() OVER (ORDER BY AVG(SALARY+BONUS) DESC) AS RANK_AVG_SAL  
FROM EMPLOYEE  
GROUP BY WORKDEPT  
ORDER BY RANK_AVG_SAL
```

- Rank the employees within a department according to their education level. Having multiple employees with the same rank in the department should not increase the next ranking value.

```
SELECT WORKDEPT, EMPNO, LASTNAME, FIRSTNME, EDLEVEL,  
       DENSE_RANK() OVER  
       (PARTITION BY WORKDEPT ORDER BY EDLEVEL DESC) AS RANK_EDLEVEL  
FROM EMPLOYEE  
ORDER BY WORKDEPT, LASTNAME
```

- Provide row numbers in the result of a query.

```
SELECT ROW_NUMBER() OVER (ORDER BY WORKDEPT, LASTNAME) AS NUMBER,
       LASTNAME, SALARY
FROM EMPLOYEE
ORDER BY WORKDEPT, LASTNAME
```

- List the top five wage earners.

```
SELECT EMPNO, LASTNAME, FIRSTNAME, TOTAL_SALARY, RANK_SALARY
FROM (SELECT EMPNO, LASTNAME, FIRSTNAME, SALARY+BONUS AS TOTAL_SALARY,
       RANK() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY
FROM EMPLOYEE) AS RANKED_EMPLOYEE
WHERE RANK_SALARY < 6
ORDER BY RANK_SALARY
```

Note that a nested table expression was used to first compute the result, including the rankings, before the rank could be used in the WHERE clause. A common table expression could also have been used.

- For each department, list employee salaries and show how much less each person makes compared to the employee in that department with the next highest salary.

```
SELECT EMPNO, WORKDEPT, LASTNAME, FIRSTNAME, JOB, SALARY,
       LEAD(SALARY, 1) OVER (PARTITION BY WORKDEPT
                           ORDER BY SALARY) - SALARY AS DELTA_SALARY
FROM EMPLOYEE
ORDER BY WORKDEPT, SALARY
```

- Calculate an employee's salary relative to the salary of the employee who was first hired for the same type of job.

```
SELECT JOB, HIREDATE, EMPNO, LASTNAME, FIRSTNAME, SALARY,
       FIRST_VALUE(SALARY) OVER (PARTITION BY JOB
                                ORDER BY HIREDATE) AS FIRST_SALARY,
       SALARY - FIRST_VALUE(SALARY) OVER (PARTITION BY JOB
                                          ORDER BY HIREDATE) AS DELTA_SALARY
FROM EMPLOYEE
ORDER BY JOB, HIREDATE
```

- Calculate the average close price for stock XYZ during the month of January, 2006. If a stock doesn't trade on a given day, its close price in the DAILYSTOCKDATA table is the null value. Instead of returning the null value for days that a stock doesn't trade, use the COALESCE function and LAG function to return the close price for the most recent day the stock was traded. Limit the search for a previous not-null close value to one month prior to January 1st, 2006.

```
WITH V1(SYMBOL, TRADINGDATE, CLOSEPRICE) AS
(
SELECT SYMBOL, TRADINGDATE,
       COALESCE(CLOSEPRICE,
               LAG(CLOSEPRICE,
                   1,
                   CAST(NULL AS DECIMAL(8,2)),
                   'IGNORE NULLS'))
       OVER (PARTITION BY SYMBOL
            ORDER BY TRADINGDATE)
FROM DAILYSTOCKDATA
WHERE SYMBOL = 'XYZ' AND
       TRADINGDATE BETWEEN '2005-12-01' AND '2006-01-31'
)
SELECT SYMBOL, AVG(CLOSEPRICE) AS AVG
FROM V1
WHERE TRADINGDATE BETWEEN '2006-01-01' AND '2006-01-31'
GROUP BY SYMBOL
```

- Calculate the 30-day moving average for stocks ABC and XYZ during the year 2005.

OLAP specifications

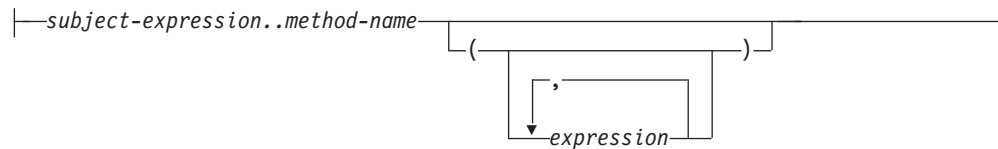
```
WITH V1(SYMBOL, TRADINGDATE, MOVINGAVG30DAY) AS
(
  SELECT SYMBOL, TRADINGDATE,
    AVG(CLOSEPRICE) OVER (PARTITION BY SYMBOL
      ORDER BY TRADINGDATE
      ROWS BETWEEN 29 PRECEDING AND CURRENT ROW)
  FROM DAILYSTOCKDATA
  WHERE SYMBOL IN ('ABC', 'XYZ')
    AND TRADINGDATE BETWEEN DATE('2005-01-01') - 2 MONTHS
      AND '2005-12-31'
)
SELECT SYMBOL, TRADINGDATE, MOVINGAVG30DAY
  FROM V1
  WHERE TRADINGDATE BETWEEN '2005-01-01' AND '2005-12-31'
  ORDER BY SYMBOL, TRADINGDATE
```

- Use an expression to define the cursor position and query a sliding window of 50 rows before that position.

```
SELECT DATE, FIRST_VALUE(CLOSEPRICE + 100) OVER
  (PARTITION BY SYMBOL
    ORDER BY DATE
    ROWS BETWEEN 50 PRECEDING AND 1 PRECEDING) AS FV
  FROM DAILYSTOCKDATA
  ORDER BY DATE
```

Method invocation

method-invocation:



Both system-generated observer and mutator methods, as well as user-defined methods are invoked using the double-dot operator.

subject-expression

An expression with a static result type that is a user-defined structured type.

method-name

The unqualified name of a method. The static type of *subject-expression* or one of its supertypes must include a method with the specified name.

(expression,...)

The arguments of *method-name* are specified within parentheses. Empty parentheses can be used to indicate that there are no arguments. The *method-name* and the data types of the specified argument expressions are used to resolve to the specific method, based on the static type of *subject-expression*.

The double-dot operator used for method invocation is a high precedence left to right infix operator. For example, the following two expressions are equivalent:

`a..b..c + x..y..z`

and

`((a..b)..c) + ((x..y)..z)`

If a method has no parameters other than its subject, it can be invoked with or without parentheses. For example, the following two expressions are equivalent:

```
point1..x
point1..x()
```

Null subjects in method calls are handled as follows:

- If a system-generated mutator method is invoked with a null subject, an error results (SQLSTATE 2202D)
- If any method other than a system-generated mutator is invoked with a null subject, the method is not executed, and its result is null. This rule includes user-defined methods with SELF AS RESULT.

When a database object (a package, view, or trigger, for example) is created, the best fit method that exists for each of its method invocations is found.

Note: Methods of types defined WITH FUNCTION ACCESS can also be invoked using the regular function notation. Function resolution considers all functions, as well as methods with function access as candidate functions. However, functions cannot be invoked using method invocation. Method resolution considers all methods and does not consider functions as candidate methods. Failure to resolve to an appropriate function or method results in an error (SQLSTATE 42884).

Example

- Use the double-dot operator to invoke a method called AREA. Assume the existence of a table called RINGS, with a column CIRCLE_COL of structured type CIRCLE. Also, assume that the method AREA has been defined previously for the CIRCLE type as AREA() RETURNS DOUBLE.

```
SELECT CIRCLE_COL..AREA() FROM RINGS
```

Subtype treatment

subtype-treatment:

```
|—TREAT—(—expression—AS—data-type—)|
```

The *subtype-treatment* is used to cast a structured type expression into one of its subtypes. The static type of *expression* must be a user-defined structured type, and that type must be the same type as, or a supertype of, *data-type*. If the type name in *data-type* is unqualified, the SQL path is used to resolve the type reference. The static type of the result of subtype-treatment is *data-type*, and the value of the subtype-treatment is the value of the expression. At run time, if the dynamic type of the expression is not *data-type* or a subtype of *data-type*, an error is returned (SQLSTATE 0D000).

Example

- If an application knows that all column object instances in a column CIRCLE_COL have the dynamic type COLOREDCIRCLE, use the following query to invoke the method RGB on such objects. Assume the existence of a table called RINGS, with a column CIRCLE_COL of structured type CIRCLE. Also, assume that COLOREDCIRCLE is a subtype of CIRCLE and that the method RGB has been defined previously for COLOREDCIRCLE as RGB() RETURNS DOUBLE.

```
SELECT TREAT (CIRCLE_COL AS COLOREDCIRCLE)..RGB()
FROM RINGS
```

Subtype treatment

At run time, if there are instances of dynamic type `CIRCLE`, an error is raised (SQLSTATE 0D000). This error can be avoided by using the `TYPE` predicate in a `CASE` expression, as follows:

```
SELECT (CASE
  WHEN CIRCLE_COL IS OF (COLOREDCIRCLE)
  THEN TREAT (CIRCLE_COL AS COLOREDCIRCLE)..RGB()
  ELSE NULL
END)
FROM RINGS
```

Sequence reference

sequence-reference:

```
| nextval-expression |
| prevval-expression |
```

nextval-expression:

```
| NEXT VALUE FOR sequence-name |
```

prevval-expression:

```
| PREVIOUS VALUE FOR sequence-name |
```

NEXT VALUE FOR *sequence-name*

A `NEXT VALUE` expression generates and returns the next value for the sequence specified by *sequence-name*.

PREVIOUS VALUE FOR *sequence-name*

A `PREVIOUS VALUE` expression returns the most recently generated value for the specified sequence for a previous statement within the current application process. This value can be referenced repeatedly by using `PREVIOUS VALUE` expressions that specify the name of the sequence. There may be multiple instances of `PREVIOUS VALUE` expressions specifying the same sequence name within a single statement; they all return the same value. In a partitioned database environment, a `PREVIOUS VALUE` expression may not return the most recently generated value.

A `PREVIOUS VALUE` expression can only be used if a `NEXT VALUE` expression specifying the same sequence name has already been referenced in the current application process, in either the current or a previous transaction (SQLSTATE 51035).

Notes

- A new value is generated for a sequence when a `NEXT VALUE` expression specifies the name of that sequence. However, if there are multiple instances of a `NEXT VALUE` expression specifying the same sequence name within a query, the counter for the sequence is incremented only once for each row of the result, and all instances of `NEXT VALUE` return the same value for a row of the result.
- The same sequence number can be used as a unique key value in two separate tables by referencing the sequence number with a `NEXT VALUE` expression for the first row (this generates the sequence value), and a `PREVIOUS VALUE` expression for the other rows (the instance of `PREVIOUS VALUE` refers to the sequence value most recently generated in the current session), as shown below:

```
INSERT INTO order(orderno, cutno)
VALUES (NEXT VALUE FOR order_seq, 123456);
```

```
INSERT INTO line_item (orderno, partno, quantity)
VALUES (PREVIOUS VALUE FOR order_seq, 987654, 1);
```

- NEXT VALUE and PREVIOUS VALUE expressions can be specified in the following places:
 - select-statement or SELECT INTO statement (within the select-clause, provided that the statement does not contain a DISTINCT keyword, a GROUP BY clause, an ORDER BY clause, a UNION keyword, an INTERSECT keyword, or EXCEPT keyword)
 - INSERT statement (within a VALUES clause)
 - INSERT statement (within the select-clause of the fullselect)
 - UPDATE statement (within the SET clause (either a searched or a positioned UPDATE statement), except that NEXT VALUE cannot be specified in the select-clause of the fullselect of an expression in the SET clause)
 - SET Variable statement (except within the select-clause of the fullselect of an expression; a NEXT VALUE expression can be specified in a trigger, but a PREVIOUS VALUE expression cannot)
 - VALUES INTO statement (within the select-clause of the fullselect of an expression)
 - CREATE PROCEDURE statement (within the routine-body of an SQL procedure)
 - CREATE TRIGGER statement within the triggered-action (a NEXT VALUE expression may be specified, but a PREVIOUS VALUE expression cannot)
- NEXT VALUE and PREVIOUS VALUE expressions cannot be specified (SQLSTATE 428F9) in the following places:
 - Join condition of a full outer join
 - DEFAULT value for a column in a CREATE or ALTER TABLE statement
 - Generated column definition in a CREATE OR ALTER TABLE statement
 - Summary table definition in a CREATE TABLE or ALTER TABLE statement
 - Condition of a CHECK constraint
 - CREATE TRIGGER statement (a NEXT VALUE expression may be specified, but a PREVIOUS VALUE expression cannot)
 - CREATE VIEW statement
 - CREATE METHOD statement
 - CREATE FUNCTION statement
 - An argument list of an XMLQUERY, XMLEXISTS, or XMLTABLE expression
- In addition, a NEXT VALUE expression cannot be specified (SQLSTATE 428F9) in the following places:
 - CASE expression
 - Parameter list of an aggregate function
 - Subquery in a context other than those explicitly allowed above
 - SELECT statement for which the outer SELECT contains a DISTINCT operator
 - Join condition of a join
 - SELECT statement for which the outer SELECT contains a GROUP BY clause
 - SELECT statement for which the outer SELECT is combined with another SELECT statement using the UNION, INTERSECT, or EXCEPT set operator

Sequence reference

- Nested table expression
- Parameter list of a table function
- WHERE clause of the outer-most SELECT statement, or a DELETE or UPDATE statement
- ORDER BY clause of the outer-most SELECT statement
- select-clause of the fullselect of an expression, in the SET clause of an UPDATE statement
- IF, WHILE, DO ... UNTIL, or CASE statement in an SQL routine
- When a value is generated for a sequence, that value is consumed, and the next time that a value is requested, a new value will be generated. This is true even when the statement containing the NEXT VALUE expression fails or is rolled back.

If an INSERT statement includes a NEXT VALUE expression in the VALUES list for the column, and if an error occurs at some point during the execution of the INSERT (it could be a problem in generating the next sequence value, or a problem with the value for another column), then an insertion failure occurs (SQLSTATE 23505), and the value generated for the sequence is considered to be consumed. In some cases, reissuing the same INSERT statement might lead to success.

For example, consider an error that is the result of the existence of a unique index for the column for which NEXT VALUE was used and the sequence value generated already exists in the index. It is possible that the next value generated for the sequence is a value that does not exist in the index and so the subsequent INSERT would succeed.

- If in generating a value for a sequence, the maximum value for the sequence is exceeded (or the minimum value for a descending sequence) and cycles are not permitted, then an error occurs (SQLSTATE 23522). In this case, the user could ALTER the sequence to extend the range of acceptable values, or enable cycles for the sequence, or DROP and CREATE a new sequence with a different data type that has a larger range of values.

For example, a sequence may have been defined with a data type of SMALLINT, and eventually the sequence runs out of assignable values. DROP and re-create the sequence with the new definition to redefine the sequence as INTEGER.

- A reference to a NEXT VALUE expression in the select statement of a cursor refers to a value that is generated for a row of the result table. A sequence value is generated for a NEXT VALUE expression for each row that is fetched from the database. If blocking is done at the client, the values may have been generated at the server prior to the processing of the FETCH statement. This can occur when there is blocking of the rows of the result table. If the client application does not explicitly FETCH all the rows that the database has materialized, then the application will not see the results of all the generated sequence values (for the materialized rows that were not returned).
- A reference to a PREVIOUS VALUE expression in the select statement of a cursor refers to a value that was generated for the specified sequence prior to the opening of the cursor. However, closing the cursor can affect the values returned by PREVIOUS VALUE for the specified sequence in subsequent statements, or even for the same statement in the event that the cursor is reopened. This would be the case when the select statement of the cursor included a reference to NEXT VALUE for the same sequence name.
- *Compatibilities*
 - For compatibility with previous versions of DB2:

- NEXTVAL and PREVVAL can be specified in place of NEXT VALUE and PREVIOUS VALUE
- For compatibility with IBM IDS:
 - *sequence-name*.NEXTVAL can be specified in place of NEXT VALUE FOR *sequence-name*
 - *sequence-name*.CURRVAL can be specified in place of PREVIOUS VALUE FOR *sequence-name*

Examples

Assume that there is a table called "order", and that a sequence called "order_seq" is created as follows:

```
CREATE SEQUENCE order_seq
  START WITH 1
  INCREMENT BY 1
  NO MAXVALUE
  NO CYCLE
  CACHE 24
```

Following are some examples of how to generate an "order_seq" sequence number with a NEXT VALUE expression:

```
INSERT INTO order(orderno, custno)
  VALUES (NEXT VALUE FOR order_seq, 123456);
```

or

```
UPDATE order
  SET orderno = NEXT VALUE FOR order_seq
  WHERE custno = 123456;
```

or

```
VALUES NEXT VALUE FOR order_seq INTO :hv_seq;
```

ROW CHANGE expression

row-change-expression:

```
|—ROW CHANGE—|—TOKEN—|—FOR—table-designator—|
|—TIMESTAMP—|
```

A ROW CHANGE expression returns a token or a timestamp that represents the last change to a row.

TOKEN

Specifies that a BIGINT value representing a relative point in the modification sequence of a row is to be returned. If the row has not been changed, the result is a token that represents when the initial value was inserted. The result can be null. ROW CHANGE TOKEN is not deterministic.

TIMESTAMP

Specifies that a TIMESTAMP value representing the last time that a row was changed is to be returned. If the row has not been changed, the result is the time that the initial value was inserted. The result can be null. ROW CHANGE TIMESTAMP is not deterministic.

FOR *table-designator*

Identifies the table in which the expression is referenced. The *table-designator*

ROW CHANGE expression

must uniquely identify a base table, view, or nested table expression (SQLSTATE 42867). If *table-designator* identifies a view or a nested table expression, the ROW CHANGE expression returns the TOKEN or TIMESTAMP of the base table of the view or nested table expression. The view or nested table expression must contain only one base table in its outer subselect (SQLSTATE 42867). The *table-designator* must be deletable (SQLSTATE 42703). For information about deletable views, see the “Notes” section of “CREATE VIEW”. The table designator of a ROW CHANGE TIMESTAMP expression must resolve to a base table that contains a row change timestamp column (SQLSTATE 55068).

Notes

- The values returned by the ROW CHANGE TOKEN expression can be used with the RID_BIT scalar function by applications that use optimistic locking.

Examples

- Return a timestamp value that corresponds to the most recent change to each row from the EMPLOYEE table for employees in department 20. Assume that the EMPLOYEE table has been altered to contain a column defined with the ROW CHANGE TIMESTAMP clause.

```
SELECT ROW CHANGE TIMESTAMP FOR EMPLOYEE
FROM EMPLOYEE WHERE DEPTNO = 20
```

- Return a BIGINT value that represents a relative point in the modification sequence of the row corresponding to employee number 3500. Also return the RID_BIT scalar function value that is to be used in an optimistic locking DELETE scenario. Specify the WITH UR option to get the latest ROW CHANGE TOKEN value.

```
SELECT ROW CHANGE TOKEN FOR EMPLOYEE, RID_BIT (EMPLOYEE)
FROM EMPLOYEE WHERE EMPNO = '3500' WITH UR
```

The above statement succeeds whether or not there is a row change timestamp column in the EMPLOYEE table. The following searched DELETE statement deletes the row specified by the ROW CHANGE TOKEN and RID_BIT values from the above SELECT statement, assuming the two parameter marker values are set to the values obtained from the above statement.

```
DELETE FROM EMPLOYEE E
WHERE RID_BIT (E) = ? AND ROW CHANGE TOKEN FOR E = ?
```

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 4000, a graphic string with a length attribute greater than 2000, or a LOB string of any size.
- The value of a host variable can be null (that is, the variable may have a negative indicator variable).
- The code page conversion of operands of predicates involving two or more operands, with the exception of LIKE, is done according to the rules for string conversions.

- Use of a structured type value is limited to the NULL predicate and the TYPE predicate.
- In a Unicode database, all predicates that accept a character or graphic string will accept any string type for which conversion is supported.

A fullselect is a form of the SELECT statement that, when used in a predicate, is also called a *subquery*.

Predicate processing for queries

A predicate is an element of a search condition that expresses or implies a comparison operation. Predicates can be grouped into four categories that are determined by how and when the predicate is used in the evaluation process. The categories are listed below, ordered in terms of performance starting with the most favorable:

- Range delimiting predicates are those used to bracket an index scan; they provide start or stop key values for the index search. These predicates are evaluated by the index manager.
- Index sargable predicates are not used to bracket a search, but are evaluated from the index if one is chosen, because the columns involved in the predicate are part of the index key. These predicates are also evaluated by the index manager.
- Data sargable predicates are predicates that cannot be evaluated by the index manager, but can be evaluated by Data Management Services (DMS). Typically, these predicates require the access of individual rows from a base table. If necessary, DMS will retrieve the columns needed to evaluate the predicate, as well as any others to satisfy the columns in the SELECT list that could not be obtained from the index.
- Residual predicates are those that require I/O beyond the simple accessing of a base table. Examples of residual predicates include those using quantified subqueries (subqueries with ANY, ALL, SOME, or IN), or reading LONG VARCHAR or large object (LOB) data that is stored separately from the table. These predicates are evaluated by Relational Data Services (RDS) and are the most expensive of the four categories of predicates.

The following table provides examples of various predicates and identifies their type based on the context in which they are used.

Note: In these examples, assume that a multi-column ascending index exists on (c1, c2, c3) and is used in evaluating the predicates where appropriate. If any column in the index is in descending order, the start and stop keys might be switched for range delimiting predicates.

Table 17. Predicate processing for different queries

Predicates	Column c1	Column c2	Column c3	Comments
c1 = 1 and c2 = 2 and c3 = 3	Range delimiting (start-stop)	Range delimiting (start-stop)	Range delimiting (start-stop)	The equality predicates on all the columns of the index can be applied as start-stop keys.
c1 = 1 and c2 = 2 and c3 ≥ 3	Range delimiting (start-stop)	Range delimiting (start-stop)	Range delimiting (start)	Columns c1 and c2 are bound by equality predicates, and the predicate on c3 is only applied as a start key.

Predicate processing for queries

Table 17. Predicate processing for different queries (continued)

Predicates	Column c1	Column c2	Column c3	Comments
c1 \geq 1 and c2 = 2	Range delimiting (start)	Range delimiting (start-stop)	Not applicable	The leading column c1 has a \geq predicate and can be used as a start key. The following column c2 has an equality predicate, and therefore can also be applied as a start-stop key.
c1 = 1 and c3 = 3	Range delimiting (start-stop)	Not applicable	Index sargable	The predicate on c3 cannot be used as a start-stop key, because there is no predicate on c2. It can, however, be applied as an index sargable predicate.
c1 = 1 and c2 > 2 and c3 = 3	Range delimiting (start-stop)	Range delimiting (start)	Index sargable	The predicate on c3 cannot be applied as a start-stop predicate because the previous column has a > predicate. Had it been a \geq instead, we would be able to use it as a start-stop key.
c1 = 1 and c2 \leq 2 and c4 = 4	Range delimiting (start-stop)	Range delimiting (stop)	Data sargable	Here the predicate on c2 is a \leq predicate. It can be used as a stop key. The predicate on c4 cannot be applied on the index and is applied as a data sargable predicate during the FETCH.
c2 = 2 and UDF_with_external_action (c4)	Not applicable	Index sargable	Residual	The leading column c1 does not have a predicate, and therefore the predicate on c2 can be applied as an index sargable predicate where the whole index is scanned. The predicate involving the user-defined function with external action is applied as a residual predicate.
c1 = 1 or c2 = 2	Index sargable	Index sargable	Not applicable	The presence of an OR does not allow us this multi-column index to be used as start-stop keys. This might have been possible had there been two indexes, one with a leading column on c1, and the other with a leading column on c2, and the DB2 optimizer chose an "index-ORing" plan. However, in this case the two predicates are treated as index sargable predicates.

Table 17. Predicate processing for different queries (continued)

Predicates	Column c1	Column c2	Column c3	Comments
c1 < 5 and (c2 = 2 or c3 = 3)	Range delimiting (stop)	Index sargable	Index sargable	Here the leading column c1 is exploited to stop the index scan from using the predicate with a stop key. The OR predicate on c2 and c3 are applied as index sargable predicates.

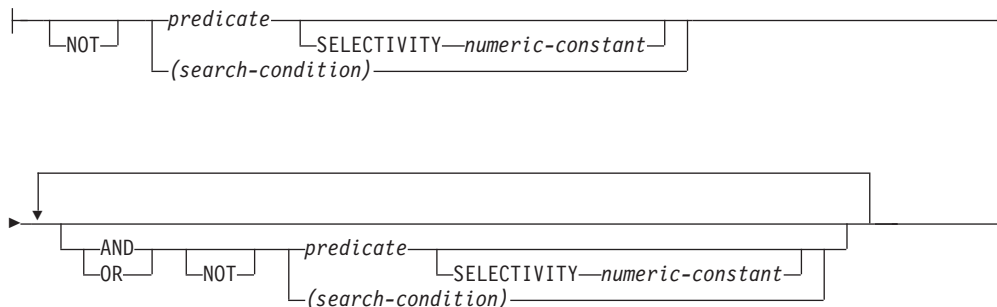
The DB2 optimizer employs the query rewrite mechanism to transform many complex user-written predicates into better performing queries, as shown in the following table:

Table 18. Query rewrite predicates

Original predicate or query	Optimized predicates	Comments
c1 between 5 and 10	c1 ≥ 5 and c1 ≤ 10	The BETWEEN predicates are rewritten into the equivalent range delimiting predicates so that they can be used internally as though the user specified the range delimiting predicates.
c1 not between 5 and 10	c1 < 5 or c1 > 10	The presence of the OR predicate does not allow the use of a start-stop key unless the DB2 optimizer chooses an index-ORing plan.
SELECT * FROM t1 WHERE EXISTS (SELECT c1 FROM t2 WHERE t1.c1 = t2.c1)	SELECT t1.* FROM t1 EOJOIN t2 WHERE t1.c1= t2.c1	The subquery might be transformed into a join.
SELECT * FROM t1 WHERE t1.c1 IN (SELECT c1 FROM t2)	SELECT t1.* FROM t1 EOJOIN t2 WHERE t1.c1= t2.c1	This is similar to the transformation for the EXISTS predicate example above.
c1 like 'abc%'	c1 ≥ 'abc X X X ' and c1 ≤ 'abc Y Y Y'	If we have c1 as the leading column of an index, DB2 generates these predicates so that they can be applied as range-delimiting start-stop predicates. Here the characters X and Y are symbolic of the lowest and highest collating character.
c1 like 'abc%def'	c1 ≥ 'abc X X X ' and c1 ≤ 'abc Y Y Y' and c1 like 'abc%def'	This is similar to the previous case, except that we have to also apply the original predicate as an index sargable predicate. This ensures that the characters def match correctly.

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 19, in which P and Q are any predicates:

Table 19. Truth Tables for AND and OR

P	Q	P AND Q	P OR Q
True	True	True	True
True	False	False	True
True	Unknown	Unknown	True
False	True	False	True
False	False	False	False
False	Unknown	False	Unknown
Unknown	True	Unknown	True
Unknown	False	False	Unknown
Unknown	Unknown	Unknown	Unknown

NOT(true) is false, NOT(false) is true, and NOT(unknown) is unknown.

Search conditions within parentheses are evaluated first. If the order of evaluation is not specified by parentheses, NOT is applied before AND, and AND is applied before OR. The order in which operators at the same precedence level are evaluated is undefined to allow for optimization of search conditions.

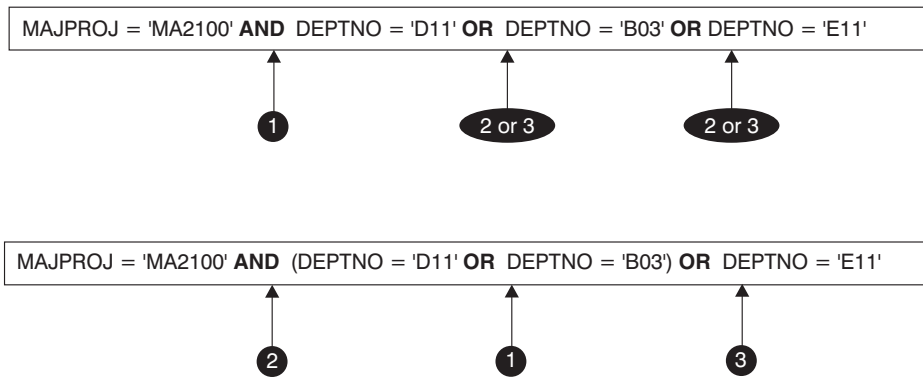


Figure 13. Search Conditions Evaluation Order

SELECTIVITY *value*

The SELECTIVITY clause is used to indicate to DB2 what the expected selectivity percentage is for the predicate. SELECTIVITY can be specified only when the predicate is a user-defined predicate.

A user-defined predicate is a predicate that consists of a user-defined function invocation, in the context of a predicate specification that matches the predicate specification on the PREDICATES clause of CREATE FUNCTION. For example, if the function foo is defined with PREDICATES WHEN=1..., then the following use of SELECTIVITY is valid:

```
SELECT *
FROM STORES
WHERE foo(parm,parm) = 1 SELECTIVITY 0.004
```

The selectivity value must be a numeric literal value in the inclusive range from 0 to 1 (SQLSTATE 42615). If SELECTIVITY is not specified, the default value is 0.01 (that is, the user-defined predicate is expected to filter out all but one percent of all the rows in the table). The SELECTIVITY default can be changed for any given function by updating its SELECTIVITY column in the SYSSTAT.ROUTINES view. An error will be returned if the SELECTIVITY clause is specified for a non user-defined predicate (SQLSTATE 428E5).

A user-defined function (UDF) can be applied as a user-defined predicate and, hence, is potentially applicable for index exploitation if:

- the predicate specification is present in the CREATE FUNCTION statement
- the UDF is invoked in a WHERE clause being compared (syntactically) in the same way as specified in the predicate specification
- there is no negation (NOT operator)

Examples

In the following query, the within UDF specification in the WHERE clause satisfies all three conditions and is considered a user-defined predicate.

```
SELECT *
FROM customers
WHERE within(location, :sanJose) = 1 SELECTIVITY 0.2
```

However, the presence of within in the following query is not index-exploitable due to negation and is not considered a user-defined predicate.

```
SELECT *
FROM customers
WHERE NOT(within(location, :sanJose) = 1) SELECTIVITY 0.3
```

Search conditions

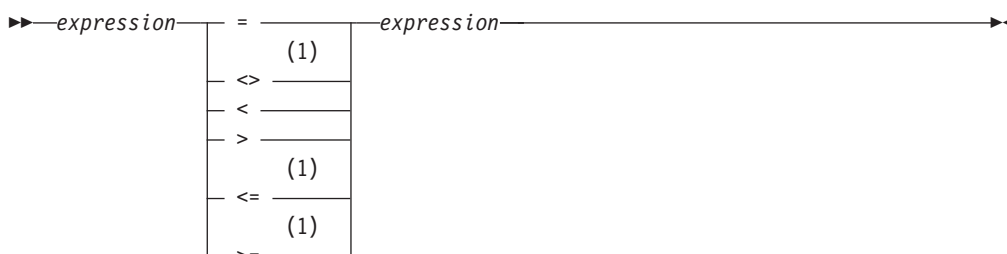
In the next example, consider identifying customers and stores that are within a certain distance of each other. The distance from one store to another is computed by the radius of the city in which the customers live.

```
SELECT *
  FROM customers, stores
 WHERE distance(customers.loc, stores.loc) <
        CityRadius(stores.loc) SELECTIVITY 0.02
```

In the above query, the predicate in the WHERE clause is considered a user-defined predicate. The result produced by CityRadius is used as a search argument to the range producer function.

However, since the result produced by CityRadius is used as a range producer function, the above user-defined predicate will not be able to make use of the index extension defined on the stores.loc column. Therefore, the UDF will make use of only the index defined on the customers.loc column.

Basic predicate



Notes:

- 1 The following forms of the comparison operators are also supported in basic and quantified predicates: $\wedge=$, $\wedge<$, $\wedge>$, \neq , $!<$, and $!>$. In code pages 437, 819, and 850, the forms $\neg=$, $\neg<$, and $\neg>$ are supported. All of these product-specific forms of the comparison operators are intended only to support existing SQL statements that use these operators, and are not recommended for use when writing new SQL statements.

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

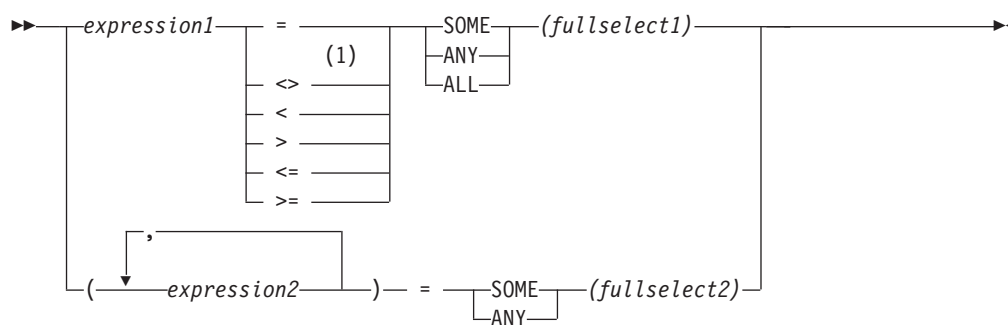
Is True If and Only If...

- $x = y$ x is equal to y
- $x \neq y$ x is not equal to y
- $x < y$ x is less than y
- $x > y$ x is greater than y
- $x \geq y$ x is greater than or equal to y
- $x \leq y$ x is less than or equal to y

Examples:

```
EMPNO= '528671'
SALARY < 20000
PRSTAFF <> :VAR1
SALARY > (SELECT AVG(SALARY) FROM EMPLOYEE)
```

Quantified predicate



Notes:

- 1 The following forms of the comparison operators are also supported in basic and quantified predicates: $\wedge=$, $\wedge<$, $\wedge>$, \neq , $!<$, and $!>$. In code pages 437, 819, and 850, the forms $\neg=$, $\neg<$, and $\neg>$ are supported. All of these product-specific forms of the comparison operators are intended only to support existing SQL statements that use these operators, and are not recommended for use when writing new SQL statements.

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.

Quantified predicate

TBLAB:

COLA	COLB
1	12
2	12
3	13
4	14
-	-

TBLXY:

COLX	COLY
2	22
3	23

Figure 14. Tables for quantified predicate examples

Example 1

```
SELECT COLA FROM TBLAB
WHERE COLA = ANY(SELECT COLX FROM TBLXY)
```

Results in 2,3. The subselect returns (2,3). COLA in rows 2 and least one of these values.

Example 2

```
SELECT COLA FROM TBLAB
WHERE COLA > ANY(SELECT COLX FROM TBLXY)
```

Results in 3,4. The subselect returns (2,3). COLA in rows 3 and 4 is greater than at least one of these values.

Example 3

```
SELECT COLA FROM TBLAB
WHERE COLA > ALL(SELECT COLX FROM TBLXY)
```

Results in 4. The subselect returns (2,3). COLA in row 4 is the only one that is greater than both these values.

Example 4

```
SELECT COLA FROM TBLAB
WHERE COLA > ALL(SELECT COLX FROM TBLXY
WHERE COLX<0)
```

Results in 1,2,3,4, null. The subselect returns no values. Thus, the predicate is true for all rows in TBLAB.

Example 5

```
SELECT * FROM TBLAB
WHERE (COLA,COLB+10) = SOME (SELECT COLX, COLY FROM TBLXY)
```

The subselect returns all entries from TBLXY. The predicate is true for the subselect, hence the result is as follows:

COLA	COLB
2	12
3	13

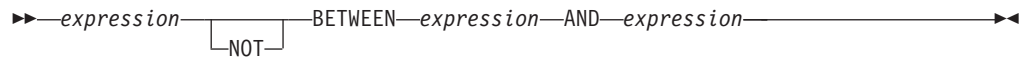
Example 6

```
SELECT * FROM TBLAB
WHERE (COLA,COLB) = ANY (SELECT COLX,COLY-10 FROM TBLXY)
```

The subselect returns COLX and COLY-10 from TBLXY. The predicate is true for the subselect, hence the result is as follows:

COLA	COLB
2	12
3	13

BETWEEN predicate



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 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.

EXISTS predicate



The EXISTS predicate tests for the existence of certain rows.

The fullselect may specify any number of columns, and

- The result is true only if the number of rows specified by the fullselect is not zero.

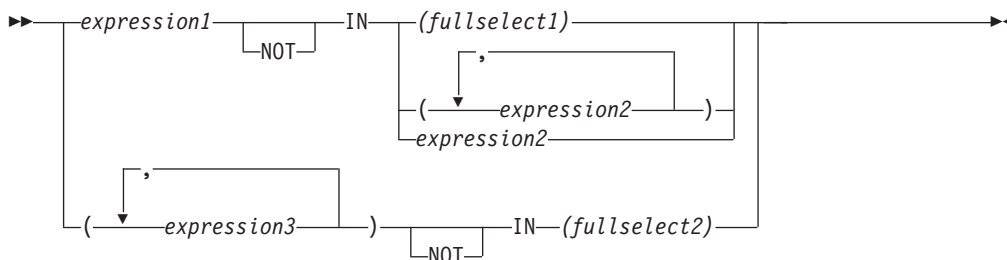
EXISTS predicate

- 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 can be used to determine the attributes of the result used in the comparison.

The values for the expressions in the IN predicate (including corresponding columns of a fullselect) can have different code pages. If a conversion is necessary, the code page is determined by applying rules for string conversions to the IN list first, and then to the predicate, using the derived code page for the IN list as the second operand.

Examples

Example 1: The following 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 20. IN Predicate example

Expressions	Type	Code Page
COL_1	column	850
HV_2	host variable	437
HV_3	host variable	437
CON_1	constant	850

When evaluating the predicate:

```
COL_1 IN (:HV_2, :HV_3, CON_4)
```

the two host variables will be converted to code page 850, based on the rules for string conversions.

Example 4: The following evaluates to true if the specified year in EMENDATE (the date an employee activity on a project ended) matches any of the values specified in the list (the current year or the two previous years):

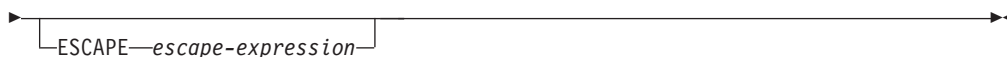
```
YEAR(EMENDATE) IN (YEAR(CURRENT DATE),
                   YEAR(CURRENT DATE - 1 YEAR),
                   YEAR(CURRENT DATE - 2 YEARS))
```

Example 5: The following 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* [NOT] LIKE *pattern-expression* ►



The LIKE predicate searches for strings that have a certain pattern. The pattern is specified by a string in which the underscore and the percent sign may have special meanings. Trailing blanks in a pattern are part of the pattern.

If the value of any of the arguments is null, the result of the LIKE predicate is unknown.

The values for *match-expression*, *pattern-expression*, and *escape-expression* are compatible string expressions. There are slight differences in the types of string expressions supported for each of the arguments. The valid types of expressions are listed under the description of each argument.

None of the expressions can yield a distinct type. However, it can be a function that casts a distinct type to its source type.

match-expression

An expression that specifies the string that is to be examined to see if it conforms to a certain pattern of characters.

The expression can be specified by:

- A constant
- A special register
- A global variable
- 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:

- A constant
- A special register
- A global variable
- A host variable
- A scalar function whose operands are any of the above
- An expression concatenating any of the above
- An SQL procedure parameter

with the following restrictions:

- 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.

The following are examples of invalid string expressions or strings:

- SQL user-defined function parameters
- Trigger transition variables

- Local variables in dynamic compound statements

A **simple description** of the use of the LIKE predicate is that the pattern is used to specify the conformance criteria for values in the *match-expression*, where:

- The underscore character (`_`) represents any single character.
- The percent sign (`%`) represents a string of zero or more characters.
- Any other character represents itself.

If the *pattern-expression* needs to include either the underscore or the percent character, the *escape-expression* is used to specify a character to precede either the underscore or the percent character in the pattern.

A **rigorous description** of the use of the LIKE predicate follows. Note that this description ignores the use of the *escape-expression*; its use is covered later.

- Let m denote the value of *match-expression* and let p denote the value of *pattern-expression*. The string p is interpreted as a sequence of the minimum number of substring specifiers so each character of p is part of exactly one substring specifier. A substring specifier is an underscore, a percent sign, or any non-empty sequence of characters other than an underscore or a percent sign.

The result of the predicate is unknown if m or p is the null value. Otherwise, the result is either true or false. The result is true if m and p are both empty strings or there exists a partitioning of m into substrings such that:

- A substring of m is a sequence of zero or more contiguous characters and each character of m is part of exactly one substring.
- If the n th substring specifier is an underscore, the n th substring of m is any single character.
- If the n th substring specifier is a percent sign, the n th substring of m is any sequence of zero or more characters.
- If the n th substring specifier is neither an underscore nor a percent sign, the n th substring of m is equal to that substring specifier and has the same length as that substring specifier.
- The number of substrings of m is the same as the number of substring specifiers.

Thus, if p is an empty string and m is not an empty string, the result is false. Similarly, it follows that if m is an empty string and p is not an empty string (except for a string containing only percent signs), the result is false.

The predicate m NOT LIKE p is equivalent to the search condition NOT (m LIKE p).

When the *escape-expression* is specified, the *pattern-expression* must not contain the escape character identified by the *escape-expression*, except when immediately followed by the escape character, the underscore character, or the percent sign character (SQLSTATE 22025).

If the *match-expression* is a character string in an MBCS database, it can contain mixed data. In this case, the pattern can include both SBCS and non-SBCS characters. For non-Unicode databases, the special characters in the pattern are interpreted as follows:


- An SBCS halfwidth underscore refers to one SBCS character.
- A non-SBCS fullwidth underscore refers to one non-SBCS character.
- An SBCS halfwidth or non-SBCS fullwidth percent sign refers to zero or more SBCS or non-SBCS characters.

LIKE predicate

In a Unicode database, there is really no distinction between "single-byte" and "non-single-byte" characters. Although the UTF-8 format is a "mixed-byte" encoding of Unicode characters, there is no real distinction between SBCS and non-SBCS characters in UTF-8. Every character is a Unicode character, regardless of the number of bytes in UTF-8 format.

In a Unicode graphic column, every non-supplementary character, including the halfwidth underscore character (U&'\005F') and the halfwidth percent sign character (U&'\0025'), is two bytes in width. In a Unicode database, special characters in a pattern are interpreted as follows:

- For character strings, a halfwidth underscore character (X'5F') or a fullwidth underscore character (X'EFBCBF') refers to one Unicode character, and a halfwidth percent sign character (X'25') or a fullwidth percent sign character (X'EFBC85') refers to zero or more Unicode characters.
- For graphic strings, a halfwidth underscore character (U&'\005F') or a fullwidth underscore character (U&'\FF3F') refers to one Unicode character, and a halfwidth percent sign character (U&'\0025') or a fullwidth percent sign character (U&'\FF05') refers to zero or more Unicode characters.
- To be recognized as special characters when a locale-sensitive UCA-based collation is in effect, the underscore character and the percent sign character must not be followed by non-spacing combining marks (diacritics). For example, the pattern U&'\%0300' (percent sign character followed by

non-spacing combining grave accent) will be interpreted as a search for  and not as a search for zero or more Unicode characters followed by a letter with a grave accent.

A Unicode supplementary character is stored as two graphic code points in a Unicode graphic column. To match a Unicode supplementary character in a Unicode graphic column, use one underscore if the database uses locale-sensitive UCA-based collation, and two underscores otherwise. To match a Unicode supplementary character in a Unicode character column, use one underscore for all collations. To match a base character with one or more trailing non-spacing combining characters, use one underscore if the database uses locale-sensitive UCA-based collation. Otherwise, use as many underscore characters as the number of non-spacing combining characters plus the base character.

escape-expression

This optional argument is an expression that specifies a character to be used to modify the special meaning of the underscore (`_`) and percent (`%`) characters in the *pattern-expression*. This allows the LIKE predicate to be used to match values that contain the actual percent and underscore characters.

The expression can be specified by any one of:

- A constant
- A special register
- A global variable
- A host variable
- A scalar function 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.

- For character columns, the result of the expression must be one character, or a binary string containing exactly one byte (SQLSTATE 22019).
- For graphic columns, the result of the expression must be one character (SQLSTATE 22019).
- The result of the expression must not be a non-spacing combining character sequence (for example, U&'\0301', Combining Acute Accent).

When escape characters are present in the pattern string, an underscore, percent sign, or escape character can represent a literal occurrence of itself. This is true if the character in question is preceded by an odd number of successive escape characters. It is not true otherwise.

In a pattern, a sequence of successive escape characters is treated as follows:

- Let S be such a sequence, and suppose that S is not part of a larger sequence of successive escape characters. Suppose also that S contains a total of n characters. Then the rules governing S depend on the value of n:
 - If n is odd, S must be followed by an underscore or percent sign (SQLSTATE 22025). S and the character that follows it represent (n-1)/2 literal occurrences of the escape character followed by a literal occurrence of the underscore or percent sign.
 - If n is even, S represents n/2 literal occurrences of the escape character. Unlike the case where n is odd, S could end the pattern. If it does not end the pattern, it can be followed by any character (except, of course, an escape character, which would violate the assumption that S is not part of a larger sequence of successive escape characters). If S is followed by an underscore or percent sign, that character has its special meaning.

Following is an illustration of the effect of successive occurrences of the escape character which, in this case, is the back slash (\).

Pattern string

Actual Pattern

- | | |
|------|------------------------------------------------------------|
| \% | A percent sign |
| \\% | A back slash followed by zero or more arbitrary characters |
| \\\% | A back slash followed by a percent sign |

The code page used in the comparison is based on the code page of the *match-expression* value.

- The *match-expression* value is never converted.
- If the code page of *pattern-expression* is different from the code page of *match-expression*, the value of *pattern-expression* is converted to the code page of *match-expression*, unless either operand is defined as FOR BIT DATA (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).

Notes

- The number of trailing blanks is significant in both the *match-expression* and the *pattern-expression*. If the strings are not the same length, the shorter string is not padded with blank spaces. For example, the expression 'PADDED ' LIKE 'PADDED' would not result in a match.

LIKE predicate

- If the pattern specified in a LIKE predicate is a parameter marker, and a fixed-length character host variable is used to replace the parameter marker, the value specified for the host variable must have the correct length. If the correct length is not specified, the select operation will not return the intended results. For example, if the host variable is defined as CHAR(10), and the value WYSE% is assigned to that host variable, the host variable is padded with blanks on assignment. The pattern used is:

```
'WYSE%      '
```

The database manager searches for all values that start with WYSE and that end with five blank spaces. If you want to search only for values that start with 'WYSE', assign a value of 'WYSE% % % % %' to the host variable.

- The pattern is matched using the collation of the database, unless either operand is defined as FOR BIT DATA, in which case the pattern is matched using a binary comparison.

Examples

- Search for the string 'SYSTEMS' appearing anywhere within the PROJNAME column in the PROJECT table.

```
SELECT PROJNAME FROM PROJECT
WHERE PROJECT.PROJNAME LIKE '%SYSTEMS%'
```

- Search for a string with a first character of 'J' that is exactly two characters long in the FIRSTNAME column of the EMPLOYEE table.

```
SELECT FIRSTNAME FROM EMPLOYEE
WHERE EMPLOYEE.FIRSTNAME LIKE 'J_'
```

- Search for a string of any length, with a first character of 'J', in the FIRSTNAME column of the EMPLOYEE table.

```
SELECT FIRSTNAME FROM EMPLOYEE
WHERE EMPLOYEE.FIRSTNAME LIKE 'J%'
```

- In the CORP_SERVERS table, search for a string in the LA_SERVERS column that matches the value in the CURRENT SERVER special register.

```
SELECT LA_SERVERS FROM CORP_SERVERS
WHERE CORP_SERVERS.LA_SERVERS LIKE CURRENT SERVER
```

- Retrieve all strings that begin with the character sequence '\ ' in column A of table T.

```
SELECT A FROM T
WHERE T.A LIKE '\_\' ESCAPE '\'
```

- Use the BLOB scalar function to obtain a one-byte escape character that is compatible with the match and pattern data types (both BLOBs).

```
SELECT COLBLOB FROM TABLET
WHERE COLBLOB LIKE :pattern_var ESCAPE BLOB(X'0E')
```

- In a Unicode database defined with the case insensitive collation UCA500R1_LEN_S1, find all names that start with 'Bill'.

```
SELECT NAME FROM CUSTDATA WHERE NAME LIKE 'Bi11%'
```

The will return the names 'Bill Smith', 'billy simon', and 'BILL JONES'.

NULL predicate

►► *expression* IS NOT NULL ◀◀

The NULL predicate tests for null values.

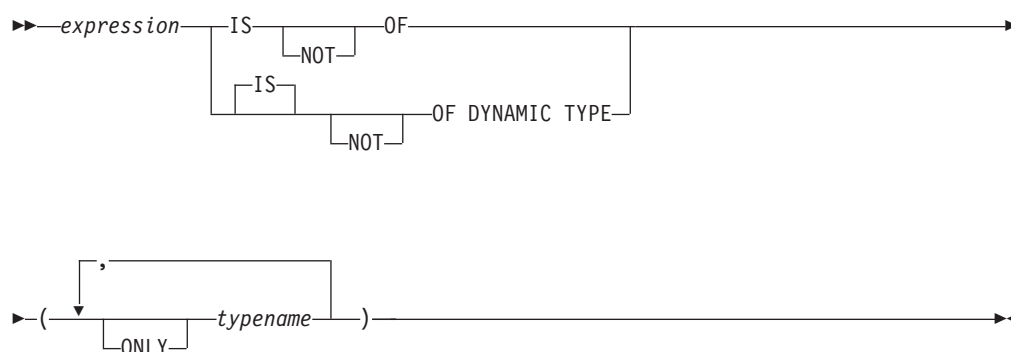
The result of a NULL predicate cannot be unknown. If the value of the expression is null, the result is true. If the value is not null, the result is false. If NOT is specified, the result is reversed.

Examples

PHONENO IS NULL

SALARY IS NOT NULL

TYPE predicate



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.

The syntax IS OF and OF DYNAMIC TYPE are equivalent alternatives for the TYPE predicate. Similarly, IS NOT OF and NOT OF DYNAMIC TYPE are equivalent alternatives.

Examples

A table hierarchy exists with root table EMPLOYEE of type EMP and subtable MANAGER of type MGR. Another table, ACTIVITIES, includes a column called

TYPE predicate

WHO_RESPONSIBLE that is defined as REF(EMP) SCOPE 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 (MGR)
```

If a table contains a column EMPLOYEE of type EMP, EMPLOYEE may contain values of type EMP as well as values of its subtypes like MGR. The following predicate

```
EMPL IS OF (MGR)
```

returns true when EMPL is not null and is actually a manager.

VALIDATED predicate

```
XML-expression IS [NOT] VALIDATED [according-to-clause]
```

according-to-clause:

```
ACCORDING TO XMLSCHEMA XML-schema-identification  
IN ( XML-schema-identification )
```

XML-schema-identification:

```
ID XML-schema-name  
URI XML-uri1  
NO NAMESPACE LOCATION XML-uri2
```

The VALIDATED predicate tests whether or not the value specified by *XML-expression* has been validated using the XMLVALIDATE function. If the value specified is null, the result of the validation constraint is unknown; otherwise, the result of the validation constraint is either true or false. The value you specify must be of type XML.

If the ACCORDING TO XMLSCHEMA clause is not specified, then XML schemas used for validation do not impact the result of the validation constraint.

Description

XML-expression

Specifies the XML value tested, where *XML-expression* can consist of an XML document, XML content, a sequence of XML nodes, an XML *column-name*, or an XML *correlation-name*.

If an XML *column-name* is specified, the predicate evaluates whether or not XML documents associated with the specified column name have been validated.

See "CREATE TRIGGER" for information about specifying correlation names of type XML as part of triggers.

IS VALIDATED or IS NOT VALIDATED

Specifies the required validation state for the *XML-expression* operand.

For a constraint that specifies IS VALIDATED to evaluate as true, the operand must have been validated. If an optional ACCORDING TO XMLSCHEMA clause includes one or several XML schemas, the operand must have been validated using one of the identified XML schemas.

For a constraint that specifies IS NOT VALIDATED to evaluate as false, the operand must be in an validated state. If an optional ACCORDING TO XMLSCHEMA clause includes one or several XML schemas, the operand must have been validated using one of the identified XML schemas.

according-to-clause

Specifies one or several XML schemas against which the operand must or must not have been validated. Only XML schemas previously registered with the XML schema repository may be specified.

ACCORDING TO XMLSCHEMA

ID *XML-schema-name*

Specifies an SQL identifier for the XML schema. The name, including the implicit or explicit SQL schema qualifier, must uniquely identify an existing XML schema in the XML schema repository at the current server. If no XML schema by this name exists in the implicitly or explicitly specified SQL schema, an error is returned (SQLSTATE 42704).

URI *XML-uri1*

Specifies the target namespace URI of the XML schema. The value of *XML-uri1* specifies a URI as a character string constant that is not empty. The URI must be the target namespace of a registered XML schema (SQLSTATE 4274A) and, if no LOCATION clause is specified, it must uniquely identify the registered XML schema (SQLSTATE 4274B).

NO NAMESPACE

Specifies that the XML schema has no target namespace. The target namespace URI is equivalent to an empty character string that cannot be specified as an explicit target namespace URI.

LOCATION *XML-uri2*

Specifies the XML schema location URI of the XML schema. The value of *XML-uri2* specifies a URI as a character string constant that is not empty. The XML schema location URI, combined with the target namespace URI, must identify a registered XML schema (SQLSTATE 4274A), and there must be only one such XML schema registered (SQLSTATE 4274B).

Examples

Example 1: Assume that column XMLCOL is defined in table T1. Retrieve only the XML values that have been validated by any XML schema.

```
SELECT XMLCOL FROM T1
WHERE XMLCOL IS VALIDATED
```

Example 2: Assume that column XMLCOL is defined in table T1. Enforce the rule that values cannot be inserted or updated unless they have been validated.

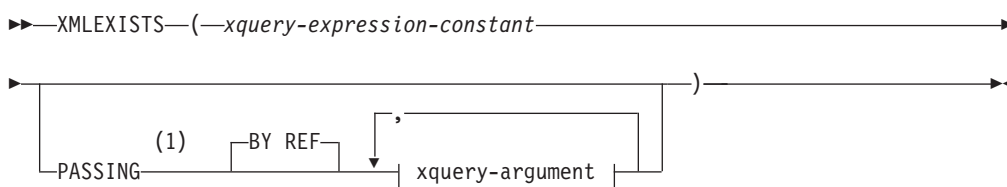
```
ALTER TABLE T1 ADD CONSTRAINT CK_VALIDATED
CHECK (XMLCOL IS VALIDATED)
```

VALIDATED predicate

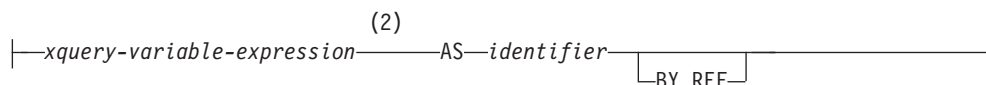
Example 3: Assume that you want to select only those rows from table T1 with XML column XMLCOL that have been validated with the XML schema URI `http://www.posample.org`.

```
SELECT XMLCOL FROM T1
WHERE XMLCOL IS VALIDATED
ACCORDING TO XMLSCHEMA URI
'http://www.posample.org'
```

XMLEXISTS predicate



xquery-argument:



Notes:

- 1 The data type cannot be DECFLOAT.
- 2 The data type of the expression cannot be DECFLOAT.

The XMLEXISTS predicate tests whether an XQuery expression returns a sequence of one or more items.

xquery-expression-constant

Specifies an SQL character string constant that is interpreted as an XQuery expression. The constant string is converted directly to UTF-8 without conversion to the database or session code page. The XQuery expression executes using an optional set of input XML values, and returns an output sequence that is tested to determine the result of the XMLEXISTS predicate. The value for *xquery-expression-constant* must not be an empty string or a string of blank characters (SQLSTATE 10505).

PASSING

Specifies input values and the manner in which these values are passed to the XQuery expression specified by *xquery-expression-constant*. By default, every unique column name that is in the scope where the function is invoked is implicitly passed to the XQuery expression using the name of the column as the variable name. If an *identifier* in a specified *xquery-argument* matches an in-scope column name, then the explicit *xquery-argument* is passed to the XQuery expression overriding that implicit column.

BY REF

Specifies that the default passing mechanism is by reference for any *xquery-variable-expression* of data type XML. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any, directly from the specified input expressions, preserving all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and

document ordering comparisons involving some nodes contained between the two input arguments might refer to nodes within the same XML node tree.

This clause has no impact on how non-XML values are passed. The non-XML values create a new copy of the value during the cast to XML.

xquery-argument

Specifies an argument that is to be passed to the XQuery expression specified by *xquery-expression-constant*. An argument specifies a value and the manner in which that value is to be passed. The argument includes an SQL expression that is evaluated.

- If the resulting value is of type XML, it becomes an *input-xml-value*. A null XML value is converted to an XML empty sequence.
- If the resulting value is not of type XML, it must be castable to the XML data type. A null value is converted to an XML empty sequence. The converted value becomes an *input-xml-value*.

When the *xquery-expression-constant* is evaluated, an XQuery variable is presented with a value equal to *input-xml-value* and a name specified by the AS clause.

xquery-variable-expression

Specifies an SQL expression whose value is available to the XQuery expression specified by *xquery-expression-constant* during execution. The expression cannot contain a sequence reference (SQLSTATE 428F9) or an OLAP function (SQLSTATE 42903). The data type of the expression cannot be DECFLOAT.

AS identifier

Specifies that the value generated by *xquery-variable-expression* will be passed to *xquery-expression-constant* as an XQuery variable. The variable name will be *identifier*. The leading dollar sign (\$) that precedes variable names in the XQuery language is not included in *identifier*. The identifier must be a valid XQuery variable name and is restricted to an XML NCName. The identifier must not be greater than 128 bytes in length. Two arguments within the same PASSING clause cannot use the same identifier (SQLSTATE 42711).

BY REF

Indicates that an XML input value is to be passed by reference. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any, directly from the specified input expressions, preserving all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons involving some nodes contained between the two input arguments might refer to nodes within the same XML node tree. If BY REF is not specified following an *xquery-variable-expression*, XML arguments are passed by way of the default passing mechanism that is provided through the syntax that follows the PASSING keyword. This option cannot be specified for non-XML values. When a non-XML value is passed, the value is converted to XML; this process creates a copy.

Notes

The XMLEXISTS predicate cannot be:

XMLEXISTS predicate

- Part of the ON clause that is associated with a JOIN operator or a MERGE statement (SQLSTATE 42972)
- Part of the GENERATE KEY USING or RANGE THROUGH clause in the CREATE INDEX EXTENSION statement (SQLSTATE 428E3)
- Part of the FILTER USING clause in the CREATE FUNCTION (External Scalar) statement, or the FILTER USING clause in the CREATE INDEX EXTENSION statement (SQLSTATE 428E4)
- Part of a check constraint or a column generation expression (SQLSTATE 42621)
- Part of a group-by-clause (SQLSTATE 42822)
- Part of an argument for a column-function (SQLSTATE 42607)

An XMLEXISTS predicate that involves a subquery might be restricted by statements that restrict subqueries.

The XMLEXISTS predicate can be used only in a database with a single database partition (SQLSTATE 42997).

Example

```
SELECT c.cid FROM customer c
WHERE XMLEXISTS('$d/*:customerinfo/*:addr[ *:city = "Aurora" ]'
PASSING info AS "d")
```

Chapter 3. Functions

Functions overview

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).

Built-in functions are provided with the database manager; they return a single result value, and are identified as part of the SYSIBM schema. Built-in 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 registered to a database in SYSCAT.ROUTINES (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, and another in a schema called SYSPROC.

Functions are classified as aggregate (column) functions, scalar functions, row functions, or table functions.

- The argument of an *aggregate function* is a collection of like values. An aggregate function returns a single value (possibly null), and can be specified in an SQL statement wherever an expression can be used.
- The arguments of a *scalar function* are individual scalar values, which can be of different types and have different meanings. A scalar function returns a single value (possibly null), and can be specified in an SQL statement wherever an expression can be used.
- The argument of a *row function* is a structured type. A row function returns a row of built-in data types and can only be specified as a transform function for a structured type.
- The arguments of a *table function* are individual scalar values, which can be of different types and have different meanings. A table function returns a table to the SQL statement, and can be specified only within the FROM clause of a SELECT statement.

The function name, combined with the schema, gives the fully qualified name of a function. The combination of schema, function name, and input parameters make up a *function signature*.

In some cases, the input parameter type is specified as a specific built-in data type, and in other cases, it is specified through a general variable like *any-numeric-type*. If a particular data type is specified, an exact match will only occur with the specified data type. If a general variable is used, each of the data types associated with that variable results in an exact match.

Additional functions may be available, because user-defined functions can be created in different schemas, using one of the function signatures as a source. You can also create external functions in your applications.

Supported functions and administrative SQL routines and views

Table 21 summarizes information about the supported functions. The function name, combined with the schema, gives the fully qualified name of a function. The “Input” column shows the expected data type 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 makes up a function signature. The “Returns” column shows the possible data types of values returned by the function. For the administrative SQL routines and views, refer to the reference information for the input and return information.

For lists of the supported built-in functions classified by type, see the following tables:

- Aggregate functions (Table 22 on page 229)
- Cast scalar functions (Table 23 on page 230)
- Partitioning scalar functions (Table 24 on page 230)
- Datetime scalar functions (Table 25 on page 231)
- Numeric scalar functions (Table 26 on page 232)
- Security scalar functions (Table 27 on page 233)
- XML functions (Table 28 on page 233)
- String scalar functions (Table 29 on page 234)
- Miscellaneous scalar functions (Table 30 on page 236)

For lists of the supported administrative SQL routines and views classified by functionality, see “Supported administrative SQL routines and views” in *Administrative Routines and Views* . These routines and views are grouped as follows:

- Activity monitor administrative SQL routines
- ADMIN_CMD stored procedure and associated administrative SQL routines
- Configuration administrative SQL routines and views
- Environment administrative views
- Health snapshot administrative SQL routines
- MQSeries® administrative SQL routines
- Security administrative SQL routines and views
- Snapshot administrative SQL routines and views
- SQL procedures administrative SQL routines
- Stepwise redistribute administrative SQL routines
- Storage management tool administrative SQL routines
- Miscellaneous administrative SQL routines and views

Table 21. Supported functions

Function name	Schema	Input	Returns	Description
“ABS or ABSVAL” on page 257	SYSIBM	Any expression that returns a built-in numeric data type.	Same data type and length as the argument.	This scalar function returns the absolute value of the argument.
“ABS or ABSVAL” on page 257	SYSFUN	<ul style="list-style-type: none"> • SMALLINT • INTEGER • BIGINT • DOUBLE 	<ul style="list-style-type: none"> • SMALLINT • INTEGER • BIGINT • DOUBLE 	This scalar function returns the absolute value of the argument.

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description
"ACOS" on page 257	SYSIBM	• DOUBLE	• DOUBLE	This scalar function returns the arccosine of the argument as an angle expressed in radians.
"ARRAY_AGG" on page 238	SYSIBM	• <i>any-type</i> ⁸ • <i>any-type</i> ⁸ , <i>any-type</i>	• an array • an array	This aggregate function aggregates a set of elements into an array.
"ASCII" on page 258	SYSFUN	• CHAR • VARCHAR(4000) • CLOB(1M)	• INTEGER • INTEGER • INTEGER	This scalar function returns the ASCII code value of the leftmost character of the argument as an integer.
"ASIN" on page 258	SYSIBM	• DOUBLE	• DOUBLE	This scalar function returns the arcsine of the argument as an angle expressed in radians.
"ATAN" on page 259	SYSIBM	• DOUBLE	• DOUBLE	This scalar function returns the arctangent of the argument as an angle expressed in radians.
"ATANH" on page 259	SYSIBM	• DOUBLE	• DOUBLE	This scalar function returns the hyperbolic arctangent of the argument, where the argument is an angle expressed in radians.
"ATAN2" on page 259	SYSIBM	• DOUBLE, DOUBLE	• DOUBLE	This scalar function returns the arctangent of <i>x</i> and <i>y</i> coordinates — specified by the first and second arguments — respectively, as an angle expressed in radians.
"AVG" on page 239	SYSIBM	• <i>numeric-type</i> ⁴	• <i>numeric-type</i> ¹	This aggregate function returns the average of a set of numbers.
"BIGINT" on page 260	SYSIBM	• <i>numeric-type</i> • VARCHAR	• BIGINT • BIGINT	This scalar function returns a 64-bit integer representation of a number or a character string in the form of an integer constant.
"BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT" on page 261	SYSIBM	• See the complete description of these functions.	• See the complete description of these functions.	These bitwise functions operate on the "two's complement" representation of the integer value of the input arguments and return the result as a corresponding base 10 integer value in a data type based on the data type of the input arguments.
"BLOB" on page 263	SYSIBM	• <i>string-type</i> • <i>string-type</i> , INTEGER	• BLOB • BLOB	This scalar function casts from source type to BLOB, with optional length.
"CARDINALITY" on page 263	SYSIBM	• an array	• BIGINT	This scalar function returns the number of elements of an array
"CEILING" on page 264	SYSIBM	• SMALLINT • INTEGER • BIGINT • DOUBLE • DECFLOAT(<i>n</i>)	• SMALLINT • INTEGER • BIGINT • DOUBLE • DECFLOAT(<i>n</i>)	This scalar function returns the smallest integer that is greater than or equal to the argument.
"CHAR" on page 264	SYSIBM	• <i>character-type</i> • <i>character-type</i> , INTEGER • <i>datetime-type</i> • <i>datetime-type</i> , <i>keyword</i> ² • SMALLINT • INTEGER • BIGINT • DECIMAL • DECIMAL, VARCHAR • DECFLOAT(<i>n</i>)	• CHAR • CHAR(<i>integer</i>) • CHAR • CHAR • CHAR(6) • CHAR(11) • CHAR(20) • CHAR(2+ <i>precision</i>) • CHAR(2+ <i>precision</i>) • CHAR(42)	This scalar function returns a string representation of the source type.
"CHAR" on page 264	SYSFUN	• DOUBLE	• CHAR(24)	This scalar function returns a character string representation of a floating-point number.
"CHARACTER_LENGTH" on page 268	SYSIBM	• <i>string-type</i> , <i>string-unit</i>	• INTEGER	This scalar function returns the length of an expression in the specified <i>string-unit</i> .

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description
"CHR" on page 270	SYSFUN	<ul style="list-style-type: none"> INTEGER 	<ul style="list-style-type: none"> CHAR(1) 	This scalar function 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.
"CLOB" on page 270	SYSIBM	<ul style="list-style-type: none"> character-type character-type, INTEGER 	<ul style="list-style-type: none"> CLOB CLOB 	This scalar function casts from source type to CLOB, with optional length.
"COALESCE" on page 271 ³	SYSIBM	<ul style="list-style-type: none"> any-type, any-union-compatible-type,... 	<ul style="list-style-type: none"> any-type 	This scalar function returns the first non-null argument in the set of arguments.
"COLLATION_KEY_BIT" on page 271	SYSIBM	<ul style="list-style-type: none"> CHAR, a string that specifies a collation VARCHAR, a string that specifies a collation GRAPHIC, a string that specifies a collation VARGRAPHIC, a string that specifies a collation CHAR, a string that specifies a collation, INTEGER VARCHAR, a string that specifies a collation, INTEGER GRAPHIC, a string that specifies a collation, INTEGER VARGRAPHIC, a string that specifies a collation, INTEGER 	<ul style="list-style-type: none"> VARCHAR FOR BIT DATA VARCHAR FOR BIT DATA VARCHAR FOR BIT DATA VARCHAR FOR BIT DATA VARCHAR FOR BIT DATA VARCHAR FOR BIT DATA VARCHAR FOR BIT DATA VARCHAR FOR BIT DATA 	This scalar function returns a string representing the collation key of the specified <i>string-expression</i> in the specified <i>collation-name</i> .
"COMPARE_DECFLOAT" on page 273	SYSIBM	<ul style="list-style-type: none"> DECFLOAT(<i>n</i>), DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> INTEGER 	This scalar function returns a SMALLINT value that indicates whether the two arguments are equal or unordered, or whether one argument is greater than the other.
"CONCAT" on page 274	SYSIBM	<ul style="list-style-type: none"> string-type, compatible-string-type 	<ul style="list-style-type: none"> max-string-type 	This scalar function returns the concatenation of two string arguments.
"CORRELATION" on page 240	SYSIBM	<ul style="list-style-type: none"> numeric-type, numeric-type non-decfloat-numeric-type, non-decfloat-numeric-type DECFLOAT(<i>n</i>), numeric-type numeric-type, DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> DOUBLE DOUBLE DECFLOAT(34) DECFLOAT(34) 	This aggregate function returns the coefficient of correlation of a set of number pairs.
"COS" on page 274	SYSIBM	<ul style="list-style-type: none"> DOUBLE 	<ul style="list-style-type: none"> DOUBLE 	This scalar function returns the cosine of the argument, where the argument is an angle expressed in radians.
"COSH" on page 274	SYSIBM	<ul style="list-style-type: none"> DOUBLE 	<ul style="list-style-type: none"> DOUBLE 	This scalar function returns the hyperbolic cosine of the argument, where the argument is an angle expressed in radians.
"COT" on page 275	SYSIBM	<ul style="list-style-type: none"> DOUBLE 	<ul style="list-style-type: none"> DOUBLE 	This scalar function returns the cotangent of the argument, where the argument is an angle expressed in radians.
"COUNT" on page 241	SYSIBM	<ul style="list-style-type: none"> any-builtin-type⁴ 	<ul style="list-style-type: none"> INTEGER 	This aggregate function returns the number of rows or values in a set of rows or values.
"COUNT_BIG" on page 242	SYSIBM	<ul style="list-style-type: none"> any-builtin-type⁴ 	<ul style="list-style-type: none"> DECIMAL(31,0) 	This aggregate function returns the number of rows or values in a set of rows or values. The result can be greater than the maximum value of INTEGER.

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description
"COVARIANCE" on page 243	SYSIBM	<ul style="list-style-type: none"> • <i>numeric-type, numeric-type</i> • <i>non-decfloat-numeric-type, non-decfloat-numeric-type</i> • DECFLOAT(<i>n</i>), <i>numeric-type</i> • <i>numeric-type, DECFLOAT(<i>n</i>)</i> 	<ul style="list-style-type: none"> • DOUBLE • DOUBLE • DECFLOAT(34) • DECFLOAT(34) 	This aggregate function returns the covariance of a set of number pairs.
"DATAPARTITIONNUM" on page 275	SYSIBM	<ul style="list-style-type: none"> • <i>any-type</i> 	<ul style="list-style-type: none"> • INTEGER 	This scalar function returns the sequence number (SYSDATAPARTITIONS.SEQNO) of the data partition in which the row resides. The argument is any column name within the table.
"DATE" on page 276	SYSIBM	<ul style="list-style-type: none"> • DATE • TIMESTAMP • DOUBLE • VARCHAR • DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> • DATE • DATE • DATE • DATE • DATE 	This scalar function returns a date from a single input value.
"DAY" on page 277	SYSIBM	<ul style="list-style-type: none"> • VARCHAR • DATE • TIMESTAMP • DECIMAL 	<ul style="list-style-type: none"> • INTEGER • INTEGER • INTEGER • INTEGER 	This scalar function returns the day part of a value.
"DAYNAME" on page 278	SYSFUN	<ul style="list-style-type: none"> • VARCHAR(26) • DATE • TIMESTAMP 	<ul style="list-style-type: none"> • VARCHAR(100) • VARCHAR(100) • VARCHAR(100) 	This scalar function returns a mixed case character string containing the name of the day (for example, Friday) for the day portion of the argument, based on what the locale was when db2start was issued.
"DAYOFWEEK" on page 278	SYSFUN	<ul style="list-style-type: none"> • VARCHAR(26) • DATE • TIMESTAMP 	<ul style="list-style-type: none"> • INTEGER • INTEGER • INTEGER 	This scalar function returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Sunday.
"DAYOFWEEK_ISO" on page 278	SYSFUN	<ul style="list-style-type: none"> • VARCHAR(26) • DATE • TIMESTAMP 	<ul style="list-style-type: none"> • INTEGER • INTEGER • INTEGER 	This scalar function returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Monday.
"DAYOFYEAR" on page 279	SYSFUN	<ul style="list-style-type: none"> • VARCHAR(26) • DATE • TIMESTAMP 	<ul style="list-style-type: none"> • INTEGER • INTEGER • INTEGER 	This scalar function returns the day of the year in the argument as an integer value in the range 1-366.
"DAYS" on page 279	SYSIBM	<ul style="list-style-type: none"> • VARCHAR • TIMESTAMP • DATE 	<ul style="list-style-type: none"> • INTEGER • INTEGER • INTEGER 	This scalar function returns an integer representation of a date.
"DBCLOB" on page 280	SYSIBM	<ul style="list-style-type: none"> • <i>graphic-type</i> • <i>graphic-type, INTEGER</i> 	<ul style="list-style-type: none"> • DBCLOB • DBCLOB 	This scalar function casts from source type to DBCLOB, with optional length.
"DBPARTITIONNUM" on page 280 ³	SYSIBM	<ul style="list-style-type: none"> • <i>any-type</i> 	<ul style="list-style-type: none"> • INTEGER 	This scalar function returns the database partition number of the row. The argument is any column name within the table.
"DECFLOAT" on page 282	SYSIBM	<ul style="list-style-type: none"> • <i>numeric-type</i> • <i>numeric-type, INTEGER</i> 	<ul style="list-style-type: none"> • DECFLOAT(34) • DECFLOAT(<i>n</i>), where <i>n</i> is the (optional) second argument of type INTEGER, either 16 or 34 	This scalar function returns the decimal floating-point representation of a number with optional precision.

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description
"DECFLOAT" on page 282	SYSIBM	<ul style="list-style-type: none"> • VARCHAR • VARCHAR, INTEGER • VARCHAR, INTEGER, VARCHAR 	<ul style="list-style-type: none"> • DECFLOAT(34) • DECFLOAT(<i>n</i>), where <i>n</i> is the (optional) second argument of type INTEGER, either 16 or 34 • DECFLOAT(<i>n</i>), where <i>n</i> is the (optional) second argument of type INTEGER, either 16 or 34 	This scalar function returns the decimal floating-point representation of a number with optional precision and decimal character.
"DECIMAL" on page 283	SYSIBM	<ul style="list-style-type: none"> • <i>numeric-type</i> • <i>numeric-type</i>, INTEGER • <i>numeric-type</i> INTEGER, INTEGER 	<ul style="list-style-type: none"> • DECIMAL • DECIMAL • DECIMAL 	This scalar function returns the decimal representation of a number, with optional precision and scale.
"DECIMAL" on page 283	SYSIBM	<ul style="list-style-type: none"> • VARCHAR • VARCHAR, INTEGER • VARCHAR, INTEGER, INTEGER • VARCHAR, INTEGER, INTEGER, VARCHAR 	<ul style="list-style-type: none"> • DECIMAL • DECIMAL • DECIMAL • DECIMAL 	This scalar function returns the decimal representation of a character string, with optional precision, scale, and decimal character.
"DECODE" on page 286	SYSIBM	<ul style="list-style-type: none"> • <i>any-type</i> 	<ul style="list-style-type: none"> • <i>any-type</i> 	This scalar function compares each specified <i>expression2</i> to <i>expression1</i> . If <i>expression1</i> is equal to <i>expression2</i> , or both <i>expression1</i> and <i>expression2</i> are null, the value of the following <i>result-expression</i> is returned. If no <i>expression2</i> matches <i>expression1</i> , the value of <i>else-expression</i> is returned; otherwise a null value is returned.
"DECRYPT_BIN and DECRYPT_CHAR" on page 287	SYSIBM	<ul style="list-style-type: none"> • VARCHAR FOR BIT DATA • VARCHAR FOR BIT DATA, VARCHAR 	<ul style="list-style-type: none"> • VARCHAR FOR BIT DATA • VARCHAR FOR BIT DATA 	This scalar function returns a value that is the result of decrypting encrypted data using a password string.
"DECRYPT_BIN and DECRYPT_CHAR" on page 287	SYSIBM	<ul style="list-style-type: none"> • VARCHAR FOR BIT DATA • VARCHAR FOR BIT DATA, VARCHAR 	<ul style="list-style-type: none"> • VARCHAR • VARCHAR 	This scalar function returns a value that is the result of decrypting encrypted data using a password string.
"DEGREES" on page 288	SYSIBM	<ul style="list-style-type: none"> • DOUBLE • DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> • DOUBLE • DECFLOAT(<i>n</i>) 	This scalar function returns the number of degrees converted from the argument expressed in radians.
"DEGREES" on page 288	SYSFUN	<ul style="list-style-type: none"> • DOUBLE 	<ul style="list-style-type: none"> • DOUBLE 	This scalar function returns the number of degrees converted from the argument expressed in radians.
"DEREF" on page 289	SYSIBM	<ul style="list-style-type: none"> • REF(<i>any-structured-type</i>) with defined scope 	<ul style="list-style-type: none"> • <i>any-structured-type</i> (same as input target type) 	This scalar function returns an instance of the target type of the reference type argument.
"DIFFERENCE" on page 289	SYSFUN	<ul style="list-style-type: none"> • VARCHAR(4000), VARCHAR(4000) 	<ul style="list-style-type: none"> • INTEGER 	This scalar function returns the difference between the sounds of the words in the two argument strings, as determined by the SOUNDEX function. A value of 4 means the strings sound the same.
"DIGITS" on page 290	SYSIBM	<ul style="list-style-type: none"> • DECIMAL 	<ul style="list-style-type: none"> • CHAR 	This scalar function returns the character string representation of a number.
"DOUBLE" on page 291	SYSIBM	<ul style="list-style-type: none"> • <i>numeric-type</i> 	<ul style="list-style-type: none"> • DOUBLE 	This scalar function returns the floating-point representation of a number.
"DOUBLE" on page 291	SYSFUN	<ul style="list-style-type: none"> • VARCHAR 	<ul style="list-style-type: none"> • DOUBLE 	This scalar function returns the floating-point number corresponding to the character string representation of a number. Leading and trailing blanks in <i>argument</i> are ignored.

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description
EMPTY_BLOB, EMPTY_CLOB, and EMPTY_DBCLOB scalar functions	SYSIBM	<ul style="list-style-type: none"> no argument 	<ul style="list-style-type: none"> BLOB 	The EMPTY_BLOB scalar function returns a zero-length value.
EMPTY_BLOB, EMPTY_CLOB, and EMPTY_DBCLOB scalar functions	SYSIBM	<ul style="list-style-type: none"> no argument 	<ul style="list-style-type: none"> CLOB 	The EMPTY_CLOB scalar function returns a zero-length value.
EMPTY_BLOB, EMPTY_CLOB, and EMPTY_DBCLOB scalar functions	SYSIBM	<ul style="list-style-type: none"> no argument 	<ul style="list-style-type: none"> DBCLOB 	The EMPTY_DBCLOB scalar function returns a zero-length value.
"ENCRYPT" on page 292	SYSIBM	<ul style="list-style-type: none"> VARCHAR VARCHAR, VARCHAR VARCHAR, VARCHAR, VARCHAR 	<ul style="list-style-type: none"> VARCHAR FOR BIT DATA VARCHAR FOR BIT DATA VARCHAR FOR BIT DATA 	This scalar function returns a value that is the result of encrypting a data string expression.
"EVENT_MON_STATE" on page 293	SYSIBM	<ul style="list-style-type: none"> VARCHAR 	<ul style="list-style-type: none"> INTEGER 	This scalar function returns the operational state of particular event monitor.
"EXP" on page 294	SYSIBM	<ul style="list-style-type: none"> DOUBLE DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> DOUBLE DECFLOAT(34) 	This scalar function returns the exponential function of the argument.
"EXP" on page 294	SYSFUN	<ul style="list-style-type: none"> DOUBLE 	<ul style="list-style-type: none"> DOUBLE 	This scalar function returns the exponential function of the argument.
"FLOAT" on page 294	SYSIBM			This scalar function is the same as DOUBLE.
"FLOOR" on page 295	SYSIBM	<ul style="list-style-type: none"> SMALLINT INTEGER BIGINT DOUBLE DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> SMALLINT INTEGER BIGINT DOUBLE DECFLOAT(<i>n</i>) 	This scalar function returns the largest integer that is less than or equal to the argument.
"GENERATE_UNIQUE" on page 295	SYSIBM	<ul style="list-style-type: none"> no argument 	<ul style="list-style-type: none"> CHAR(13) FOR BIT DATA 	This scalar function returns a bit data character string that is unique compared to any other execution of the same function.
"GETHINT" on page 297	SYSIBM	<ul style="list-style-type: none"> VARCHAR or CLOB 	<ul style="list-style-type: none"> VARCHAR 	This scalar function returns the password hint if one is found.
"GRAPHIC" on page 297	SYSIBM	<ul style="list-style-type: none"> <i>graphic-type</i> <i>graphic-type</i>, INTEGER 	<ul style="list-style-type: none"> GRAPHIC GRAPHIC 	This scalar function casts from source type to GRAPHIC, with optional length.
"GREATEST" on page 299	SYSIBM	<ul style="list-style-type: none"> <i>any-builtin-type</i>⁹ 	<ul style="list-style-type: none"> same as input type 	This scalar function returns the maximum value in a set of values.
"GROUPING" on page 244	SYSIBM	<ul style="list-style-type: none"> <i>any-type</i> 	<ul style="list-style-type: none"> SMALLINT 	This aggregate function is used with grouping-sets and super-groups to indicate sub-total rows generated by a grouping set. The value returned is 0 or 1. A value of 1 means that the value of the argument in the returned row is a null value, and the row was generated for a grouping set. This generated row provides a sub-total for a grouping set.
"HASHEDVALUE" on page 299 ³	SYSIBM	<ul style="list-style-type: none"> <i>any-type</i> 	<ul style="list-style-type: none"> INTEGER 	This scalar function returns the distribution map index (0 to 4095) of the row. The argument is a column name within a table.
"HEX" on page 300	SYSIBM	<ul style="list-style-type: none"> <i>any-builtin-type</i> 	<ul style="list-style-type: none"> VARCHAR 	This scalar function returns the hexadecimal representation of a value.

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description
"HOUR" on page 302	SYSIBM	<ul style="list-style-type: none"> • VARCHAR • TIME • TIMESTAMP • DECIMAL 	<ul style="list-style-type: none"> • INTEGER • INTEGER • INTEGER • INTEGER 	This scalar function returns the hour part of a value.
"IDENTITY_VAL_LOCAL" on page 302	SYSIBM		<ul style="list-style-type: none"> • DECIMAL 	This scalar function returns the most recently assigned value for an identity column.
"INSERT" on page 306	SYSIBM	<ul style="list-style-type: none"> • See the complete description of this function. 	<ul style="list-style-type: none"> • See the complete description of this function. 	This scalar function returns a string in which, beginning at start in the specified source-string, length of the specified code units have been deleted and insert-string has been inserted.
"INSERT" on page 306	SYSFUN	<ul style="list-style-type: none"> • VARCHAR(4000), INTEGER, INTEGER, VARCHAR(4000) • CLOB(1M), INTEGER, INTEGER, CLOB(1M) • BLOB(1M), INTEGER, INTEGER, BLOB(1M) 	<ul style="list-style-type: none"> • VARCHAR(4000) • CLOB(1M) • BLOB(1M) 	This scalar function returns a string, where <i>argument3</i> bytes have been deleted from <i>argument1</i> (beginning at <i>argument2</i>), and <i>argument4</i> has been inserted into <i>argument1</i> (beginning at <i>argument2</i>).
"INTEGER" on page 309	SYSIBM	<ul style="list-style-type: none"> • <i>numeric-type</i> • VARCHAR 	<ul style="list-style-type: none"> • INTEGER • INTEGER 	This scalar function returns the integer representation of a number.
"JULIAN_DAY" on page 310	SYSFUN	<ul style="list-style-type: none"> • VARCHAR(26) • DATE • TIMESTAMP 	<ul style="list-style-type: none"> • INTEGER • INTEGER • INTEGER 	This scalar function 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 <i>argument</i> .
"LCASE" on page 311	SYSIBM			This scalar function is the same as LOWER.
"LCASE (Locale sensitive)" on page 311	SYSIBM			This scalar function is the same as LOWER.
"LEAST" on page 311	SYSIBM	<ul style="list-style-type: none"> • <i>any-builtin-type</i>⁹ 	<ul style="list-style-type: none"> • same as input type 	This scalar function returns the minimum value in a set of values.
"LEFT" on page 312	SYSIBM	<ul style="list-style-type: none"> • CHAR or VARCHAR, INTEGER • CLOB, INTEGER • GRAPHIC or VARGRAPHIC, INTEGER • DBCLOB, INTEGER 	<ul style="list-style-type: none"> • VARCHAR • CLOB • VARGRAPHIC • DBCLOB 	This scalar function returns the leftmost string of string-expression of length expressed in the specified string unit. If string-expression is a character string, the result is a character string. If string-expression is a graphic string, the result is a graphic string.
"LEFT" on page 312	SYSFUN	<ul style="list-style-type: none"> • VARCHAR(4000), INTEGER • CLOB(1M), INTEGER • BLOB(1M), INTEGER 	<ul style="list-style-type: none"> • VARCHAR(4000) • CLOB(1M) • BLOB(1M) 	This scalar function returns a string consisting of the leftmost <i>argument2</i> bytes in <i>argument1</i> .
"LENGTH" on page 314	SYSIBM	<ul style="list-style-type: none"> • <i>any-builtin-type</i> • <i>any-builtin-type, string-unit</i> 	<ul style="list-style-type: none"> • INTEGER • INTEGER 	This scalar function returns the length of the operand in bytes (except for double-byte string types, which return the length in double-byte characters).
"LOWER" on page 321	SYSIBM	<ul style="list-style-type: none"> • CHAR • VARCHAR 	<ul style="list-style-type: none"> • CHAR • VARCHAR 	This scalar function returns a string in which all the characters have been converted to lowercase characters.
"LN" on page 316	SYSIBM	<ul style="list-style-type: none"> • DOUBLE • DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> • DOUBLE • DECFLOAT(<i>n</i>) 	This scalar function returns the natural logarithm of the argument (same as LOG).
"LN" on page 316	SYSFUN	<ul style="list-style-type: none"> • DOUBLE 	<ul style="list-style-type: none"> • DOUBLE 	This scalar function returns the natural logarithm of the argument (same as LOG).

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description
“LOCATE scalar function” on page 317	SYSIBM	<ul style="list-style-type: none"> • <i>string-type, compatible-string-type</i> • <i>string-type, compatible-string-type, INTEGER</i> • <i>string-type, compatible-string-type, string-unit</i> • <i>string-type, compatible-string-type, INTEGER, string-unit</i> 	<ul style="list-style-type: none"> • INTEGER • INTEGER • INTEGER • INTEGER 	This scalar function returns the starting position of the first occurrence of <i>argument1</i> within <i>argument2</i> . If the optional INTEGER argument is specified, it indicates the character position in <i>argument2</i> at which the search is to begin. If <i>argument1</i> is not found within <i>argument2</i> , the value 0 is returned.
“LOG10” on page 320	SYSIBM	<ul style="list-style-type: none"> • DOUBLE • DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> • DOUBLE • DECFLOAT(<i>n</i>) 	This scalar function returns the base 10 logarithm of the argument.
“LOG10” on page 320	SYSFUN	<ul style="list-style-type: none"> • DOUBLE 	<ul style="list-style-type: none"> • DOUBLE 	This scalar function returns the base 10 logarithm of the argument.
“LOWER (Locale sensitive)” on page 321	SYSIBM	<ul style="list-style-type: none"> • CHAR, a string that specifies a locale • VARCHAR, a string that specifies a locale • GRAPHIC, a string that specifies a locale • VARGRAPHIC, a string that specifies a locale • CHAR, a string that specifies a locale, INTEGER • VARCHAR, a string that specifies a locale, INTEGER • GRAPHIC, a string that specifies a locale, INTEGER • VARGRAPHIC, a string that specifies a locale, INTEGER 	<ul style="list-style-type: none"> • VARCHAR • VARCHAR • VARGRAPHIC • VARGRAPHIC • VARCHAR • VARCHAR • VARGRAPHIC • VARGRAPHIC 	This scalar function returns a string in which all characters have been converted to lowercase characters using the rules from the Unicode standard associated with the specified locale.
“LTRIM” on page 323	SYSIBM	<ul style="list-style-type: none"> • CHAR • VARCHAR • GRAPHIC • VARGRAPHIC 	<ul style="list-style-type: none"> • VARCHAR • VARCHAR • VARGRAPHIC • VARGRAPHIC 	This scalar function returns the characters of the argument with leading blanks removed.
“MAX” on page 245	SYSIBM	<ul style="list-style-type: none"> • <i>any-builtin-type</i>⁵ 	<ul style="list-style-type: none"> • same as input type 	This aggregate function returns the maximum value in a set of values.
“MAX” on page 324	SYSIBM	<ul style="list-style-type: none"> • <i>any-builtin-type</i>⁹ 	<ul style="list-style-type: none"> • same as input type 	This scalar function returns the maximum value in a set of values.
“MAX_CARDINALITY” on page 324	SYSIBM	<ul style="list-style-type: none"> • an array 	<ul style="list-style-type: none"> • BIGINT 	This scalar function returns the maximum number of elements that an array can contain.
“MICROSECOND” on page 325	SYSIBM	<ul style="list-style-type: none"> • VARCHAR • TIMESTAMP • DECIMAL 	<ul style="list-style-type: none"> • INTEGER • INTEGER • INTEGER 	This scalar function returns the microsecond (time-unit) part of a value.
“MIDNIGHT_SECONDS” on page 325	SYSFUN	<ul style="list-style-type: none"> • VARCHAR(26) • TIME • TIMESTAMP 	<ul style="list-style-type: none"> • INTEGER • INTEGER • INTEGER 	This scalar function 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 <i>argument</i> .
“MIN” on page 246	SYSIBM	<ul style="list-style-type: none"> • <i>any-builtin-type</i>⁵ 	<ul style="list-style-type: none"> • same as input type 	This aggregate function returns the minimum value in a set of values.
“MIN” on page 326	SYSIBM	<ul style="list-style-type: none"> • <i>any-builtin-type</i>⁹ 	<ul style="list-style-type: none"> • same as input type 	This scalar function returns the minimum value in a set of values.

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description
"MINUTE" on page 327	SYSIBM	<ul style="list-style-type: none"> • VARCHAR • TIME • TIMESTAMP • DECIMAL 	<ul style="list-style-type: none"> • INTEGER • INTEGER • INTEGER • INTEGER 	This scalar function returns the minute part of a value.
"MOD" on page 328	SYSFUN	<ul style="list-style-type: none"> • SMALLINT, SMALLINT • INTEGER, INTEGER • BIGINT, BIGINT 	<ul style="list-style-type: none"> • SMALLINT • INTEGER • BIGINT 	This scalar function returns the remainder (modulus) of <i>argument1</i> divided by <i>argument2</i> . The result is negative only if <i>argument1</i> is negative.
"MONTH" on page 328	SYSIBM	<ul style="list-style-type: none"> • VARCHAR • DATE • TIMESTAMP • DECIMAL 	<ul style="list-style-type: none"> • INTEGER • INTEGER • INTEGER • INTEGER 	This scalar function returns the month part of a value.
"MONTHNAME" on page 329	SYSFUN	<ul style="list-style-type: none"> • VARCHAR(26) • DATE • TIMESTAMP 	<ul style="list-style-type: none"> • VARCHAR(100) • VARCHAR(100) • VARCHAR(100) 	This scalar function returns a mixed case character string containing the name of the month (for example, January) for the month portion of the argument that is a date or a timestamp, based on what the locale was when the database was started.
"MULTIPLY_ALT" on page 329	SYSIBM	<ul style="list-style-type: none"> • <i>exact-numeric-type</i>, • <i>exact-numeric-type</i> 	<ul style="list-style-type: none"> • DECIMAL 	This scalar function returns the product of two arguments as a decimal value. This function is useful when the sum of the argument precisions is greater than 31.
"NORMALIZE_DECFLOAT" on page 330	SYSIBM	<ul style="list-style-type: none"> • DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> • DECFLOAT(<i>n</i>) 	This scalar function returns a decimal floating-point value that is the result of the argument set to its simplest form.
"NULLIF" on page 331 ³	SYSIBM	<ul style="list-style-type: none"> • <i>any-type</i>⁵, • <i>any-comparable-type</i>⁵ 	<ul style="list-style-type: none"> • <i>any-type</i> 	This scalar function returns NULL if the arguments are equal, or returns the first argument if they are not equal.
"NVL" on page 331	SYSIBM	<ul style="list-style-type: none"> • <i>any-type</i>, • <i>any-union-compatible-type</i>,... 	<ul style="list-style-type: none"> • <i>any-type</i> 	This scalar function returns the first argument that is not null.
"OCTET_LENGTH" on page 332	SYSIBM	<ul style="list-style-type: none"> • <i>string-type</i> 	<ul style="list-style-type: none"> • INTEGER 	This scalar function returns the length of an expression in octets (bytes).
"OVERLAY" on page 332	SYSIBM	<ul style="list-style-type: none"> • See the complete description of this function. 	<ul style="list-style-type: none"> • See the complete description of this function. 	This scalar function returns a string in which, beginning at <i>start</i> in the specified <i>source-string</i> , <i>length</i> of the specified code units have been deleted and <i>insert-string</i> has been inserted.
"PARAMETER" on page 336	SYSIBM	<ul style="list-style-type: none"> • INTEGER 	<ul style="list-style-type: none"> • See the complete description of this function. 	This scalar function represents a position in an SQL statement where the value is provided dynamically by XQuery as part of the invocation of the db2-fn:sqlquery function.
"POSITION scalar function" on page 336	SYSIBM	<ul style="list-style-type: none"> • <i>string-type</i>, <i>string-type</i>, • <i>string-unit</i> 	<ul style="list-style-type: none"> • INTEGER 	This scalar function returns the starting position of <i>argument2</i> within <i>argument1</i> .
"POSSTR scalar function" on page 339	SYSIBM	<ul style="list-style-type: none"> • <i>string-type</i>, • <i>compatible-string-type</i> 	<ul style="list-style-type: none"> • INTEGER 	This scalar function returns the position at which one string is contained in another.
"POWER" on page 341	SYSIBM	<ul style="list-style-type: none"> • INTEGER, INTEGER • BIGINT, BIGINT • DOUBLE, INTEGER • DOUBLE, DOUBLE • DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> • INTEGER • BIGINT • DOUBLE • DOUBLE • DECFLOAT(34) 	This scalar function returns the value of <i>argument1</i> to the power of <i>argument2</i> .
"POWER" on page 341	SYSFUN	<ul style="list-style-type: none"> • INTEGER, INTEGER • BIGINT, BIGINT • DOUBLE, INTEGER • DOUBLE, DOUBLE 	<ul style="list-style-type: none"> • INTEGER • BIGINT • DOUBLE • DOUBLE 	This scalar function returns the value of <i>argument1</i> to the power of <i>argument2</i> .

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description
"QUANTIZE" on page 342	SYSIBM	<ul style="list-style-type: none"> numeric-type, numeric-type 	<ul style="list-style-type: none"> DECFLOAT(<i>n</i>), where <i>n</i> is 16 if both arguments are DECFLOAT(16); otherwise DECFLOAT(34) 	This scalar function returns a decimal floating-point number that is equal in value and sign to the first argument, and whose exponent is equal to the exponent of the second argument.
"QUARTER" on page 343	SYSFUN	<ul style="list-style-type: none"> VARCHAR(26) DATE TIMESTAMP 	<ul style="list-style-type: none"> INTEGER INTEGER INTEGER 	This scalar function returns an integer value in the range 1 to 4, representing the quarter of the year for the date specified in the argument.
"RADIANS" on page 343	SYSIBM	<ul style="list-style-type: none"> DOUBLE DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> DOUBLE DECFLOAT(<i>n</i>) 	This scalar function returns the number of radians converted from the argument, which is expressed in degrees.
"RADIANS" on page 343	SYSFUN	<ul style="list-style-type: none"> DOUBLE 	<ul style="list-style-type: none"> DOUBLE 	This scalar function returns the number of radians converted from the argument, which is expressed in degrees.
"RAISE_ERROR" on page 344 ³	SYSIBM	<ul style="list-style-type: none"> VARCHAR, VARCHAR 	<ul style="list-style-type: none"> any-type⁶ 	This scalar function raises an error in the SQLCA. The sqlstate that is to be returned is indicated by <i>argument1</i> . The second argument contains any text that is to be returned.
"RAND" on page 345	SYSFUN	<ul style="list-style-type: none"> no argument INTEGER 	<ul style="list-style-type: none"> DOUBLE DOUBLE 	This scalar function returns a random floating point value between 0 and 1, using the argument as an optional seed value.
"REAL" on page 345	SYSIBM	<ul style="list-style-type: none"> numeric-type 	<ul style="list-style-type: none"> REAL 	This scalar function returns the single-precision floating-point representation of a number.
"REC2XML" on page 346	SYSIBM	<ul style="list-style-type: none"> DECIMAL, VARCHAR, VARCHAR, any-type⁷ 	<ul style="list-style-type: none"> VARCHAR 	This scalar function returns a string formatted with XML tags, containing column names and column data.
"Regression functions" on page 247	SYSIBM	<ul style="list-style-type: none"> non-decfloat-numeric-type, non-decfloat-numeric-type DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> DOUBLE DECFLOAT(<i>n</i>) 	The REGR_AVGX aggregate function returns quantities used to compute diagnostic statistics.
"Regression functions" on page 247	SYSIBM	<ul style="list-style-type: none"> non-decfloat-numeric-type, non-decfloat-numeric-type DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> DOUBLE DECFLOAT(<i>n</i>) 	The REGR_AVGY aggregate function returns quantities used to compute diagnostic statistics.
"Regression functions" on page 247	SYSIBM	<ul style="list-style-type: none"> non-decfloat-numeric-type, non-decfloat-numeric-type DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> INTEGER DECFLOAT(<i>n</i>) 	The REGR_COUNT aggregate function returns the number of non-null number pairs used to fit the regression line.
"Regression functions" on page 247	SYSIBM	<ul style="list-style-type: none"> non-decfloat-numeric-type, non-decfloat-numeric-type DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> DOUBLE DECFLOAT(<i>n</i>) 	The REGR_INTERCEPT or REGR_ICPT aggregate function returns the y-intercept of the regression line.
"Regression functions" on page 247	SYSIBM	<ul style="list-style-type: none"> non-decfloat-numeric-type, non-decfloat-numeric-type DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> DOUBLE DECFLOAT(<i>n</i>) 	The REGR_R2 aggregate function returns the coefficient of determination for the regression.
"Regression functions" on page 247	SYSIBM	<ul style="list-style-type: none"> non-decfloat-numeric-type, non-decfloat-numeric-type DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> DOUBLE DECFLOAT(<i>n</i>) 	The REGR_SLOPE aggregate function returns the slope of the line.
"Regression functions" on page 247	SYSIBM	<ul style="list-style-type: none"> non-decfloat-numeric-type, non-decfloat-numeric-type DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> DOUBLE DECFLOAT(<i>n</i>) 	The REGR_SXX aggregate function returns quantities used to compute diagnostic statistics.
"Regression functions" on page 247	SYSIBM	<ul style="list-style-type: none"> non-decfloat-numeric-type, non-decfloat-numeric-type DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> DOUBLE DECFLOAT(<i>n</i>) 	The REGR_SXY aggregate function returns quantities used to compute diagnostic statistics.
"Regression functions" on page 247	SYSIBM	<ul style="list-style-type: none"> non-decfloat-numeric-type, non-decfloat-numeric-type DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> DOUBLE DECFLOAT(<i>n</i>) 	The REGR_SYY aggregate function returns quantities used to compute diagnostic statistics.

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description
"REPEAT" on page 350	SYSFUN	<ul style="list-style-type: none"> • VARCHAR(4000), INTEGER • CLOB(1M), INTEGER • BLOB(1M), INTEGER 	<ul style="list-style-type: none"> • VARCHAR(4000) • CLOB(1M) • BLOB(1M) 	This scalar function returns a character string composed of <i>argument1</i> repeated <i>argument2</i> times.
"REPLACE" on page 350	SYSFUN	<ul style="list-style-type: none"> • VARCHAR(4000), VARCHAR(4000), VARCHAR(4000) • CLOB(1M), CLOB(1M), CLOB(1M) • BLOB(1M), BLOB(1M), BLOB(1M) 	<ul style="list-style-type: none"> • VARCHAR(4000) • CLOB(1M) • BLOB(1M) 	This scalar function replaces all occurrences of <i>argument2</i> in <i>argument1</i> with <i>argument3</i> .
"RID_BIT and RID" on page 352	SYSIBM	<ul style="list-style-type: none"> • Table designator 	<ul style="list-style-type: none"> • VARCHAR(16) FOR BIT DATA 	The RID_BIT scalar function returns the row identifier (RID) of a row in a character string format. The RID_BIT function is preferred over the RID function.
"RID_BIT and RID" on page 352	SYSIBM	<ul style="list-style-type: none"> • Table designator 	<ul style="list-style-type: none"> • BIGINT 	The RID scalar function returns the RID of a row in large integer format. This function is not supported in partitioned database environments.
"RIGHT" on page 353	SYSIBM	<ul style="list-style-type: none"> • CHAR or VARCHAR, INTEGER • CLOB, INTEGER • GRAPHIC or VARGRAPHIC, INTEGER • DBCLOB, INTEGER 	<ul style="list-style-type: none"> • VARCHAR • CLOB • VARGRAPHIC • DBCLOB 	This scalar function returns the rightmost string of string-expression of length, expressed in the specified string unit. If string-expression is a character string, the result is a character string. If string-expression is a graphic string, the result is a graphic string.
"RIGHT" on page 353	SYSFUN	<ul style="list-style-type: none"> • VARCHAR(4000), INTEGER • CLOB(1M), INTEGER • BLOB(1M), INTEGER 	<ul style="list-style-type: none"> • VARCHAR(4000) • CLOB(1M) • BLOB(1M) 	This scalar function returns a string consisting of the rightmost <i>argument2</i> bytes in <i>argument1</i> .
"ROUND" on page 356	SYSIBM	<ul style="list-style-type: none"> • INTEGER, INTEGER • BIGINT, INTEGER • DOUBLE, INTEGER • DECFLOAT(<i>n</i>), INTEGER 	<ul style="list-style-type: none"> • INTEGER • BIGINT • DOUBLE • DECFLOAT(<i>n</i>) 	This scalar function returns the first argument rounded to <i>argument2</i> places right of the decimal point. If <i>argument2</i> is negative, <i>argument1</i> is rounded to the absolute value of <i>argument2</i> places to the left of the decimal point.
"RTRIM" on page 358	SYSIBM	<ul style="list-style-type: none"> • CHAR • VARCHAR • GRAPHIC • VARGRAPHIC 	<ul style="list-style-type: none"> • VARCHAR • VARCHAR • VARGRAPHIC • VARGRAPHIC 	This scalar function returns the characters of the argument with trailing blanks removed.
"SECLABEL" on page 358	SYSIBM	<ul style="list-style-type: none"> • CHAR, a string in security label string format • VARCHAR, a string in security label string format • GRAPHIC, a string in security label string format • VARGRAPHIC, a string in security label string format 	<ul style="list-style-type: none"> • DB2SECURITYLABEL 	This scalar function returns an unnamed security label.

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description
"SECLABEL_BY_NAME" on page 359	SYSIBM	<ul style="list-style-type: none"> • CHAR, DB2SECURITYLABEL • VARCHAR, DB2SECURITYLABEL • GRAPHIC, DB2SECURITYLABEL • VARGRAPHIC, DB2SECURITYLABEL 	<ul style="list-style-type: none"> • DB2SECURITYLABEL 	This scalar function returns a specific security label.
"SECLABEL_TO_CHAR" on page 360	SYSIBM	<ul style="list-style-type: none"> • CHAR, DB2SECURITYLABEL • VARCHAR, DB2SECURITYLABEL • GRAPHIC, DB2SECURITYLABEL • VARGRAPHIC, DB2SECURITYLABEL 	<ul style="list-style-type: none"> • a string in security label string format 	This scalar function accepts a security label and returns a string that contains all elements in the security label.
"SECOND" on page 361	SYSIBM	<ul style="list-style-type: none"> • VARCHAR • TIME • TIMESTAMP • DECIMAL 	<ul style="list-style-type: none"> • INTEGER • INTEGER • INTEGER • INTEGER 	This scalar function returns the second (time unit) part of a value.
"SIGN" on page 362	SYSIBM	<ul style="list-style-type: none"> • SMALLINT • INTEGER • BIGINT • DOUBLE • DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> • SMALLINT • INTEGER • BIGINT • DOUBLE • DECFLOAT(<i>n</i>) 	This scalar function returns an indicator of the sign of the argument. If the argument is less than zero, -1 is returned. If the argument equals zero, 0 is returned. If the argument is greater than zero, 1 is returned.
"SIGN" on page 362	SYSFUN	<ul style="list-style-type: none"> • SMALLINT • INTEGER • BIGINT • DOUBLE 	<ul style="list-style-type: none"> • SMALLINT • INTEGER • BIGINT • DOUBLE 	This scalar function returns an indicator of the sign of the argument. If the argument is less than zero, -1 is returned. If the argument equals zero, 0 is returned. If the argument is greater than zero, 1 is returned.
"SIN" on page 363	SYSIBM	<ul style="list-style-type: none"> • DOUBLE 	<ul style="list-style-type: none"> • DOUBLE 	This scalar function returns the sine of the argument, where the argument is an angle expressed in radians.
"SINH" on page 363	SYSIBM	<ul style="list-style-type: none"> • DOUBLE 	<ul style="list-style-type: none"> • DOUBLE 	This scalar function returns the hyperbolic sine of the argument, where the argument is an angle expressed in radians.
"SMALLINT" on page 363	SYSIBM	<ul style="list-style-type: none"> • <i>numeric-type</i> • VARCHAR 	<ul style="list-style-type: none"> • SMALLINT • SMALLINT 	This scalar function returns the small integer representation of a number.
"SOUNDEX" on page 364	SYSFUN	<ul style="list-style-type: none"> • VARCHAR(4000) 	<ul style="list-style-type: none"> • CHAR(4) 	This scalar function returns a 4-character code representing the sound of the words in the argument. This result can be compared with the sound of other strings.
"SPACE" on page 365	SYSFUN	<ul style="list-style-type: none"> • INTEGER 	<ul style="list-style-type: none"> • VARCHAR(4000) 	This scalar function returns a character string consisting of <i>argument1</i> blanks.
"SQRT" on page 365	SYSIBM	<ul style="list-style-type: none"> • DOUBLE • DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> • DOUBLE • DECFLOAT(<i>n</i>) 	This scalar function returns the square root of the argument.
"SQRT" on page 365	SYSFUN	<ul style="list-style-type: none"> • DOUBLE 	<ul style="list-style-type: none"> • DOUBLE 	This scalar function returns the square root of the argument.
"STDDEV" on page 250	SYSIBM	<ul style="list-style-type: none"> • DOUBLE • DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> • DOUBLE • DECFLOAT(<i>n</i>) 	This aggregate function returns the standard deviation of a set of numbers.

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description
"STRIP" on page 366	SYSIBM	<ul style="list-style-type: none"> CHAR VARCHAR GRAPHIC VARGRAPHIC CHAR, CHAR VARCHAR, CHAR GRAPHIC, CHAR VARGRAPHIC, CHAR 	<ul style="list-style-type: none"> CHAR VARCHAR GRAPHIC VARGRAPHIC CHAR VARCHAR GRAPHIC VARGRAPHIC 	This scalar function removes leading or trailing blanks or other specified leading or trailing characters from a string expression.
"SUBSTR" on page 366	SYSIBM	<ul style="list-style-type: none"> string-type, INTEGER string-type, INTEGER, INTEGER 	<ul style="list-style-type: none"> string-type string-type 	This scalar function returns a substring of string <i>argument1</i> , starting at <i>argument2</i> . The substring is <i>argument3</i> bytes long. If <i>argument3</i> is not specified, the remainder of the string is assumed.
"SUBSTRING" on page 369	SYSIBM	<ul style="list-style-type: none"> string-type, INTEGER, string-unit string-type, INTEGER, INTEGER, string-unit 	<ul style="list-style-type: none"> string-type string-type 	This scalar function returns a substring of string <i>argument1</i> , starting at <i>argument2</i> . The substring is <i>argument3</i> characters long. If the second INTEGER argument is not specified, the remainder of the string is assumed.
"SUM" on page 251	SYSIBM	<ul style="list-style-type: none"> numeric-type⁴ 	<ul style="list-style-type: none"> max-numeric-type¹ 	This aggregate function returns the sum of a set of numbers.
"TABLE_NAME" on page 371	SYSIBM	<ul style="list-style-type: none"> VARCHAR VARCHAR, VARCHAR 	<ul style="list-style-type: none"> VARCHAR(128) VARCHAR(128) 	This scalar function returns an unqualified name of a table or view based on the object name specified in <i>argument1</i> , and the optional schema name specified in <i>argument2</i> . The returned value is used to resolve aliases.
"TABLE_SCHEMA" on page 372	SYSIBM	<ul style="list-style-type: none"> VARCHAR VARCHAR, VARCHAR 	<ul style="list-style-type: none"> VARCHAR(128) VARCHAR(128) 	This scalar function returns the schema name portion of a two-part table or view name (given by the object name in <i>argument1</i> and the optional schema name in <i>argument2</i>). The returned value is used to resolve aliases.
"TAN" on page 373	SYSIBM	<ul style="list-style-type: none"> DOUBLE 	<ul style="list-style-type: none"> DOUBLE 	This scalar function returns the tangent of the argument, where the argument is an angle expressed in radians.
"TANH" on page 374	SYSIBM	<ul style="list-style-type: none"> DOUBLE 	<ul style="list-style-type: none"> DOUBLE 	This scalar function returns the hyperbolic tangent of the argument, where the argument is an angle expressed in radians.
"TIME" on page 374	SYSIBM	<ul style="list-style-type: none"> TIME TIMESTAMP VARCHAR 	<ul style="list-style-type: none"> TIME TIME TIME 	This scalar function returns a time from a value.
"TIMESTAMP" on page 375	SYSIBM	<ul style="list-style-type: none"> TIMESTAMP VARCHAR VARCHAR, VARCHAR VARCHAR, TIME DATE, VARCHAR DATE, TIME 	<ul style="list-style-type: none"> TIMESTAMP TIMESTAMP TIMESTAMP TIMESTAMP TIMESTAMP TIMESTAMP 	This scalar function returns a timestamp from a value or a pair of values.
"TIMESTAMP_FORMAT" on page 376	SYSIBM	<ul style="list-style-type: none"> VARCHAR, VARCHAR 	<ul style="list-style-type: none"> TIMESTAMP 	This scalar function returns a timestamp from a character string (<i>argument1</i>) that has been interpreted using a format template (<i>argument2</i>).
"TIMESTAMP_ISO" on page 379	SYSFUN	<ul style="list-style-type: none"> DATE TIME TIMESTAMP VARCHAR(26) 	<ul style="list-style-type: none"> TIMESTAMP TIMESTAMP TIMESTAMP TIMESTAMP 	This scalar function 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.

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description																		
"TIMESTAMPDIFF" on page 379	SYSFUN	<ul style="list-style-type: none"> INTEGER, CHAR(22) 	<ul style="list-style-type: none"> INTEGER 	<p>This scalar function returns an estimated number of intervals of type <i>argument1</i>, 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 interval types are:</p> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50px; text-align: right;">1</td> <td>Fractions of a second</td> </tr> <tr> <td style="text-align: right;">2</td> <td>Seconds</td> </tr> <tr> <td style="text-align: right;">4</td> <td>Minutes</td> </tr> <tr> <td style="text-align: right;">8</td> <td>Hours</td> </tr> <tr> <td style="text-align: right;">16</td> <td>Days</td> </tr> <tr> <td style="text-align: right;">32</td> <td>Weeks</td> </tr> <tr> <td style="text-align: right;">64</td> <td>Months</td> </tr> <tr> <td style="text-align: right;">128</td> <td>Quarters</td> </tr> <tr> <td style="text-align: right;">256</td> <td>Years</td> </tr> </table>	1	Fractions of a second	2	Seconds	4	Minutes	8	Hours	16	Days	32	Weeks	64	Months	128	Quarters	256	Years
1	Fractions of a second																					
2	Seconds																					
4	Minutes																					
8	Hours																					
16	Days																					
32	Weeks																					
64	Months																					
128	Quarters																					
256	Years																					
"TO_CHAR" on page 380	SYSIBM	<ul style="list-style-type: none"> same as VARCHAR_FORMAT 	<ul style="list-style-type: none"> same as VARCHAR_FORMAT 	This scalar function returns a character representation of a timestamp.																		
"TO_DATE" on page 381	SYSIBM	<ul style="list-style-type: none"> same as TIMESTAMP_FORMAT 	<ul style="list-style-type: none"> same as TIMESTAMP_FORMAT 	This scalar function returns a timestamp from a character string.																		
"TOTALORDER" on page 381	SYSIBM	<ul style="list-style-type: none"> DECFLOAT(<i>n</i>), DECFLOAT(<i>n</i>) 	<ul style="list-style-type: none"> SMALLINT 	This scalar function returns a SMALLINT value of -1, 0, or 1 that indicates the comparison order of two arguments.																		
"TRANSLATE scalar function" on page 382	SYSIBM	<ul style="list-style-type: none"> CHAR VARCHAR CHAR, VARCHAR, VARCHAR VARCHAR, VARCHAR, VARCHAR CHAR, VARCHAR, VARCHAR, VARCHAR VARCHAR, VARCHAR, VARCHAR, VARCHAR GRAPHIC, VARGRAPHIC, VARGRAPHIC VARGRAPHIC, VARGRAPHIC, VARGRAPHIC GRAPHIC, VARGRAPHIC, VARGRAPHIC, VARGRAPHIC VARGRAPHIC, VARGRAPHIC, VARGRAPHIC, VARGRAPHIC 	<ul style="list-style-type: none"> CHAR VARCHAR CHAR VARCHAR CHAR VARCHAR GRAPHIC VARGRAPHIC GRAPHIC VARGRAPHIC 	This scalar function returns a string in which one or more characters may have been converted into other characters.																		
"TRIM" on page 384	SYSIBM	<ul style="list-style-type: none"> CHAR VARCHAR GRAPHIC VARGRAPHIC CHAR, CHAR CHAR, VARCHAR CHAR, GRAPHIC CHAR, VARGRAPHIC 	<ul style="list-style-type: none"> CHAR VARCHAR GRAPHIC VARGRAPHIC CHAR VARCHAR GRAPHIC VARGRAPHIC 	This scalar function removes leading or trailing blanks or other specified leading or trailing characters from a string expression.																		

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description
"TRUNCATE" on page 385	SYSIBM	<ul style="list-style-type: none"> INTEGER, INTEGER BIGINT, INTEGER DOUBLE, INTEGER DECFLOAT(<i>n</i>), INTEGER 	<ul style="list-style-type: none"> INTEGER BIGINT DOUBLE DECFLOAT(<i>n</i>) 	This scalar function returns <i>argument1</i> truncated to <i>argument2</i> places right of the decimal point. If <i>argument2</i> is negative, <i>argument1</i> is truncated to the absolute value of <i>argument2</i> places to the left of the decimal point.
"TYPE_ID" on page 386 ³	SYSIBM	<ul style="list-style-type: none"> <i>any-structured-type</i> 	<ul style="list-style-type: none"> INTEGER 	This scalar function 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.
"TYPE_NAME" on page 387 ³	SYSIBM	<ul style="list-style-type: none"> <i>any-structured-type</i> 	<ul style="list-style-type: none"> VARCHAR(18) 	This scalar function returns the unqualified name of the dynamic data type of the argument.
"TYPE_SCHEMA" on page 387 ³	SYSIBM	<ul style="list-style-type: none"> <i>any-structured-type</i> 	<ul style="list-style-type: none"> VARCHAR(128) 	This scalar function returns the schema name of the dynamic type of the argument.
"UCASE" on page 388	SYSIBM			This scalar function is the same as UPPER.
"UCASE (Locale sensitive)" on page 388	SYSIBM			This scalar function is the same as UPPER.
"UPPER" on page 388	SYSFUN	<ul style="list-style-type: none"> VARCHAR 	<ul style="list-style-type: none"> VARCHAR 	This scalar function returns a string in which all the characters have been converted to uppercase characters.
"UPPER" on page 388	SYSIBM	<ul style="list-style-type: none"> CHAR VARCHAR 	<ul style="list-style-type: none"> CHAR VARCHAR 	This scalar function returns a string in which all the characters have been converted to uppercase characters.
"UPPER (Locale sensitive)" on page 388	SYSIBM	<ul style="list-style-type: none"> CHAR, a string that specifies a locale VARCHAR, a string that specifies a locale GRAPHIC, a string that specifies a locale VARGRAPHIC, a string that specifies a locale CHAR, a string that specifies a locale, INTEGER VARCHAR, a string that specifies a locale, INTEGER GRAPHIC, a string that specifies a locale, INTEGER VARGRAPHIC, a string that specifies a locale, INTEGER 	<ul style="list-style-type: none"> VARCHAR VARCHAR VARGRAPHIC VARGRAPHIC VARCHAR VARCHAR VARGRAPHIC VARGRAPHIC 	This scalar function returns a string in which all characters have been converted to uppercase characters using the rules from the Unicode standard associated with the specified locale.
"VALUE" on page 390 ³	SYSIBM			This scalar function is the same as COALESCE.
"VARCHAR" on page 390	SYSIBM	<ul style="list-style-type: none"> <i>character-type</i> <i>character-type</i>, INTEGER <i>datetime-type</i> 	<ul style="list-style-type: none"> VARCHAR VARCHAR VARCHAR 	This scalar function returns a VARCHAR representation of the first argument. If a second argument is present, it specifies the length of the result.
"VARCHAR_FORMAT" on page 393	SYSIBM	<ul style="list-style-type: none"> TIMESTAMP, VARCHAR VARCHAR, VARCHAR 	<ul style="list-style-type: none"> VARCHAR VARCHAR 	This scalar function returns a character representation of a timestamp (<i>argument1</i>), formatted according to a template (<i>argument2</i>).
"VARGRAPHIC" on page 396	SYSIBM	<ul style="list-style-type: none"> <i>graphic-type</i> <i>graphic-type</i>, INTEGER VARCHAR 	<ul style="list-style-type: none"> VARGRAPHIC VARGRAPHIC VARGRAPHIC 	This scalar function returns a VARGRAPHIC representation of the first argument. If a second argument is present, it specifies the length of the result.
"VARIANCE" on page 252	SYSIBM	<ul style="list-style-type: none"> DOUBLE DECFLOAT(<i>n</i>), INTEGER 	<ul style="list-style-type: none"> DOUBLE DECFLOAT(<i>n</i>) 	This aggregate function returns the variance of a set of numbers.

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description
"WEEK" on page 398	SYSFUN	<ul style="list-style-type: none"> • VARCHAR(26) • DATE • TIMESTAMP 	<ul style="list-style-type: none"> • INTEGER • INTEGER • INTEGER 	This scalar function returns the week of the year in the argument as an integer value in the range of 1-54.
"WEEK_ISO" on page 398	SYSFUN	<ul style="list-style-type: none"> • VARCHAR(26) • DATE • TIMESTAMP 	<ul style="list-style-type: none"> • INTEGER • INTEGER • INTEGER 	This scalar function returns the week of the year in the argument as an integer value in the range of 1-53. The first day of a week is Monday. Week 1 is the first week of the year to contain a Thursday.
"XMLAGG" on page 252	SYSIBM	<ul style="list-style-type: none"> • XML 	<ul style="list-style-type: none"> • XML 	This aggregate function returns an XML sequence containing an item for each non-null value in a set of XML values.
"XMLATTRIBUTES" on page 399	SYSIBM	<ul style="list-style-type: none"> • Any SQL expression, but not a structured type; cannot include a scalar fullselect. 	<ul style="list-style-type: none"> • XML 	This scalar function constructs XML attributes from the arguments.
"XMLCOMMENT" on page 400	SYSIBM	<ul style="list-style-type: none"> • <i>character-type</i> 	<ul style="list-style-type: none"> • XML 	This scalar function returns an XML value with a single XQuery comment node with the input argument as the content.
"XMLCONCAT" on page 401	SYSIBM	<ul style="list-style-type: none"> • XML 	<ul style="list-style-type: none"> • XML 	This scalar function returns a sequence containing the concatenation of a variable number of XML input arguments.
"XMLDOCUMENT" on page 402	SYSIBM	<ul style="list-style-type: none"> • XML 	<ul style="list-style-type: none"> • XML 	This scalar function returns an XML value with a single XQuery document node with zero or more children nodes.
"XMLELEMENT" on page 403	SYSIBM	<ul style="list-style-type: none"> • XML 	<ul style="list-style-type: none"> • XML 	This scalar function returns an XML value that is an XML element node.
"XMLFOREST" on page 408	SYSIBM	<ul style="list-style-type: none"> • Any SQL expression, but not a structured type; cannot include a scalar fullselect. 	<ul style="list-style-type: none"> • XML 	This scalar function returns an XML value that is a sequence of XML element nodes.
"XMLGROUP" on page 254	SYSIBM	<ul style="list-style-type: none"> • Any SQL expression, but not a structured type. 	<ul style="list-style-type: none"> • XML 	This aggregate function returns an XML value with a single XQuery document node containing one top-level element node.
"XMLNAMESPACES" on page 411	SYSIBM	<ul style="list-style-type: none"> • <i>character-type</i> 	<ul style="list-style-type: none"> • XML 	This scalar function constructs namespace declarations from the arguments.
"XMLPARSE" on page 413	SYSIBM	<ul style="list-style-type: none"> • <i>character-type</i> • BLOB 	<ul style="list-style-type: none"> • XML • XML 	This scalar function parses the argument as an XML document and returns an XML value.
"XMLPI" on page 415	SYSIBM	<ul style="list-style-type: none"> • <i>character-type</i> 	<ul style="list-style-type: none"> • XML 	This scalar function returns an XML value with a single XQuery processing instruction node.
"XMLQUERY" on page 416	SYSIBM	<ul style="list-style-type: none"> • <i>character-type</i> 	<ul style="list-style-type: none"> • XML 	This scalar function returns an XML value from the evaluation of an XQuery expression possibly using specified input arguments as XQuery variables.
"XMLROW" on page 418	SYSIBM	<ul style="list-style-type: none"> • Any SQL expression, but not a structured type. 	<ul style="list-style-type: none"> • XML 	This scalar function returns an XML value with a single XQuery document node containing one top-level element node.
"XMLSERIALIZE" on page 420	SYSIBM	<ul style="list-style-type: none"> • <i>character-type</i> • BLOB 	<ul style="list-style-type: none"> • XML • XML 	This scalar function returns a serialized XML value of the specified data type generated from the argument.
"XMLTABLE" on page 432	SYSIBM	<ul style="list-style-type: none"> • See the complete description of this function. 	<ul style="list-style-type: none"> • A table based on columns specified in input arguments. 	This table function returns a table from the evaluation of XQuery expressions, possibly using specified input arguments as XQuery variables. Each sequence item in the result sequence of the row XQuery expression represents a row of the result table.
"XMLTEXT" on page 422	SYSIBM	<ul style="list-style-type: none"> • <i>character-type</i> 	<ul style="list-style-type: none"> • XML 	This scalar function returns an XML value with a single XQuery text node having the input argument as the content.

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description
"XMLVALIDATE" on page 423	SYSIBM	<ul style="list-style-type: none"> XML 	<ul style="list-style-type: none"> XML 	This scalar function returns a copy of the input XML value augmented with information obtained from XML schema validation, including default values and type annotations.
"XMLXSROBJECTID" on page 427	SYSIBM	<ul style="list-style-type: none"> XML 	<ul style="list-style-type: none"> BIGINT 	This scalar function returns the XSR object identifier of the XML schema used to validate the XML document that is specified in the argument.
"XSLTRANSFORM" on page 428	SYSIBM	<ul style="list-style-type: none"> XML 	<ul style="list-style-type: none"> XML 	This scalar function converts XML data into other formats, including the conversion of XML documents that conform to one XML schema into documents that conform to another schema.
"YEAR" on page 431	SYSIBM	<ul style="list-style-type: none"> VARCHAR DATE TIMESTAMP DECIMAL 	<ul style="list-style-type: none"> INTEGER INTEGER INTEGER INTEGER 	This scalar function returns the year part of a value.
"+"	SYSIBM	<ul style="list-style-type: none"> <i>numeric-type, numeric-type</i> 	<ul style="list-style-type: none"> <i>max-numeric-type</i> 	Adds two numeric operands. If either operand is a special decimal floating-point value, the rules for general arithmetic operations for decimal floating-point apply. See "General arithmetic operation rules for decimal floating-point" in "General arithmetic operation rules for decimal floating-point" on page 156.
"+"	SYSIBM	<ul style="list-style-type: none"> <i>numeric-type</i> 	<ul style="list-style-type: none"> <i>numeric-type</i> 	Unary plus operator.
"+"	SYSIBM	<ul style="list-style-type: none"> DATE, DECIMAL(8,0) TIME, DECIMAL(6,0) TIMESTAMP, DECIMAL(20,6) DECIMAL(8,0), DATE DECIMAL(6,0), TIME DECIMAL(20,6), TIMESTAMP <i>datetime-type, DOUBLE, labeled-duration-code</i> <i>datetime-type, DECFLOAT, labeled-duration-code</i> 	<ul style="list-style-type: none"> DATE TIME TIMESTAMP DATE TIME TIMESTAMP <i>datetime-type</i> <i>datetime-type</i> 	Datetime plus operator.
"-"	SYSIBM	<ul style="list-style-type: none"> <i>numeric-type, numeric-type</i> 	<ul style="list-style-type: none"> <i>max-numeric-type</i> 	Subtracts two numeric operands. If either operand is a special decimal floating-point value, the rules for general arithmetic operations for decimal floating-point apply. See "General arithmetic operation rules for decimal floating-point" in "General arithmetic operation rules for decimal floating-point" on page 156.
"-"	SYSIBM	<ul style="list-style-type: none"> <i>numeric-type</i> 	<ul style="list-style-type: none"> <i>numeric-type</i>¹ 	Unary minus operator.

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description
"_"	SYSIBM	<ul style="list-style-type: none"> • DATE, DATE • TIME, TIME • TIMESTAMP, TIMESTAMP • DATE, VARCHAR • TIME, VARCHAR • TIMESTAMP, VARCHAR • VARCHAR, DATE • VARCHAR, TIME • VARCHAR, TIMESTAMP • DATE, DECIMAL(8,0) • TIME, DECIMAL(6,0) • TIMESTAMP, DECIMAL(20,6) • <i>datetime-type</i>, DOUBLE, <i>labeled-duration-code</i> • <i>datetime-type</i>, DECFLOAT, <i>labeled-duration-code</i> 	<ul style="list-style-type: none"> • DECIMAL(8,0) • DECIMAL(6,0) • DECIMAL(20,6) • DECIMAL(8,0) • DECIMAL(6,0) • DECIMAL(20,6) • DECIMAL(8,0) • DECIMAL(6,0) • DECIMAL(20,6) • DATE • TIME • TIMESTAMP • <i>datetime-type</i> • <i>datetime-type</i> 	Datetime minus operator.
"*"	SYSIBM	<ul style="list-style-type: none"> • <i>numeric-type</i>, <i>numeric-type</i> 	<ul style="list-style-type: none"> • <i>max-numeric-type</i> 	Multiplies two numeric operands. If either operand is a special decimal floating-point value, the rules for general arithmetic operations for decimal floating-point apply. See "General arithmetic operation rules for decimal floating-point" in "General arithmetic operation rules for decimal floating-point" on page 156.
"/"	SYSIBM	<ul style="list-style-type: none"> • <i>numeric-type</i>, <i>numeric-type</i> 	<ul style="list-style-type: none"> • <i>max-numeric-type</i> 	Divides two numeric operands. If either operand is a special decimal floating-point value, the rules for general arithmetic operations for decimal floating-point apply. See "General arithmetic operation rules for decimal floating-point" in "General arithmetic operation rules for decimal floating-point" on page 156.
" "	SYSIBM			Same as CONCAT.
<p>Notes</p> <ul style="list-style-type: none"> • 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. • The LONG_VARCHAR scalar function and LONG_VARGRAPHIC scalar function continue to be supported but are deprecated and might be removed in a future release. 				

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description
Key to Table				
<i>any-builtin-type</i> Any data type that is not a distinct type.				
<i>any-type</i> Any type defined to the database.				
<i>any-structured-type</i> Any user-defined structured type defined to the database.				
<i>any-comparable-type</i> Any type that is comparable with other argument types as defined in "Assignments and comparisons" on page 92.				
<i>any-union-compatible-type</i> Any type that is compatible with other argument types as defined in "Rules for result data types" on page 106.				
<i>character-type</i> Any of the character string types: CHAR, VARCHAR, CLOB.				
<i>compatible-string-type</i> A string type that comes from the same grouping as the other argument (for example, if one argument is a <i>character-type</i> the other must also be a <i>character-type</i>).				
<i>datetime-type</i> Any of the datetime types: DATE, TIME, TIMESTAMP.				
<i>exact-numeric-type</i> Any of the exact numeric types: SMALLINT, INTEGER, BIGINT, DECIMAL.				
<i>graphic-type</i> Any of the double byte character string types: GRAPHIC, VARGRAPHIC, DBCLOB.				
<i>labeled-duration-code</i> 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 "Expressions" 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.				
1		YEAR or YEARS		
2		MONTH or MONTHS		
3		DAY or DAYS		
4		HOURL or HOURS		
5		MINUTE or MINUTES		
6		SECOND or SECONDS		
7		MICROSECOND or MICROSECONDS		
<i>LOB-type</i> Any of the large object types: BLOB, CLOB, DBCLOB.				
<i>max-numeric-type</i> The maximum numeric type of the arguments where maximum is defined as the rightmost <i>numeric-type</i> .				
<i>max-string-type</i> The maximum string type of the arguments where maximum is defined as the rightmost <i>character-type</i> or <i>graphic-type</i> . If arguments are BLOB, the <i>max-string-type</i> is BLOB.				
<i>non-decfloat-numeric-type</i> Any of the numeric types except decimal floating point: SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, DOUBLE.				
<i>numeric-type</i> Any of the numeric types: SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, DOUBLE, DECFLOAT.				
<i>string-unit</i> Specifies the unit to use when determining the length of a string; can be OCTETS, CODEUNITS16, or CODEUNITS32.				
<i>string-type</i> Any type from <i>character-type</i> , <i>graphic-type</i> or BLOB.				

Supported functions and administrative SQL routines and views

Table 21. Supported functions (continued)

Function name	Schema	Input	Returns	Description
Table Footnotes				
1				When the input parameter is SMALLINT, the result type is INTEGER. When the input parameter is REAL, the result type is DOUBLE.
2				Keywords allowed are ISO, USA, EUR, JIS, and LOCAL. This function signature is not supported as a sourced function.
3				This function cannot be used as a source function.
4				The keyword ALL or DISTINCT may be used before the first parameter. If DISTINCT is specified, the use of LOB-type or a user-defined structured type is not supported.
5				The use of LOB-type or a user-defined structured type is not supported.
6				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.
7				The use of <i>graphic-type</i> or <i>LOB-type</i> is not supported.
8				The use of REFERENCE, SYSPROC.DB2SECURITYLABEL, XML, or user-defined types is not supported.
9				The use of ARRAY, LOB-type, XML, a distinct type based on any of these types, or a structured type is not supported.

Table 22. Aggregate functions

Function	Description
"ARRAY_AGG" on page 238	Aggregates a set of elements into an array.
"AVG" on page 239	Returns the average of a set of numbers.
"CORRELATION" on page 240	Returns the coefficient of correlation of a set of number pairs.
"COUNT" on page 241	Returns the number of rows or values in a set of rows or values.
"COUNT_BIG" on page 242	Returns the number of rows or values in a set of rows or values. The result can be greater than the maximum value of INTEGER.
"COVARIANCE" on page 243	Returns the covariance of a set of number pairs.
"GROUPING" on page 244	Used with grouping-sets and super-groups to indicate sub-total rows generated by a grouping set. The value returned is 0 or 1. A value of 1 means that the value of the argument in the returned row is a null value, and the row was generated for a grouping set. This generated row provides a sub-total for a grouping set.
"MAX" on page 245	Returns the maximum value in a set of values.
"MIN" on page 246	Returns the minimum value in a set of values.
"Regression functions" on page 247	The REGR_AVGX aggregate function returns quantities used to compute diagnostic statistics.
"Regression functions" on page 247	The REGR_AVGY aggregate function returns quantities used to compute diagnostic statistics.
"Regression functions" on page 247	The REGR_COUNT aggregate function returns the number of non-null number pairs used to fit the regression line.
"Regression functions" on page 247	The REGR_INTERCEPT or REGR_ICPT aggregate function returns the y-intercept of the regression line.
"Regression functions" on page 247	The REGR_R2 aggregate function returns the coefficient of determination for the regression.
"Regression functions" on page 247	The REGR_SLOPE aggregate function returns the slope of the line.
"Regression functions" on page 247	The REGR_SXX aggregate function returns quantities used to compute diagnostic statistics.

Supported functions and administrative SQL routines and views

Table 22. Aggregate functions (continued)

Function	Description
"Regression functions" on page 247	The REGR_SXY aggregate function returns quantities used to compute diagnostic statistics.
"Regression functions" on page 247	The REGR_SYY aggregate function returns quantities used to compute diagnostic statistics.
"STDDEV" on page 250	Returns the standard deviation of a set of numbers.
"SUM" on page 251	Returns the sum of a set of numbers.
"VARIANCE" on page 252	Returns the variance of a set of numbers.

Table 23. Cast scalar functions

Function	Description
"BIGINT" on page 260	Returns a 64-bit integer representation of a number or a character string in the form of an integer constant.
"BLOB" on page 263	Returns a BLOB representation of a string of any type.
"CHAR" on page 264	Returns a CHARACTER representation of a value.
"CLOB" on page 270	Returns a CLOB representation of a value.
"DATE" on page 276	Returns a DATE from a value.
"DBCLOB" on page 280	Returns a DBCLOB representation of a string.
"DECFLOAT" on page 282	Returns the decimal floating-point representation of a number.
"DECIMAL" on page 283	Returns a DECIMAL representation of a number.
"DOUBLE" on page 291	Returns a DOUBLE PRECISION representation of a number.
EMPTY_BLOB, EMPTY_CLOB, and EMPTY_DBCLOB scalar functions	Return a zero-length value of the associated data type.
"FLOAT" on page 294	Returns a FLOAT representation of a number.
"GRAPHIC" on page 297	Returns a GRAPHIC representation of a string.
"INTEGER" on page 309	Returns an INTEGER representation of a number.
"REAL" on page 345	Returns a REAL representation of a number.
"SMALLINT" on page 363	Returns a SMALLINT representation of a number.
"TIME" on page 374	Returns a TIME from a value.
"TIMESTAMP" on page 375	Returns a TIMESTAMP from a value or a pair of values.
"VARCHAR" on page 390	Returns a VARCHAR representation of a value.
"VARGRAPHIC" on page 396	Returns a VARGRAPHIC representation of a value.

Table 24. Partitioning scalar functions

Function	Description
"DATAPARTITIONNUM" on page 275	Returns the sequence number (SYSDATAPARTITIONS.SEQNO) of the data partition in which the row resides. The argument is any column name within the table.
"DBPARTITIONNUM" on page 280	Returns the database partition number of the row. The argument is any column name within the table.

Supported functions and administrative SQL routines and views

Table 24. Partitioning scalar functions (continued)

Function	Description
"HASHEDVALUE" on page 299	Returns the distribution map index (0 to 4095) of the row. The argument is a column name within a table.

Table 25. Datetime scalar functions

Function	Description
"DAY" on page 277	Returns the day part of a value.
"DAYNAME" on page 278	Returns a mixed case character string containing the name of the day (for example, Friday) for the day portion of the argument, based on what the locale was when db2start was issued.
"DAYOFWEEK" on page 278	Returns the day of the week from a value, where 1 is Sunday and 7 is Saturday.
"DAYOFWEEK_ISO" on page 278	Returns the day of the week from a value, where 1 is Monday and 7 is Sunday.
"DAYOFYEAR" on page 279	Returns the day of the year from a value.
"DAYS" on page 279	Returns an integer representation of a date.
"HOUR" on page 302	Returns the hour part of a value.
"JULIAN_DAY" on page 310	Returns an integer value representing the number of days from January 1, 4712 B.C. to the date specified in the argument.
"MICROSECOND" on page 325	Returns the microsecond part of a value.
"MIDNIGHT_SECONDS" on page 325	Returns an integer value representing the number of seconds between midnight and a specified time value.
"MINUTE" on page 327	Returns the minute part of a value.
"MONTH" on page 328	Returns the month part of a value.
"MONTHNAME" on page 329	Returns a mixed case character string containing the name of the month (for example, January) for the month portion of the argument that is a date or a timestamp, based on what the locale was when the database was started.
"QUARTER" on page 343	Returns an integer that represents the quarter of the year in which a date resides.
"SECOND" on page 361	Returns the seconds part of a value.
"TIMESTAMP_FORMAT" on page 376	Returns a timestamp from a character string (<i>argument1</i>) that has been interpreted using a format template (<i>argument2</i>).
"TIMESTAMP_ISO" on page 379	Returns a timestamp value based on a date, time, or timestamp argument. If the argument is a date, it inserts zero for all the time elements. If the argument is a time, it inserts the value of CURRENT DATE for the date elements, and zero for the fractional time element.
"TIMESTAMPDIFF" on page 379	Returns an estimated number of intervals of type <i>argument1</i> , based on the difference between two timestamps. The second argument is the result of subtracting two timestamp types and converting the result to CHAR.
"TO_CHAR" on page 380	Returns a CHARACTER representation of a timestamp.

Supported functions and administrative SQL routines and views

Table 25. Datetime scalar functions (continued)

Function	Description
"TO_DATE" on page 381	Returns a timestamp from a character string.
"VARCHAR_FORMAT" on page 393	Returns a CHARACTER representation of a timestamp (<i>argument1</i>), formatted according to a template (<i>argument2</i>).
"WEEK" on page 398	Returns the week of the year from a value, where the week starts with Sunday.
"WEEK_ISO" on page 398	Returns the week of the year from a value, where the week starts with Monday.
"YEAR" on page 431	Returns the year part of a value.

Table 26. Numeric scalar functions

Function	Description
"ABS or ABSVAL" on page 257	Returns the absolute value of a number.
"ACOS" on page 257	Returns the arc cosine of a number, in radians.
"ASIN" on page 258	Returns the arc sine of a number, in radians.
"ATAN" on page 259	Returns the arc tangent of a number, in radians.
"ATANH" on page 259	Returns the hyperbolic arc tangent of a number, in radians.
"ATAN2" on page 259	Returns the arc tangent of x and y coordinates as an angle expressed in radians.
"CEILING" on page 264	Returns the smallest integer value that is greater than or equal to a number.
"COMPARE_DECFLOAT" on page 273	Returns a SMALLINT value that indicates whether the two arguments are equal or unordered, or whether one argument is greater than the other.
"COS" on page 274	Returns the cosine of a number.
"COSH" on page 274	Returns the hyperbolic cosine of a number.
"COT" on page 275	Returns the cotangent of the argument, where the argument is an angle expressed in radians.
"DEGREES" on page 288	Returns the number of degrees of an angle.
"DIGITS" on page 290	Returns a character-string representation of the absolute value of a number.
"EXP" on page 294	Returns a value that is the base of the natural logarithm (e) raised to a power specified by the argument.
"FLOOR" on page 295	Returns the largest integer value that is less than or equal to a number.
"LN" on page 316	Returns the natural logarithm of a number.
"LOG10" on page 320	Returns the common logarithm (base 10) of a number.
"MOD" on page 328	Returns the remainder of the first argument divided by the second argument.
"MULTIPLY_ALT" on page 329	Returns the product of two arguments as a decimal value. This function is useful when the sum of the argument precisions is greater than 31.
"NORMALIZE_DECFLOAT" on page 330	Returns a decimal floating-point value that is the result of the argument set to its simplest form.

Supported functions and administrative SQL routines and views

Table 26. Numeric scalar functions (continued)

Function	Description
"POWER" on page 341	Returns the result of raising the first argument to the power of the second argument.
"QUANTIZE" on page 342	Returns a decimal floating-point number that is equal in value and sign to the first argument, and whose exponent is equal to the exponent of the second argument.
"RADIANS" on page 343	Returns the number of radians for an argument that is expressed in degrees.
"RAND" on page 345	Returns a random number.
"ROUND" on page 356	Returns a numeric value that has been rounded to the specified number of decimal places.
"SIGN" on page 362	Returns the sign of a number.
"SIN" on page 363	Returns the sine of a number.
"SINH" on page 363	Returns the hyperbolic sine of a number.
"SQRT" on page 365	Returns the square root of a number.
"TAN" on page 373	Returns the tangent of a number.
"TANH" on page 374	Returns the hyperbolic tangent of a number.
"TOTALORDER" on page 381	Returns a SMALLINT value of -1, 0, or 1 that indicates the comparison order of two arguments.
"TRUNCATE" on page 385	Returns a number value that has been truncated at a specified number of decimal places.

Table 27. Security scalar functions

Function	Description
"SECLABEL" on page 358	Returns an unnamed security label.
"SECLABEL_BY_NAME" on page 359	Returns a specific security label.
"SECLABEL_TO_CHAR" on page 360	Accepts a security label and returns a string that contains all elements in the security label.

Table 28. XML functions

Function	Description
"PARAMETER" on page 336	Represents a position in an SQL statement where the value is provided dynamically by XQuery as part of the invocation of the db2-fn:sqlquery function.
"XMLAGG" on page 252	Returns an XML sequence containing an item for each non-null value in a set of XML values.
"XMLATTRIBUTES" on page 399	Constructs XML attributes from the arguments.
"XMLCOMMENT" on page 400	Returns an XML value with a single XQuery comment node with the input argument as the content.
"XMLCONCAT" on page 401	Returns a sequence containing the concatenation of a variable number of XML input arguments.
"XMLDOCUMENT" on page 402	Returns an XML value with a single XQuery document node with zero or more children nodes.

Supported functions and administrative SQL routines and views

Table 28. XML functions (continued)

Function	Description
“XMLELEMENT” on page 403	Returns an XML value that is an XML element node.
“XMLFOREST” on page 408	Returns an XML value that is a sequence of XML element nodes.
“XMLGROUP” on page 254	Returns an XML value with a single XQuery document node containing one top-level element node.
“XMLNAMESPACES” on page 411	Constructs namespace declarations from the arguments.
“XMLPARSE” on page 413	Parses the argument as an XML document and returns an XML value.
“XMLPI” on page 415	Returns an XML value with a single XQuery processing instruction node.
“XMLQUERY” on page 416	Returns an XML value from the evaluation of an XQuery expression possibly using specified input arguments as XQuery variables.
“XMLROW” on page 418	Returns an XML value with a single XQuery document node containing one top-level element node.
“XMLSERIALIZE” on page 420	Returns a serialized XML value of the specified data type generated from the argument.
“XMLTABLE” on page 432	Returns a table from the evaluation of XQuery expressions, possibly using specified input arguments as XQuery variables. Each sequence item in the result sequence of the row XQuery expression represents a row of the result table.
“XMLTEXT” on page 422	Returns an XML value with a single XQuery text node having the input argument as the content.
“XMLVALIDATE” on page 423	Returns a copy of the input XML value augmented with information obtained from XML schema validation, including default values and type annotations.
“XMLXSROBJECTID” on page 427	Returns the XSR object identifier of the XML schema used to validate the XML document that is specified in the argument
“XSLTRANSFORM” on page 428	Converts XML data into other formats, including the conversion of XML documents that conform to one XML schema into documents that conform to another schema.

Table 29. String scalar functions

Function	Description
“ASCII” on page 258	Returns the ASCII code value of the leftmost character of the argument as an integer.
“CHARACTER_LENGTH” on page 268	Returns the length of an expression in the specified <i>string-unit</i> .
“CHR” on page 270	Returns the character that has the ASCII code value specified by the argument.
“COLLATION_KEY_BIT” on page 271	Returns a VARCHAR FOR BIT DATA string representing the collation key of the specified <i>string-expression</i> in the specified <i>collation-name</i> .
“CONCAT” on page 274	Returns a string that is the concatenation of two strings.

Supported functions and administrative SQL routines and views

Table 29. String scalar functions (continued)

Function	Description
“DECRYPT_BIN and DECRYPT_CHAR” on page 287	Returns a value that is the result of decrypting encrypted data using a password string.
“DIFFERENCE” on page 289	Returns the difference between the sounds of the words in two argument strings, as determined by the SOUNDEX function. A value of 4 means the strings sound the same.
“ENCRYPT” on page 292	Returns a value that is the result of encrypting a data string expression.
“GENERATE_UNIQUE” on page 295	Returns a bit data character string that is unique compared to any other execution of the same function.
“GETHINT” on page 297	Returns the password hint if one is found.
“INSERT” on page 306	Returns a string, where <i>argument3</i> bytes have been deleted from <i>argument1</i> (beginning at <i>argument2</i>), and <i>argument4</i> has been inserted into <i>argument1</i> (beginning at <i>argument2</i>).
“LCASE” on page 311	Returns a string in which all the SBCS characters have been converted to lowercase characters.
“LCASE (Locale sensitive)” on page 311	Returns a string in which all characters have been converted to lowercase characters using the rules from the Unicode standard associated with the specified locale.
“LOWER (Locale sensitive)” on page 321	Returns a string in which all characters have been converted to lowercase characters using the rules from the Unicode standard associated with the specified locale.
“LEFT” on page 312	Returns the leftmost characters from a string.
“LOCATE scalar function” on page 317	Returns the starting position of one string within another string.
“LOWER” on page 321	Returns a string in which all the characters have been converted to lowercase characters.
“LTRIM” on page 323	Removes blanks from the beginning of a string expression.
“OCTET_LENGTH” on page 332	Returns the length of an expression in octets (bytes).
“OVERLAY” on page 332	Returns a string in which, beginning at <i>start</i> in the specified <i>source-string</i> , <i>length</i> of the specified code units have been deleted and <i>insert-string</i> has been inserted.
“POSITION scalar function” on page 336	Returns the starting position of <i>argument2</i> within <i>argument1</i> .
“POSSTR scalar function” on page 339	Returns the starting position of one string within another string.
“REPEAT” on page 350	Returns a character string composed of <i>argument1</i> repeated <i>argument2</i> times.
“REPLACE” on page 350	Replaces all occurrences of <i>argument2</i> in <i>argument1</i> with <i>argument3</i> .
“RIGHT” on page 353	Returns the rightmost characters from a string.
“RTRIM” on page 358	Removes blanks from the end of a string expression.
“SOUNDEX” on page 364	Returns a 4-character code representing the sound of the words in the argument. This result can be compared with the sound of other strings.
“SPACE” on page 365	Returns a character string that consists of a specified number of blanks.

Supported functions and administrative SQL routines and views

Table 29. String scalar functions (continued)

Function	Description
"STRIP" on page 366	Removes leading or trailing blanks or other specified leading or trailing characters from a string expression.
"SUBSTR" on page 366	Returns a substring of a string.
"SUBSTRING" on page 369	Returns a substring of a string.
"TRANSLATE scalar function" on page 382	Returns a string in which one or more characters in a string are converted to other characters.
"TRIM" on page 384	Removes leading or trailing blanks or other specified leading or trailing characters from a string expression.
"UCASE" on page 388	The UCASE function is identical to the TRANSLATE function except that only the first argument (<i>char-string-exp</i>) is specified.
"UCASE (Locale sensitive)" on page 388	Returns a string in which all characters have been converted to uppercase characters using the rules from the Unicode standard associated with the specified locale.
"UPPER" on page 388	Returns a string in which all the characters have been converted to uppercase characters.
"UPPER (Locale sensitive)" on page 388	Returns a string in which all characters have been converted to uppercase characters using the rules from the Unicode standard associated with the specified locale.

Table 30. Miscellaneous scalar functions

Function	Description
"BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT" on page 261	These bitwise functions operate on the "two's complement" representation of the integer value of the input arguments and return the result as a corresponding base 10 integer value in a data type based on the data type of the input arguments.
"CARDINALITY" on page 263	Returns a value of type BIGINT representing the number of elements of an array
"COALESCE" on page 271	Returns the first argument that is not null.
"DECODE" on page 286	Compares each specified <i>expression2</i> to <i>expression1</i> . If <i>expression1</i> is equal to <i>expression2</i> , or both <i>expression1</i> and <i>expression2</i> are null, the value of the following <i>result-expression</i> is returned. If no <i>expression2</i> matches <i>expression1</i> , the value of <i>else-expression</i> is returned; otherwise a null value is returned.
"DEREF" on page 289	Returns an instance of the target type of the reference type argument.
"EVENT_MON_STATE" on page 293	Returns the operational state of particular event monitor.
"GREATEST" on page 299	Returns the maximum value in a set of values.
"HEX" on page 300	Returns a hexadecimal representation of a value.
"IDENTITY_VAL_LOCAL" on page 302	Returns the most recently assigned value for an identity column.
"LEAST" on page 311	Returns the minimum value in a set of values.
"LENGTH" on page 314	Returns the length of a value.
"MAX" on page 324	Returns the maximum value in a set of values.

Supported functions and administrative SQL routines and views

Table 30. Miscellaneous scalar functions (continued)

Function	Description
"MAX_CARDINALITY" on page 324	Returns a value of type BIGINT representing the maximum number of elements that an array can contain.
"MIN" on page 326	Returns the minimum value in a set of values.
"NULLIF" on page 331	Returns a null value if the arguments are equal; otherwise, it returns the value of the first argument.
"NVL" on page 331	Returns the first argument that is not null.
"RAISE_ERROR" on page 344	Raises an error in the SQLCA. The sqlstate that is to be returned is indicated by <i>argument1</i> . The second argument contains any text that is to be returned.
"REC2XML" on page 346	Returns a string formatted with XML tags, containing column names and column data.
"RID_BIT and RID" on page 352	The RID_BIT scalar function returns the row identifier (RID) of a row in a character string format. The RID scalar function returns the RID of a row in large integer format. The RID function is not supported in partitioned database environments. The RID_BIT function is preferred over the RID function.
"TABLE_NAME" on page 371	Returns an unqualified name of a table or view based on the object name specified in <i>argument1</i> , and the optional schema name specified in <i>argument2</i> . The returned value is used to resolve aliases.
"TABLE_SCHEMA" on page 372	Returns the schema name portion of a two-part table or view name (given by the object name in <i>argument1</i> and the optional schema name in <i>argument2</i>). The returned value is used to resolve aliases.
"TYPE_ID" on page 386	Returns the internal data type identifier of the dynamic data type of the argument. The result of this function is not portable across databases.
"TYPE_NAME" on page 387	Returns the unqualified name of the dynamic data type of the argument.
"TYPE_SCHEMA" on page 387	Returns the schema name of the dynamic data type of the argument.
"VALUE" on page 390	Returns the first argument that is not null.

Aggregate functions

The argument of an aggregate function is a set of values derived from an expression. The expression can include columns, but cannot include a *scalar-fullselect*, another column function, or an XMLQUERY or XMLEXISTS expression (SQLSTATE 42607). The scope of the set is a group or an intermediate result table.

If a GROUP BY clause is specified in a query, and the intermediate result of the FROM, WHERE, GROUP BY, and HAVING clauses is the empty set, the aggregate functions are not applied; the result of the query is the empty set; the SQLCODE is set to +100; and the SQLSTATE is set to '02000'.

If a GROUP BY clause is *not* specified in a query, and the intermediate result of the FROM, WHERE, and HAVING clauses is the empty set, the aggregate functions are applied to the empty set.

Aggregate functions

For example, the result of the following SELECT statement is the number of distinct values of JOBCODE for employees in department D01:

```
SELECT COUNT(DISTINCT JOBCODE)
FROM CORPDATA.EMPLOYEE
WHERE WORKDEPT = 'D01'
```

The keyword DISTINCT is not considered to be an argument of the function, but rather a specification of an operation that is performed before the function is applied. If DISTINCT is specified, duplicate values are eliminated. When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

If ALL is implicitly or explicitly specified, duplicate values are not eliminated.

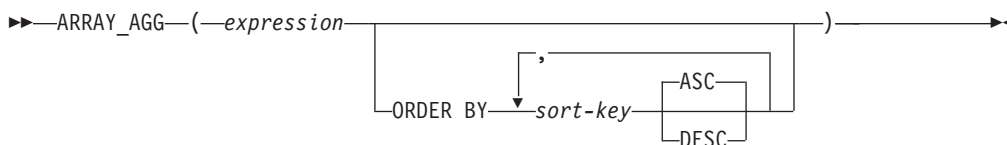
For compatibility with other SQL implementations, UNIQUE can be specified as a synonym for DISTINCT in aggregate functions.

Expressions can be used in aggregate functions. For example:

```
SELECT MAX(BONUS + 1000)
INTO :TOP_SALESREP_BONUS
FROM EMPLOYEE
WHERE COMM > 5000
```

Aggregate functions can be qualified with a schema name (for example, SYSIBM.COUNT(*)).

ARRAY_AGG



The schema is SYSIBM.

The ARRAY_AGG function aggregates a set of elements into an array. The data type of the expression must be a data type that can be specified in a CREATE TYPE (Array) statement (SQLSTATE 42884).

If *sort-key* is specified, it determines the order of the aggregated elements in the array. If *sort-key* is not specified, the ordering of elements within the array is not deterministic. If *sort-key* is not specified, and ARRAY_AGG is specified more than once in the same SELECT clause, the same ordering of elements within the array is used for each result of ARRAY_AGG.

If a SELECT clause has multiple occurrences of XMLAGG or ARRAY_AGG that specify *sort-key*, all the sort keys must be identical (SQLSTATE 428GZ).

The ARRAY_AGG function can only be specified within an SQL procedure in the following specific contexts (SQLSTATE 42887):

- The select-list of a SELECT INTO statement
- The select-list of a fullselect in the definition of a cursor that is not scrollable
- The select-list of a scalar subquery on the right side of a SET statement

ARRAY_AGG cannot be used as part of an OLAP function, the SELECT statement that uses ARRAY_AGG cannot contain an ORDER BY clause or a DISTINCT clause, and the SELECT or HAVING clause cannot contain a subquery or call an SQL function.

Example:

- Given the following DDL:

```
CREATE TYPE PHONELIST AS DECIMAL(10, 0)ARRAY[10]

CREATE TABLE EMPLOYEE (
  ID          INTEGER NOT NULL,
  PRIORITY   INTEGER NOT NULL,
  PHONENUMBER DECIMAL(10, 0),
  PRIMARY KEY(ID, PRIORITY))
```

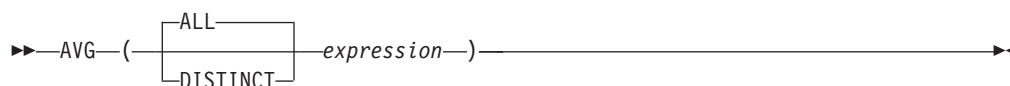
Create a procedure that uses a SELECT INTO statement to return the prioritized list of contact numbers under which an employee can be reached.

```
CREATE PROCEDURE GETPHONENUMBERS
  (IN EMPID INTEGER,
   OUT NUMBERS PHONELIST)
BEGIN
  SELECT ARRAY_AGG(PHONENUMBER ORDER BY PRIORITY)
  INTO NUMBERS
  FROM EMPLOYEE
  WHERE ID = EMPID;
END
```

Create a procedure that uses a SET statement to return the list of an employee's contact numbers in an arbitrary order.

```
CREATE PROCEDURE GETPHONENUMBERS
  (IN EMPID INTEGER,
   OUT NUMBERS PHONELIST)
BEGIN
  SET NUMBERS =
  (SELECT ARRAY_AGG(PHONENUMBER)
   FROM EMPLOYEE
   WHERE ID = EMPID);
END
```

AVG



The schema is SYSIBM.

The AVG function returns the average 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, except for a decimal result data

AVG

type. For decimal results, their sum must be within the range supported by a decimal data type having a precision of 31 and a scale identical to the scale of the argument values. 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.
- The result is DECFLOAT(34) if the argument is DECFLOAT(*n*).

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, redundant duplicate values are eliminated. When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the average value of 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.

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—(—*expression1*—,—*expression2*—)—►►

The schema is SYSIBM.

The CORRELATION function returns the coefficient of correlation of a set of number pairs.

The argument values must be numbers.

If either argument is decimal floating-point, the result is DECFLOAT(34); otherwise, the result is a double-precision floating-point number. The result can be null. When not null, the result is between -1 and 1.

The function is applied to the set of (*expression1*, *expression2*) pairs derived from the argument values by the elimination of all pairs for which either *expression1* or *expression2* is null.

If the function is applied to an empty set, or if either $\text{STDDEV}(\text{expression1})$ or $\text{STDDEV}(\text{expression2})$ 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:

$$\frac{\text{COVARIANCE}(\text{expression1}, \text{expression2})}{(\text{STDDEV}(\text{expression1}) * \text{STDDEV}(\text{expression2}))}$$

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

CORR can be specified in place of CORRELATION.

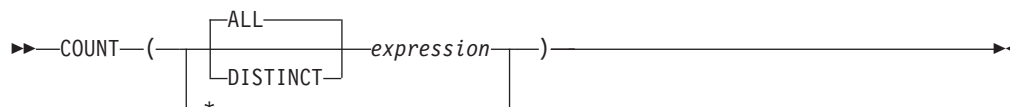
Example:

- Using the EMPLOYEE table, set the host variable CORRLN (double-precision floating point) to the correlation between salary and bonus for those employees in department (WORKDEPT) 'A00'.

```
SELECT CORRELATION(SALARY, BONUS)
  INTO :CORRLN
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00'
```

CORRLN is set to approximately 9.99853953399538E-001 when using the sample table.

COUNT



The schema is SYSIBM.

The COUNT function returns the number of rows or values in a set of rows or values.

If DISTINCT is specified, the resulting data type of *expression* cannot be a LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB, XML, distinct type on any of these types, or structured type (SQLSTATE 42907). Otherwise the result data type of *expression* can be any data type.

COUNT

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'
```

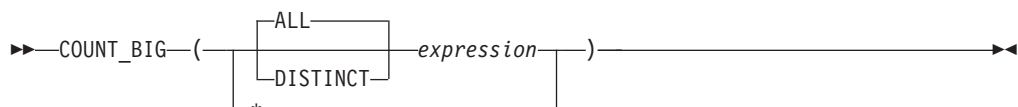
Results in FEMALE being set to 13 when using the sample table.

- 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.

```
SELECT COUNT(DISTINCT WORKDEPT)
INTO :FEMALE_IN_DEPT
FROM EMPLOYEE
WHERE SEX = 'F'
```

Results in FEMALE_IN_DEPT being set to 5 when using the sample table. (There is at least one female in departments A00, C01, D11, D21, and E11.)

COUNT_BIG



The 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 specified, the resulting data type of *expression* cannot be a LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB, XML, distinct type on any of these types, or structured type (SQLSTATE 42907). Otherwise the result data type of *expression* can be any data type.

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.

```
CREATE FUNCTION RICK.COUNT() RETURNS DECIMAL(15,0)
SOURCE SYSIBM.COUNT_BIG();
SET CURRENT PATH RICK, SYSTEM PATH;
SELECT COUNT(*) FROM EMPLOYEE;
```

Note how the sourced function is defined with no parameters to support COUNT(*). This only works if you name the function COUNT and do not qualify the function with the schema name when it is used. To get the same effect as COUNT(*) with a name other than COUNT, invoke the function with no parameters. Thus, if RICK.COUNT had been defined as RICK.MYCOUNT instead, the query would have to be written as follows:

```
SELECT MYCOUNT() FROM EMPLOYEE;
```

If the count is taken on a specific column, the sourced function must specify the type of the column. The following statements created a sourced function that will take any CHAR column as a argument and use COUNT_BIG to perform the counting.

```
CREATE FUNCTION RICK.COUNT(CHAR()) RETURNS DOUBLE
SOURCE SYSIBM.COUNT_BIG(CHAR());
SELECT COUNT(DISTINCT WORKDEPT) FROM EMPLOYEE;
```

COVARIANCE

►►—COVARIANCE—(—*expression1*—,—*expression2*—)—◄◄

The schema is SYSIBM.

The COVARIANCE function returns the (population) covariance of a set of number pairs.

The argument values must be numbers.

COVARIANCE

If either argument is decimal floating-point, the result is DECFLOAT(34); otherwise, the result is a double-precision floating-point number. The result can be null.

The function is applied to the set of (*expression1*,*expression2*) pairs derived from the argument values by the elimination of all pairs for which either *expression1* or *expression2* is null.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the covariance of the value pairs in the set. The 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.

COVAR can be specified in place of COVARIANCE.

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 1.6888888888889E+006 when using the sample table.

GROUPING

►►—GROUPING—(—*expression*—)—————►►

The schema is SYSIBM.

Used in conjunction with grouping-sets and super-groups, the GROUPING function returns a value that indicates whether or not a row returned in a GROUP BY answer set is a row generated by a grouping set that excludes the column represented by *expression*.

The 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:

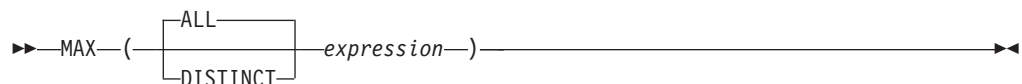
```
SELECT SALES_DATE, SALES_PERSON,
       SUM(SALES) AS UNITS_SOLD,
       GROUPING(SALES_DATE) AS DATE_GROUP,
       GROUPING(SALES_PERSON) AS SALES_GROUP
FROM SALES
GROUP BY CUBE (SALES_DATE, SALES_PERSON)
ORDER BY SALES_DATE, SALES_PERSON
```

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



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.

The resulting data type of *expression* cannot be a LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB, distinct type on any of these types, or structured type (SQLSTATE 42907).

MAX

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
```

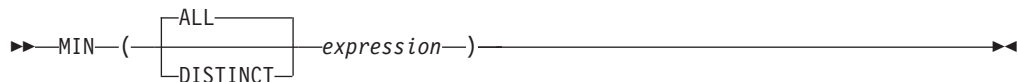
Results in LAST_PROJ being set to 'WELD LINE PLANNING' when using the sample table.

- 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



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.

The resulting data type of *expression* cannot be a LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB, distinct type on any of these types, or structured type (SQLSTATE 42907).

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'
```

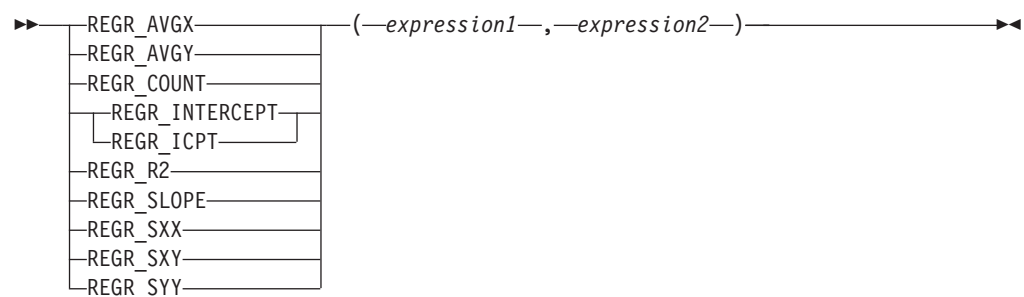
Results in COMM_SPREAD being set to 1118 (that is, 2580 - 1462) when using the sample table.

- 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



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 (that is, a "y value"). The second element of each pair (*expression2*) is interpreted as a value of the independent variable (that is, an "x value").

Regression functions

The REGR_COUNT function returns the number of non-null number pairs used to fit the regression line (see below).

The REGR_INTERCEPT (or REGR_ICPT) function returns the y-intercept of the regression line ("b" in the above equation).

The REGR_R2 function returns the coefficient of determination ("R-squared" or "goodness-of-fit") for the regression.

The REGR_SLOPE function returns the slope of the line ("a" in the above equation).

The REGR_AVGX, REGR_AVGY, REGR_SXX, REGR_SXY, and REGR_SYY functions return quantities that can be used to compute various diagnostic statistics needed for the evaluation of the quality and statistical validity of the regression model (see below).

The argument values must be numbers.

The data type of the result of REGR_COUNT is integer. For the remaining functions, if either argument is DECFLOAT(*n*), the data type of the result is DECFLOAT(34); otherwise, the data type of the result is double-precision floating-point. If either argument is a special decimal floating-point value, the rules for general arithmetic operations for decimal floating-point apply. See "General arithmetic operation rules for decimal floating-point" in "General arithmetic operation rules for decimal floating-point" on page 156.

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

$$1 - ((1 - \text{REGR_R2}) * ((\text{REGR_COUNT} - 1) / (\text{REGR_COUNT} - 2)))$$

Standard error

$$\text{SQRT}((\text{REGR_SYY} - (\text{POWER}(\text{REGR_SXY}, 2) / \text{REGR_SXX})) / (\text{REGR_COUNT} - 2))$$

Total sum of squares

$$\text{REGR_SYY}$$

Regression sum of squares

$$\text{POWER}(\text{REGR_SXY}, 2) / \text{REGR_SXX}$$

Residual sum of squares

$$(\text{Total sum of squares}) - (\text{Regression sum of squares})$$

t statistic for slope

$$\text{REGR_SLOPE} * \text{SQRT}(\text{REGR_SXX}) / (\text{Standard error})$$

t statistic for y-intercept

$$\text{REGR_INTERCEPT} / ((\text{Standard error}) * \text{SQRT}((1 / \text{REGR_COUNT}) + (\text{POWER}(\text{REGR_AVGX}, 2) / \text{REGR_SXX})))$$

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.

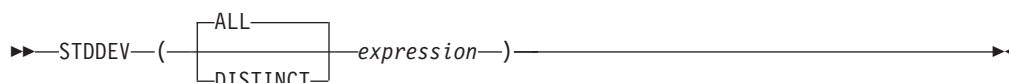
```
SELECT REGR_SLOPE(BONUS, SALARY), REGR_INTERCEPT(BONUS, SALARY),
REGR_R2(BONUS, SALARY), REGR_COUNT(BONUS, SALARY),
REGR_AVGX(BONUS, SALARY), REGR_AVGY(BONUS, SALARY),
REGR_SXX(BONUS, SALARY), REGR_SYY(BONUS, SALARY),
REGR_SXY(BONUS, SALARY)
INTO :SLOPE, :ICPT,
:RSQR, :CNT,
:AVGSAL, :AVGBONUS,
:SXX, :SYY,
:SXY
FROM EMPLOYEE
WHERE WORKDEPT = 'A00'
```

Regression functions

When using the sample table, the host variables are set to the following approximate values:

```
SLOPE: +1.71002671916749E-002
ICPT: +1.00871888623260E+002
RSQR: +9.99707928128685E-001
CNT: 3
AVGSAL: +4.28333333333333E+004
AVGBONUS: +8.33333333333333E+002
SXX: +2.96291666666667E+008
SYY: +8.66666666666667E+004
SXY: +5.06666666666667E+006
```

STDDEV



The schema is SYSIBM.

The STDDEV function returns the standard deviation ($/n$) of a set of numbers. The formula used to calculate STDDEV is:

$$\text{STDDEV} = \text{SQRT}(\text{VARIANCE})$$

where SQRT(VARIANCE) is the square root of the variance.

The argument values must be numbers.

If the argument is DECFLOAT(n), the result is DECFLOAT(n); otherwise, the result is double-precision floating-point. The result can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are eliminated. When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the standard deviation of the values in the set.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

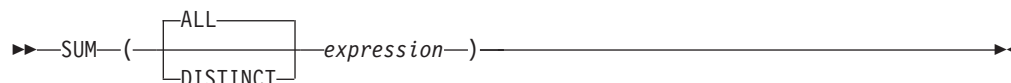
Example:

- Using the EMPLOYEE table, set the host variable DEV (double-precision floating point) to the standard deviation of the salaries of employees in department (WORKDEPT) 'A00'.

```
SELECT STDDEV(SALARY)
  INTO :DEV
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00'
```

DEV is set to a number with an approximate value of 9938.00.

SUM



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.
- The result is DECFLOAT(34) if the argument is DECFLOAT(*n*).

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, redundant duplicate values are also eliminated. When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the sum of the values in the set.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

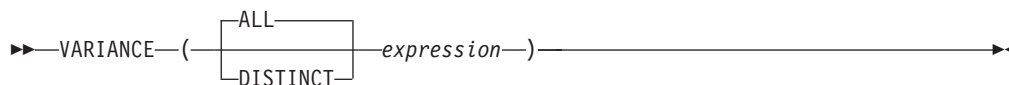
Example:

- Using the EMPLOYEE table, set the host variable JOB_BONUS (decimal(9,2)) to the total bonus (BONUS) paid to clerks (JOB='CLERK').

```
SELECT SUM(BONUS)
  INTO :JOB_BONUS
  FROM EMPLOYEE
  WHERE JOB = 'CLERK'
```

Results in JOB_BONUS being set to 2800 when using the sample table.

VARIANCE



The schema is SYSIBM.

The VARIANCE function returns the variance of a set of numbers.

The argument values must be numbers.

If the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*); otherwise, the result is double-precision floating-point. The result can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are eliminated. When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the variance of the values in the set.

The order in which the values are added is undefined, but every intermediate result must be within the range of the result data type.

VAR can be specified in place of VARIANCE.

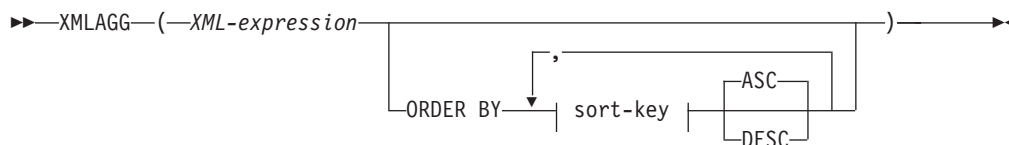
Example:

- 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.

XMLAGG



The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLAGG function returns an XML sequence containing an item for each non-null value in a set of XML values.

XML-expression

Specifies an expression of data type XML.

ORDER BY

Specifies the order of the rows from the same grouping set that are processed in the aggregation. If the ORDER BY clause is omitted, or if the ORDER BY clause cannot distinguish the order of the column data, the rows in the same grouping set are arbitrarily ordered.

sort-key

The sort key can be a column name or a *sort-key-expression*. Note that if the sort key is a constant, it does not refer to the position of the output column (as in the ordinary ORDER BY clause), but it is simply a constant, which implies no sort key.

The data type of the result is XML.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If the *XML-expression* argument can be null, the result can be null. If the set of values is empty, the result is the null value. Otherwise, the result is an XML sequence containing an item for each value in the set.

Note:

1. **Support in multiple database partition databases:** The result, at the outer level of XML value function nesting, must be an argument of the XMLSERIALIZE function.
2. **Support in OLAP expressions:** XMLAGG cannot be used as a column function of an OLAP aggregation function (SQLSTATE 42601).

Example:

Note: XMLAGG does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- Construct a department element for each department, containing a list of employees sorted by last name.

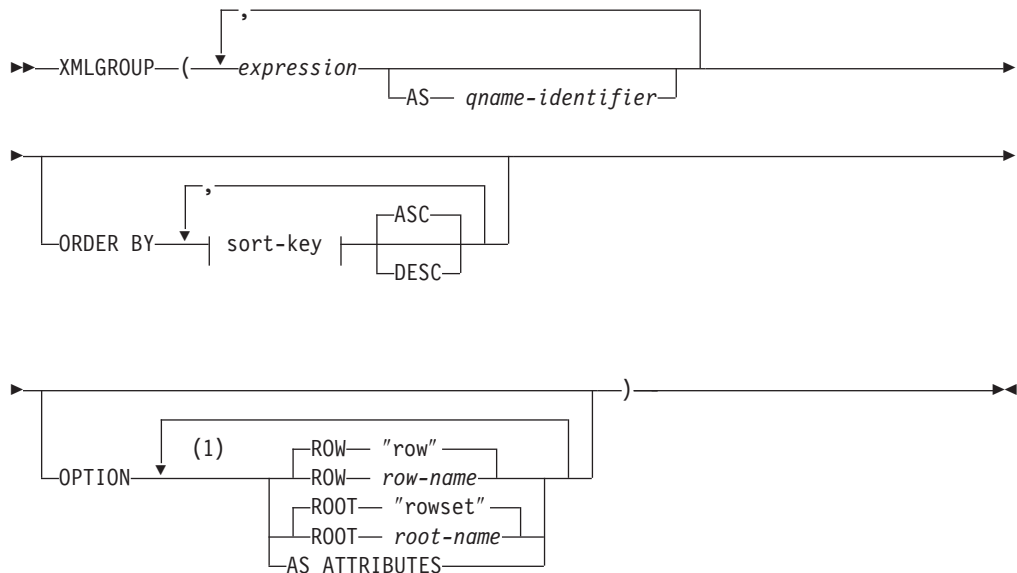
```
SELECT XMLSERIALIZE(
  CONTENT XMLELEMENT(
    NAME "Department", XMLATTRIBUTES(
      E.WORKDEPT AS "name"
    ),
    XMLAGG(
      XMLELEMENT(
        NAME "emp", E.LASTNAME
      )
      ORDER BY E.LASTNAME
    )
  )
  AS CLOB(110)
)
AS "dept_list"
FROM EMPLOYEE E
WHERE E.WORKDEPT IN ('C01','E21')
GROUP BY WORKDEPT
```

This query produces the following result:

```
dept_list
-----
<Department name="C01">
  <emp>KWAN</emp>
  <emp>NICHOLLS</emp>
  <emp>QUINTANA</emp>
</Department>
<Department name="E21">
  <emp>GOUNOT</emp>
  <emp>LEE</emp>
  <emp>MEHTA</emp>
  <emp>SPENSER</emp>
</Department>
```

XMLGROUP

The XMLGROUP function returns an XML value with a single XQuery document node containing one top-level element node. This is an aggregate expression that will return a single-rooted XML document from a group of rows where each row is mapped to a row subelement.



Notes:

- 1 The same clause must not be specified more than once.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

expression

The content of each generated XML element node (or the value of each generated attribute) is specified by an expression. The data type of *expression* cannot be a structured type (SQLSTATE 42884). The expression can be any SQL expression. If the expression is not a simple column reference, a *qname-identifier* must be specified.

AS qname-identifier

Specifies the XML element name or attribute name as an SQL identifier. The *qname-identifier* must be of the form of an XML qualified name, or QName

(SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope (SQLSTATE 42635). If *qname-identifier* is not specified, *expression* must be a column name (SQLSTATE 42703). The element name or attribute name is created from the column name using the fully escaped mapping from a column name to an QName.

OPTION

Specifies additional options for constructing the XML value. If no OPTION clause is specified, the default behavior applies.

ROW *row-name*

Specifies the name of the element to which each row is mapped. If this option is not specified, the default element name is "row".

ROOT *root-name*

Specifies the name of the root element node. If this option is not specified, the default root element name is "rowset"

AS ATTRIBUTES

Specifies that each expression is mapped to an attribute value with column name or *qname-identifier* serving as the attribute name.

ORDER BY

Specifies the order of the rows from the same grouping set that are processed in the aggregation. If the ORDER BY clause is omitted, or if the ORDER BY clause cannot distinguish the order of the column data, the rows in the same grouping set are arbitrarily ordered.

sort-key

The sort key can be a column name or a *sort-key-expression*. Note that if the sort key is a constant, it does not refer to the position of the output column (as in the ordinary ORDER BY clause), but it is simply a constant, which implies no sort key.

Notes

The default behavior defines a simple mapping between a result set and an XML value. Some additional notes about function behavior apply:

- By default, each row is transformed into an XML element named "row" and each column is transformed into a nested element with the column name serving as the element name.
- The null handling behavior is NULL ON NULL. A NULL value in a column maps to the absence of the subelement. If all column values are NULL, no row element will be generated.
- The binary encoding scheme for BLOB and FOR BIT DATA data types is base64Binary encoding.
- By default, the elements corresponding to the rows in a group are children of a root element named "rowset".
- The order of the row subelements in the root element will be the same as the order in which the rows are returned in the query result set.
- A document node will be added implicitly to the root element to make the XML result a well-formed single-rooted XML document

Examples

Assume the following table T1 with integer columns C1 and C2 that contain numeric data stored in a relational format.

C1	C2
1	2
-	2
1	-
-	-

4 record(s) selected.

- The following example shows an XMLGroup query and output fragment with default behavior, using a single top-level element to represent the table:

```
SELECT XMLGROUP(C1, C2)FROM T1
<rowset>
  <row>
    <C1>1</C1>
    <C2>2</C2>
  </row>
  <row>
    <C2>2</C2>
  </row>
  <row>
    <C1>1</C1>
  </row>
</rowset>
```

1 record(s) selected.

- The following example shows an XMLGroup query and output fragment with attribute centric mapping. Instead of appearing as nested elements as in the previous example, relational data is mapped to element attributes:

```
SELECT XMLGROUP(C1, C2 OPTION AS ATTRIBUTES) FROM T1
<rowset>
  <row C1="1" C2="2"/>
  <row C2="2"/>
  <row C1="1"/>
</rowset>
```

1 record(s) selected.

- The following example shows an XMLGroup query and output fragment with the default <rowset> root element replaced by <document> and the default <row> element replaced by <entry>. Columns C1 and C2 are returned as <column1> and <column2> elements, and the return set is ordered by column C1:

```
SELECT XMLGROUP(
  C1 AS "column1", C2 AS "column2"
  ORDER BY C1 OPTION ROW "entry" ROOT "document")
FROM T1
<document>
  <entry>
    <column1>1</column1>
    <column2>2</column2>
  </entry>
  <entry>
    <column1>1</column1>
  </entry>
  <entry>
    <column2>2</column2>
  </entry>
</document>
```


Scalar functions

A scalar function 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 to a set of values.

The result of the following SELECT statement has as many rows as there are employees in department D01:

```
SELECT EMPNO, LASTNAME, YEAR(CURRENT DATE - BRTHDATE)
FROM EMPLOYEE
WHERE WORKDEPT = 'D01'
```

Scalar functions can be qualified with a schema name (for example, SYSIBM.CHAR(123)).

In a Unicode database, all scalar functions that accept a character or graphic string will accept any string types for which conversion is supported.

ABS or ABSVAL

►► ABS (—expression—) ◀◀
 └─ ABSVAL ─┘

The schema is SYSIBM.

This function was first available in FixPak 2 of Version 7.1. The SYSFUN version of the ABS (or ABSVAL) function continues to be available.

Returns the absolute value of the argument. The argument can be any built-in numeric data type.

The result has the same data type and length attribute as the argument. The result can be null; if the argument is null, the result is the null value. If the argument is the maximum negative value for SMALLINT, INTEGER or BIGINT, the result is an overflow error.

Example:

```
ABS(-51234)
```

returns an INTEGER with a value of 51234.

ACOS

►► ACOS (—expression—) ◀◀

The schema is SYSIBM. (The SYSFUN version of the ACOS function continues to be available.)

ACOS

Returns the arccosine of the argument as an angle expressed in radians.

The argument can be of any built-in numeric data type (except for DECFLOAT). 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 can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

Example:

Assume that the host variable ACOSINE is a DECIMAL(10,9) host variable with a value of 0.070737202.

```
SELECT ACOS(:ACOSINE)
FROM SYSIBM.SYSDUMMY1
```

This statement returns the approximate value 1.49.

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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed. 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 SYSIBM. (The SYSFUN version of the ASIN function continues to be available.)

Returns the arcsine on the argument as an angle expressed in radians.

The argument can be of any built-in numeric type (except for DECFLOAT). 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 can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

ATAN

►► ATAN(*—expression—*) ◄◄

The schema is SYSIBM. (The SYSFUN version of the ATAN function continues to be available.)

Returns the arctangent of the argument as an angle expressed in radians.

The argument can be of any built-in numeric data type (except for DECFLOAT). 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 can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

ATAN2

►► ATAN2(*—expression—*, *—expression—*) ◄◄

The schema is SYSIBM. (The SYSFUN version of the ATAN2 function continues to be available.)

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 (except for DECFLOAT). 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 the argument can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

ATANH

►► ATANH(*—expression—*) ◄◄

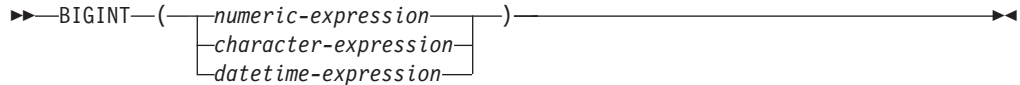
The schema is SYSIBM.

Returns the hyperbolic arctangent of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric data type (except for DECFLOAT). 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 can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

BIGINT



The schema is SYSIBM.

The BIGINT function returns a 64-bit integer representation of a number, character string, date, time, or timestamp in the form of an integer constant. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

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 big 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.

datetime-expression

An expression that is of one of the following data types:

- DATE. The result is a BIGINT value representing the date as *yyyymmdd*.
- TIME. The result is a BIGINT value representing the time as *hhmmss*.
- TIMESTAMP. The result is a BIGINT value representing the timestamp as *yyyymmddhhmmss*. The microseconds portion of the timestamp value is not included in the result.

The result of the function is a big integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

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
```

- Assume that the column RECEIVED (timestamp) has an internal value equivalent to '1988-12-22-14.07.21.136421'.

```
BIGINT(RECEIVED)
```

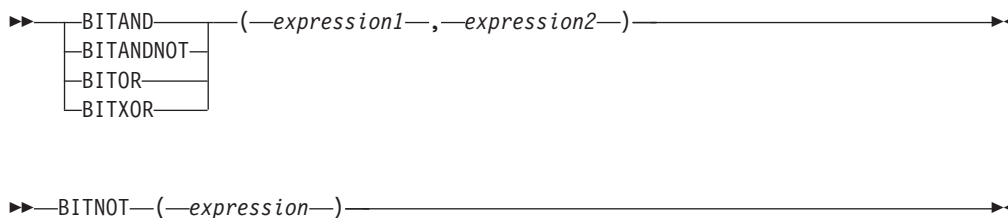
results in the value 19 881 222 140 721.

- Assume that the column STARTTIME (time) has an internal value equivalent to '12:03:04'.

```
BIGINT(STARTTIME)
```

results in the value 120 304.

BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT



The schema is SYSIBM.

These bitwise functions operate on the "two's complement" representation of the integer value of the input arguments and return the result as a corresponding base 10 integer value in a data type based on the data type of the input arguments.

Table 31. The bit manipulation functions

Function	Description	A bit in the two's complement representation of the result is:
BITAND	Performs a bitwise AND operation.	1 only if the corresponding bits in both arguments are 1.
BITANDNOT	Clears any bit in the first argument that is in the second argument.	Zero if the corresponding bit in the second argument is 1; otherwise, the result is copied from the corresponding bit in the first argument.
BITOR	Performs a bitwise OR operation.	1 unless the corresponding bits in both arguments are zero.
BITXOR	Performs a bitwise exclusive OR operation.	1 unless the corresponding bits in both arguments are the same.
BITNOT	Performs a bitwise NOT operation.	Opposite of the corresponding bit in the argument.

BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT

The arguments must be integer values represented by the data types SMALLINT, INTEGER, BIGINT, or DECFLOAT. Arguments of type DECIMAL, REAL, or DOUBLE are cast to DECFLOAT. The value is truncated to a whole number.

The bit manipulation functions can operate on up to 16 bits for SMALLINT, 32 bits for INTEGER, 64 bits for BIGINT, and 113 bits for DECFLOAT. The range of supported DECFLOAT values includes integers from -2^{112} to $2^{112} - 1$, and special values such as NaN or INFINITY are not supported (SQLSTATE 42815). If the two arguments have different data types, the argument supporting fewer bits is cast to a value with the data type of the argument supporting more bits. This cast impacts the bits that are set for negative values. For example, -1 as a SMALLINT value has 16 bits set to 1, which when cast to an INTEGER value has 32 bits set to 1.

The result of the functions with two arguments has the data type of the argument that is highest in the data type precedence list for promotion. If either argument is DECFLOAT, the data type of the result is DECFLOAT(34). If either argument can be null, the result can be null; if either argument is null, the result is the null value.

The result of the BITNOT function has the same data type as the input argument, except that DECIMAL, REAL, DOUBLE, or DECFLOAT(16) returns DECFLOAT(34). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Due to differences in internal representation between data types and on different hardware platforms, using functions (such as HEX) or host language constructs to view or compare internal representations of BIT function results and arguments is data type-dependent and not portable. The data type- and platform-independent way to view or compare BIT function results and arguments is to use the actual integer values.

Use of the BITXOR function is recommended to toggle bits in a value. Use the BITANDNOT function to clear bits. BITANDNOT(val, pattern) operates more efficiently than BITAND(val, BITNOT(pattern)).

Examples:

The following examples are based on an ITEM table with a PROPERTIES column of type INTEGER.

- Return all items for which the third property bit is set.

```
SELECT ITEMID FROM ITEM
WHERE BITAND(PROPERTIES, 4) = 4
```

- Return all items for which the fourth or the sixth property bit is set.

```
SELECT ITEMID FROM ITEM
WHERE BITAND(PROPERTIES, 40) <> 0
```

- Clear the twelfth property of the item whose ID is 3412.

```
UPDATE ITEM
SET PROPERTIES = BITANDNOT(PROPERTIES, 2048)
WHERE ITEMID = 3412
```

- Set the fifth property of the item whose ID is 3412.

```
UPDATE ITEM
SET PROPERTIES = BITOR(PROPERTIES, 16)
WHERE ITEMID = 3412
```

BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT

- Toggle the eleventh property of the item whose ID is 3412.

```
UPDATE ITEM
  SET PROPERTIES = BITXOR(PROPERTIES, 1024)
  WHERE ITEMID = 3412
```

- Switch all the bits in a 16-bit value that has only the second bit on.

```
VALUES BITNOT(CAST(2 AS SMALLINT))
```

returns -3 (with a data type of SMALLINT).

BLOB

►► 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 CONCAT ': ' ) CONCAT TOPOGRAPHIC_MAP
FROM ONTARIO_SERIES_4
WHERE TOPOGRAPHIC_MAP LIKE BLOB('%Pellow Island%')
```

CARDINALITY

►► CARDINALITY (—*array-expression*—)

The schema is SYSIBM.

The CARDINALITY function returns a value of type BIGINT representing the number of elements of an array.

The argument can be either an SQL procedure variable or parameter of an array data type or a cast specification of a parameter marker to an array data type. The value returned by the CARDINALITY function is the highest subindex for which the array has an assigned element. This includes elements that have been assigned the null value.

CARDINALITY

The function returns 0 if the array is empty. The result can be null; if the argument is null, the result is the null value.

Example:

Assume that type INT_ARRAY is defined as:

```
CREATE TYPE INT_ARRAY
AS INTEGER ARRAY[100]
```

The SET statement in the following fragment of SQL PL code assigns variable LEN the value 4:

```
BEGIN
DECLARE LEN INTEGER;
DECLARE MYARRAY INT_ARRAY;

SET INT_ARRAY = ARRAY[0,0,1,1];
SET LEN = CARDINALITY(MYARRAY);
END
```

CEILING

►►—CEILING—(*—expression—*)—

The schema is SYSIBM. (The SYSFUN version of the CEILING function continues to be available.)

Returns the smallest integer value greater than or equal to the argument.

The argument can be of any built-in numeric type. The result of the function has the same data type and length attribute as the argument except that the scale is 0 if the argument is DECIMAL. For example, an argument with a data type of DECIMAL(5,5) returns DECIMAL(5,0).

The result can be null if the argument can be null or if the argument is not a decimal floating-point number and the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

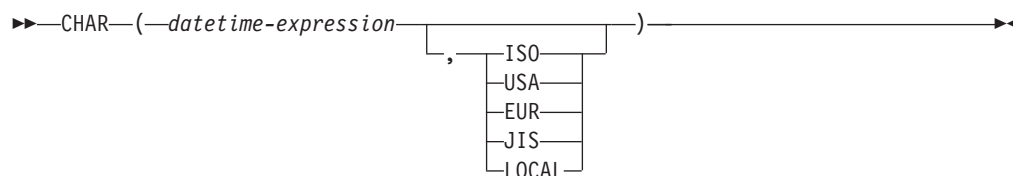
CEIL can be specified in place of CEILING.

CHAR

Character to Character:

►►—CHAR—(*—character-expression—* [*—integer—*])—

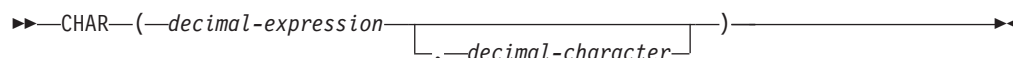
Datetime to Character:



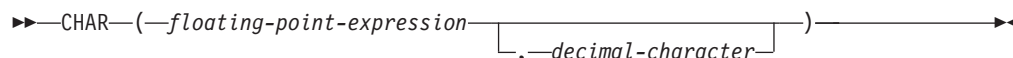
Integer to Character:



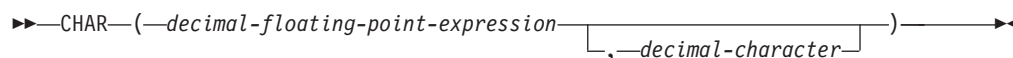
Decimal to Character:



Floating-point to Character:



Decimal Floating-point to Character:



The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature. The `SYSFUN.CHAR(floating-point-expression)` signature continues to be available. In this case, the decimal character is locale sensitive, and therefore returns either a period or a comma, depending on the locale of the database server.

The CHAR function returns a fixed-length character string representation of:

- A character string, if the first argument is any type of character string
- A datetime value, if the first argument is a date, time, or timestamp
- An integer number, if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number, if the first argument is a decimal number
- A double-precision floating-point number, if the first argument is a DOUBLE or REAL
- A decimal floating-point number, if the first argument is a DECFLOAT

The first argument must be of a built-in data type. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

Note: The CAST expression can also be used to return a string expression.

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.

Character to Character

character-expression

An expression that returns a value that is of the CHAR, VARCHAR, LONG VARCHAR, or CLOB data type.

integer

The length attribute of 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, the result is truncated. 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).

Datetime to Character

datetime-expression

An expression that is of 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 is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

time The result is the character string representation of the time in the format specified by the second argument. The length of the result is 8. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

timestamp

The result is the character string representation of the timestamp. The length of the result is 26. The second argument is not applicable and must not be specified (SQLSTATE 42815).

The code page of the string is the code page of the database at the application server.

Integer to Character

integer-expression

An expression that returns a value that is of 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, which represent the significant digits in the argument, and is preceded by a minus sign if the argument is negative. The result is left justified.

- If the first argument is a small integer, the length of the result is 6.
- If the first argument is a large integer, the length of the result is 11.
- If the first argument is a big integer, the length of the result is 20.

If the number of bytes in the result is less than the defined length of the result, the result is padded on the right with blanks.

The code page of the 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 are required, the DECIMAL scalar function can be used first to make the change.

decimal-character

Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is 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.

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).

decimal-character

Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is 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 the *decimal-character* and a sequence of digits. If the number of bytes in the result is less than 24, the result is padded on the right with blanks.

The code page of the string is the code page of the database at the application server.

Decimal Floating-point to Character*decimal-floating-point-expression*

An expression that returns a value that is a decimal floating-point data type (DECFLOAT).

decimal-character

Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

CHAR

The result is the fixed-length character string representation of the argument. The length of the result is 42. If the number of characters in the result is less than 42, the result is padded on the right with blank characters to a length of 42.

Examples:

- Assume that the PRSTDATE column has an internal value equivalent to 1988-12-25. The following function returns the value '12/25/1988'.

```
CHAR(PRSTDATE, USA)
```

- Assume that the STARTING column has an internal value equivalent to 17:12:30, and that the host variable HOUR_DUR (decimal(6,0)) is a time duration with a value of 050000 (that is, 5 hours). The following function returns the value '5:12 PM'.

```
CHAR(STARTING, USA)
```

The following function returns the value '10:12 PM'.

```
CHAR(STARTING + :HOUR_DUR, USA)
```

- Assume that the RECEIVED column (TIMESTAMP) has an internal value equivalent to the combination of the PRSTDATE and STARTING columns. The following function returns the value '1988-12-25-17.12.30.000000'.

```
CHAR(RECEIVED)
```

- The LASTNAME column is defined as VARCHAR(15). The following function returns the values in this column as fixed-length character strings that are 10 bytes long. LASTNAME values that are more than 10 bytes long (excluding trailing blanks) are truncated and a warning is returned.

```
SELECT CHAR(LASTNAME,10) FROM EMPLOYEE
```

- The EDLEVEL column is defined as SMALLINT. The following function returns the values in this column as fixed-length character strings. An EDLEVEL value of 18 is returned as the CHAR(6) value '18 ' ('18' followed by four blanks).

```
SELECT CHAR(EDLEVEL) FROM EMPLOYEE
```

- The SALARY column is defined as DECIMAL with a precision of 9 and a scale of 2. The current value (18357.50) is to be displayed with a comma as the decimal character (18357,50). The following function returns the value '00018357,50'.

```
CHAR(SALARY, ',')
```

- Values in the SALARY column are to be subtracted from 20000.25 and displayed with the default decimal character. The following function returns the value '-0001642.75'.

```
CHAR(20000.25 - SALARY)
```

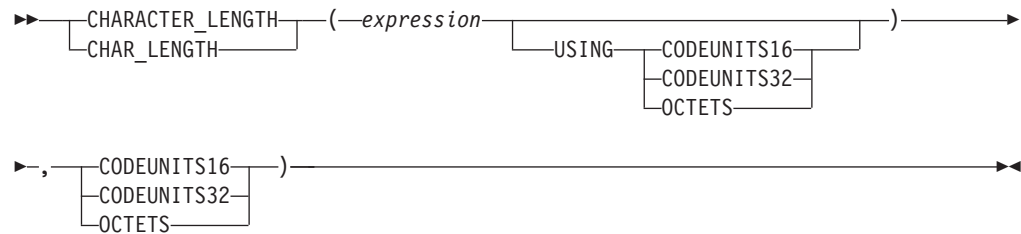
- Assume that the host variable SEASONS_TICKETS is defined as INTEGER and has a value of 10000. The following function returns the value '10000.00 '.

```
CHAR(DECIMAL(:SEASONS_TICKETS,7,2))
```

- Assume that the host variable DOUBLE_NUM is defined as DOUBLE and has a value of -987.654321E-35. The following function returns the value '-9.87654321E-33 '. Because the result data type is CHAR(24), there are nine trailing blanks in the result.

```
CHAR(:DOUBLE_NUM)
```

CHARACTER_LENGTH



The schema is SYSIBM.

The CHARACTER_LENGTH function returns the length of *expression* in the specified string unit.

expression

An expression that returns a value of a built-in character or graphic string.

CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the string unit of the result. CODEUNITS16 specifies that the result is to be expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that the result is to be expressed in 32-bit UTF-32 code units. OCTETS specifies that the result is to be expressed in bytes.

If a string unit is specified as CODEUNITS16 or CODEUNITS32, and *expression* is a binary string or bit data, an error is returned (SQLSTATE 428GC). If a string unit is specified as OCTETS and *expression* is a binary string, an error is returned (SQLSTATE 42815). For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The length of character and graphic strings includes trailing blanks. The length of varying-length strings is the actual length and not the maximum length.

Examples:

- Assume that NAME is a VARCHAR(128) column, encoded in Unicode UTF-8, that contains the value 'Jürgen'. The following two queries return the value 6:

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

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

The following two queries return the value 7:

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

```
SELECT LENGTH(NAME)
FROM T1 WHERE NAME = 'Jürgen'
```

- The following examples work with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown below in different Unicode encoding forms:

	'&'	'N'	'~'	'A'	'B'
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'

CHARACTER_LENGTH

	'&'	'N'	'~'	'A'	'B'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'
UTF-32BE	X'0001D11E'	X'0000004E'	X'00000303'	X'00000041'	X'00000042'

Assume that the variable UTF8_VAR contains the UTF-8 representation of the string.

```
SELECT CHARACTER_LENGTH(UTF8_VAR, CODEUNITS16),  
       CHARACTER_LENGTH(UTF8_VAR, CODEUNITS32),  
       CHARACTER_LENGTH(UTF8_VAR, OCTETS)  
FROM SYSIBM.SYSDUMMY1
```

returns the values 6, 5, and 9, respectively.

Assume that the variable UTF16_VAR contains the UTF-16BE representation of the string.

```
SELECT CHARACTER_LENGTH(UTF16_VAR, CODEUNITS16),  
       CHARACTER_LENGTH(UTF16_VAR, CODEUNITS32),  
       CHARACTER_LENGTH(UTF16_VAR, OCTETS)  
FROM SYSIBM.SYSDUMMY1
```

returns the values 6, 5, and 12, respectively.

CHR

►► CHR(—*expression*—)◄◄

The schema is SYSFUN.

Returns the character that has the ASCII code value specified by the argument. If *expression* is 0, the result is the blank character (X'20').

The argument can be either INTEGER or SMALLINT. The value of the argument should be between 0 and 255; otherwise, the return value is the character that has the ASCII code value corresponding to 255.

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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

character-string-expression

An expression that returns a value that is a character string. The expression cannot be a character string defined as FOR BIT DATA (SQLSTATE 42846).

integer

An integer value specifying the length attribute of the resulting CLOB data

type. The value must be between 0 and 2 147 483 647. If a value for *integer* is not specified, the length of the result is the same as the length of the first argument.

The result of the function is a CLOB. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

COALESCE

►►—COALESCE—(—*expression*—, —*expression*—) —————►►

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. They can be of either a built-in or user-defined data type. (This function cannot be used as a source function when creating a user-defined function. Because this function accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.)

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
```

COLLATION_KEY_BIT

►►—COLLATION_KEY_BIT—(—*string-expression*—, —*collation-name*—, —*length*—) —————►►

The schema is SYSIBM.

The COLLATION_KEY_BIT function returns a VARCHAR FOR BIT DATA string representing the collation key of the *string-expression* in the specified *collation-name*.

COLLATION_KEY_BIT

The results of COLLATION_KEY_BIT for two strings can be binary compared to determine their order within the specified *collation-name*. For the comparison to be meaningful, the results used must be from the same *collation-name*.

string-expression

An expression that returns a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC string for which the collation key should be determined. If *string-expression* is a CHAR or VARCHAR, the expression must not be FOR BIT DATA (SQLSTATE 429BM).

If *string-expression* is not in UTF-16, this function performs code page conversion of *string-expression* to UTF-16. If the result of the code page conversion contains at least one substitution character, this function will return a collation key of the UTF-16 string with the substitution character(s) and the warning flag SQLWARN8 in the SQLCA will be set to 'W'.

collation-name

A character constant that specifies the collation to use when determining the collation key. The value of *collation-name* is not case sensitive and must be one of the "Unicode Collation Algorithm-based collations" in *Internationalization Guide* or "language-aware collations for Unicode data" in *Internationalization Guide* (SQLSTATE 42616).

length

An expression that specifies the length attribute of the result in bytes. If specified, *length* must be an integer between 1 and 32 672 (SQLSTATE 42815).

If a value for *length* is not specified, the length of the result is determined as follows:

Table 32. Determining the result length

String Argument Data Type	Result Data Type Length
CHAR(<i>n</i>) or VARCHAR(<i>n</i>)	Minimum of 12 <i>n</i> bytes and 32 672 bytes
GRAPHIC(<i>n</i>) or VARGRAPHIC(<i>n</i>)	Minimum of 12 <i>n</i> bytes and 32 672 bytes

Regardless of whether *length* is specified or not, if the length of the collation key is longer than the length of the result data type, an error is returned (SQLSTATE 42815). The actual result length of the collation key is approximately six times the length of *string-expression* after it has been converted to UTF-16.

If *string-expression* is an empty string, the result is a valid collation key that can have a non-zero length.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples:

The following query orders employees by their surnames using the language-aware collation for German in code page 923:

```
SELECT FIRSTNAME, LASTNAME
FROM EMPLOYEE
ORDER BY COLLATION_KEY_BIT (LASTNAME, 'SYSTEM_923_DE')
```

The following query uses a culturally correct comparison to find the departments of employees in the province of Québec:


```

SELECT E.WORKDEPT
FROM EMPLOYEE AS E INNER JOIN SALES AS S
ON COLLATION_KEY_BIT(E.LASTNAME, 'UCA400R1_LFR') =
   COLLATION_KEY_BIT(S.SALES_PERSON, 'UCA400R1_LFR')
WHERE S.REGION = 'Quebec'

```

COMPARE_DECFLOAT

►►—COMPARE_DECFLOAT—(—*expression1*—,—*expression2*—)—————►►

The schema is SYSIBM.

The COMPARE_DECFLOAT function returns a SMALLINT value that indicates whether the two arguments are equal or unordered, or whether one argument is greater than the other.

expression1

An expression that returns a value of any built-in numeric data type. If the argument is not DECFLOAT(34), it is logically converted to DECFLOAT(34) for processing.

expression2

An expression that returns a value of any built-in numeric data type. If the argument is not DECFLOAT(34), it is logically converted to DECFLOAT(34) for processing.

The value of *expression1* is compared with the value of *expression2*, and the result is returned according to the following rules:

- If both arguments are finite, the comparison is algebraic and follows the procedure for decimal floating-point subtraction. If the difference is exactly zero with either sign, the arguments are equal. If a non-zero difference is positive, the first argument is greater than the second argument. If a non-zero difference is negative, the first argument is less than the second.
- Positive zero and negative zero compare as equal.
- Positive infinity compares equal to positive infinity.
- Positive infinity compares greater than any finite number.
- Negative infinity compares equal to negative infinity.
- Negative infinity compares less than any finite number.
- Numeric comparison is exact. The result is determined for finite operands as if range and precision were unlimited. No overflow or underflow condition can occur.
- If either argument is NaN or sNaN (positive or negative), the result is unordered.

The result value is as follows:

- 0 if the arguments are exactly equal
- 1 if *expression1* is less than *expression2*
- 2 if *expression1* is greater than *expression2*
- 3 if the arguments are unordered

The result of the function is a SMALLINT value. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Examples:

COMPARE_DECFLOAT

- The following examples show the values that are returned by the COMPARE_DECFLOAT function, given a variety of input decimal floating-point values:

```
COMPARE_DECFLOAT(DECFLOAT(2.17), DECFLOAT(2.17)) = 0
COMPARE_DECFLOAT(DECFLOAT(2.17), DECFLOAT(2.170)) = 2
COMPARE_DECFLOAT(DECFLOAT(2.170), DECFLOAT(2.17)) = 1
COMPARE_DECFLOAT(DECFLOAT(2.17), DECFLOAT(0.0)) = 2
COMPARE_DECFLOAT(INFINITY, INFINITY) = 0
COMPARE_DECFLOAT(INFINITY, -INFINITY) = 2
COMPARE_DECFLOAT(DECFLOAT(-2), INFINITY) = 1
COMPARE_DECFLOAT(NAN, NAN) = 3
COMPARE_DECFLOAT(DECFLOAT(-0.1), SNAN) = 3
```

CONCAT

(1)
▶▶ CONCAT (—*expression1*—, —*expression2*—) ▶▶

Notes:

1 || can 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 whose length is the sum of the lengths of the two arguments. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

COS

▶▶ COS (—*expression*—) ▶▶

The schema is SYSIBM. (The SYSFUN version of the COS function continues to be available.)

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 (except for DECFLOAT). 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 can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

COSH

▶▶ COSH (—*expression*—) ▶▶

The schema is SYSIBM.

Returns the hyperbolic cosine of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric data type (except for DECFLOAT). 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 can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

COT

►►—COT—(*expression*)—◄◄

The schema is SYSIBM. (The SYSFUN version of the COT function continues to be available.)

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 (except for DECFLOAT). 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 can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

DATAPARTITIONNUM

►►—DATAPARTITIONNUM—(*column-name*)—◄◄

The schema is SYSIBM.

The DATAPARTITIONNUM function returns the sequence number (SYSDATAPARTITIONS.SEQNO) of the data partition in which the row resides. Data partitions are sorted by range, and sequence numbers start at 0. For example, the DATAPARTITIONNUM function returns 0 for a row that resides in the data partition with the lowest range.

The argument must be the qualified or unqualified name of any column in the table. Because row-level information is returned, the result is the same regardless of which column is specified. The column can have any data type.

If *column-name* references a column in a view, the expression for the column in the view must reference a column of the underlying base table, and the view must be deletable. A nested or common table expression follows the same rules as a view.

DATAPARTITIONNUM

The data type of the result is INTEGER and is never null.

This function cannot be used as a source function when creating a user-defined function. Because the function accepts any data type as an argument, it is not necessary to create additional signatures to support user-defined distinct types.

The DATAPARTITIONNUM function cannot be used within check constraints or in the definition of generated columns (SQLSTATE 42881). The DATAPARTITIONNUM function cannot be used in a materialized query table (MQT) definition (SQLSTATE 428EC).

Example:

- ```
SELECT DATAPARTITIONNUM (EMPNO)
FROM EMPLOYEE
```

To convert a sequence number that is returned by DATAPARTITIONNUM (for example, 0) to a data partition name that can be used in other SQL statements (such as, for example, ALTER TABLE...DETACH PARTITION), you can query the SYSCAT.DATAPARTITIONS catalog view. Include the SEQNO obtained from DATAPARTITIONNUM in the WHERE clause, as shown in the following example.

```
SELECT DATAPARTITIONNAME
FROM SYSCAT.DATAPARTITIONS
WHERE TABNAME = 'EMPLOYEE' AND SEQNO = 0
```

results in the value 'PART0'.

## DATE

►►—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 string representation of a date or timestamp, or a string of length 7 that is not a CLOB, LONG VARCHAR, DBCLOB, or LONG VARGRAPHIC.

Only Unicode databases support an argument that is a graphic string representation of a date or a timestamp. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

If the argument is a string of length 7, it must represent a valid date in the form *yyyymmdd*, where *yyyy* are digits denoting a year, and *mmdd* 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 string with a length of 7:
  - The result is the date represented by the string.

Examples:

Assume that the column RECEIVED (timestamp) has an internal value equivalent to '1988-12-25-17.12.30.000000'.

- This example results in an internal representation of '1988-12-25'.  
`DATE(RECEIVED)`
- 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(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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- 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).

## DAY

```
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 (for example, 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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is VARCHAR(100). The result can be null; if the argument is null, the result is the null value.

## DAYOFWEEK

►►—DAYOFWEEK—(—*expression*—)—————►►

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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

## DAYOFWEEK\_ISO

►►—DAYOFWEEK\_ISO—(—*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 Monday.

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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

## 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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

## 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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The 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'.

## DAYS

```
SELECT DAYS(PRENDATE) - DAYS(PRSTDATE)
 INTO :EDUCATION_DAYS
 FROM PROJECT
 WHERE PROJNO = 'IF2000'
```

Results in EDUCATION\_DAYS being set to 396.

- 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.

In a Unicode database, if a supplied argument is a character string, it is first converted to a graphic string before the function is executed. When the output string is truncated, such that the last character is a high surrogate, that surrogate is either:

- Left as is, if the supplied argument is a character string
- Converted to the blank character (X'0020'), if the supplied argument is a graphic string

Do not rely on these behaviors, because they might change in a future release.

The result of the function is a DBCLOB. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

*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.

## DBPARTITIONNUM

►► DBPARTITIONNUM ( *column-name* )

The schema is SYSIBM.



The DBPARTITIONNUM function returns the database partition number for a row. For example, if used in a SELECT clause, it returns the database partition number for each row in the result set.

The argument must be the qualified or unqualified name of any column in the table. Because row-level information is returned, the result is the same regardless of which column is specified. The column can have any data type.

If *column-name* references a column in a view, the expression for the column in the view must reference a column of the underlying base table, and the view must be deletable. A nested or common table expression follows the same rules as a view.

The specific row (and table) for which the database partition number is returned by the DBPARTITIONNUM function is determined from the context of the SQL statement that uses the function.

The database partition number returned on transition variables and tables is derived from the current transition values of the distribution key columns. For example, in a before insert trigger, the function returns the projected database partition number, given the current values of the new transition variables. However, the values of the distribution key columns might be modified by a subsequent before insert trigger. Thus, the final database partition number of the row when it is inserted into the database might differ from the projected value.

The data type of the result is INTEGER and is never null. If there is no db2nodes.cfg file, the result is 0.

This function cannot be used as a source function when creating a user-defined function. Because the function accepts any data type as an argument, it is not necessary to create additional signatures to support user-defined distinct types.

The DBPARTITIONNUM function cannot be used on replicated tables, within check constraints, or in the definition of generated columns (SQLSTATE 42881).

For compatibility with previous versions of DB2, NODENUMBER can be specified in place of DBPARTITIONNUM.

Examples:

- Count the number of instances in which the row for a given employee in the EMPLOYEE table is on a different database partition than the description of the employee's department in the DEPARTMENT table.

```
SELECT COUNT(*) FROM DEPARTMENT D, EMPLOYEE E
WHERE D.DEPTNO=E.WORKDEPT
AND DBPARTITIONNUM(E.LASTNAME) <> DBPARTITIONNUM(D.DEPTNO)
```

- Join the EMPLOYEE and DEPARTMENT tables so that the rows of the two tables are on the same database partition.

```
SELECT * FROM DEPARTMENT D, EMPLOYEE E
WHERE DBPARTITIONNUM(E.LASTNAME) = DBPARTITIONNUM(D.DEPTNO)
```

- Using a before trigger on the EMPLOYEE table, log the employee number and the projected database partition number of any new row in the EMPLOYEE table in a table named EMPINSERTLOG1.

```
CREATE TRIGGER EMPINSLOGTRIG1
BEFORE INSERT ON EMPLOYEE
REFERENCING NEW AS NEWTABLE
```

## DBPARTITIONNUM

```
FOR EACH ROW
INSERT INTO EMPINSERTLOG1
VALUES (NEWTABLE.EMPNO, DBPARTITIONNUM
(NEWTABLE.EMPNO))
```

## DECFLOAT

### Numeric to Decimal floating-point::

|—DECFLOAT—(—*numeric-expression*— $\left[ \begin{array}{l} , -34 \\ , -16 \end{array} \right]$ —)——|

### Character to Decimal floating-point::

|—DECFLOAT—(—*string-expression*— $\left[ \begin{array}{l} , -34 \\ , -16 \end{array} \right]$ —)——|

$\left[ \begin{array}{l} , -34 \\ , -16 \end{array} \right]$   $\left[ \begin{array}{l} , -decimal-character \\ , -decimal-character \end{array} \right]$

The schema is SYSIBM.

The DECFLOAT function returns a decimal floating-point representation of a number or a string representation of a number.

*numeric-expression*

An expression that returns a value of any built-in numeric data type.

*string-expression*

An expression that returns a character string value whose length is not greater than the maximum length of a character constant (4000 bytes). It cannot have a CLOB or LONG VARCHAR data type. Leading and trailing blanks are removed from the string. The resulting substring must conform to the rules for forming an SQL integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018) and not be greater than 42 bytes (SQLSTATE 42820)..

### 34 or 16

Specifies the number of digits of precision for the result. The default is 34.

*decimal-character*

Specifies the single-byte character constant used to delimit the decimal digits in *character-expression* from the whole part of the number. The character cannot be a digit, plus (+), minus (-), or blank, and it can appear at most once in *character-expression*.

The result is the same number that would result from `CAST(string-expression AS DECFLOAT(n))` or `CAST(numeric-expression AS DECFLOAT(n))`. Leading and trailing blanks are removed from the string.

The result of the function is a decimal floating-point number with the implicitly or explicitly specified number of digits of precision. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

If necessary, the source is rounded to the precision of the target. The CURRENT DECFLOAT ROUNDING MODE special register determines the rounding mode.

**Note:** To increase the portability of applications, use the CAST specification.

**Note:** All numeric values are interpreted as integer, decimal, or floating-point constants and then cast to decimal floating-point. The use of a floating-point constant can result in round-off errors and is therefore strongly discouraged. Use the string to decimal floating-point version of the DECFLOAT function instead.

Example:

- Use the DECFLOAT function in order to force a DECFLOAT data type to be returned in a select-list for the EDLEVEL column (data type = SMALLINT) in the EMPLOYEE table. The EMPNO column should also appear in the select list.

```
SELECT EMPNO, DECFLOAT(EDLEVEL,16)
FROM EMPLOYEE
```

## DECIMAL

### Numeric to Decimal:

```
DECIMAL(numeric-expression)
DECIMAL(numeric-expression, precision-integer)
DECIMAL(numeric-expression, precision-integer, scale-integer)
```

### Character to Decimal:

```
DECIMAL(character-expression)
DECIMAL(character-expression, precision-integer)
DECIMAL(character-expression, precision-integer, scale-integer, decimal-character)
```

### Datetime to Decimal:

```
DECIMAL(datetime-expression)
DECIMAL(datetime-expression, precision-integer)
DECIMAL(datetime-expression, precision-integer, scale-integer)
```

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 an integer number
- A character string representation of a floating-point number
- A character string representation of a decimal floating-point number
- A datetime value if the argument is a date, time, or timestamp

In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

## DECIMAL

The result of the function is a decimal number with precision  $p$  and scale  $s$ , where  $p$  and  $s$  are the second and third arguments, respectively. 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 *precision-integer* depends on the data type of *numeric-expression*:

- 31 for decimal floating-point
- 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 precision  $p$  and scale  $s$ , where  $p$  and  $s$  are the second and third arguments, respectively. 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 used to delimit the decimal digits in *character-expression* from the whole part of the number. The character cannot be a digit, plus (+), minus (-), or blank, and it can appear at most once in *character-expression* (SQLSTATE 42815).

The result is a decimal number with precision  $p$  and scale  $s$ , where  $p$  and  $s$  are the second and third arguments, respectively. Digits are truncated from

the end of the decimal number if the number of digits to the right of the decimal character is greater than the scale. An error occurs if the number of significant digits to the left of the decimal character (the whole part of the number) in *character-expression* is greater than  $p-s$  (SQLSTATE 22003). The default decimal character is not valid in the substring if a different value for the *decimal-character* argument is specified (SQLSTATE 22018).

### Datetime to Decimal

#### *datetime-expression*

An expression that is of one of the following data types:

- DATE. The result is a DECIMAL(8,0) value representing the date as *yyyymmdd*.
- TIME. The result is a DECIMAL(6,0) value representing the time as *hhmmss*.
- TIMESTAMP. The result is a DECIMAL(20,6) value representing the timestamp as *yyyymmddhhmmss.nnnnnn*.

This function allows the user to specify a precision, or a precision and a scale. However, a scale cannot be specified without specifying a precision. The default value for (precision,scale) is (8,0) for DATE, (6,0) for TIME, and (20,6) for TIMESTAMP.

The result is a decimal number with precision  $p$  and scale  $s$ , where  $p$  and  $s$  are the second and third arguments, respectively. Digits are truncated from the end if the number of digits to the right of the decimal character is greater than the scale. An error occurs if the number of significant digits to the left of the decimal character (the whole part of the number) in *datetime-expression* is greater than  $p-s$  (SQLSTATE 22003).

DEC can be specified in place of DECIMAL.

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).

```
SELECT PRSTDATE + 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, '.')
```

## DECIMAL

This fails because a period (.) is specified as the decimal character, but a comma (,) appears in the first argument as a delimiter.

- Assume that the column STARTING (time) has an internal value equivalent to '12:10:00'.

**DECIMAL**(STARTING)

results in the value 121 000.

- Assume that the column RECEIVED (timestamp) has an internal value equivalent to '1988-12-22-14.07.21.136421'.

**DECIMAL**(RECEIVED)

results in the value 19 881 222 140 721.136421.

- The following table shows the decimal result and resulting precision and scale for various datetime input values.

| <b>DECIMAL(arguments)</b>                  | <b>Precision and Scale</b> | <b>Result</b>         |
|--------------------------------------------|----------------------------|-----------------------|
| DECIMAL(2000-03-21)                        | (8,0)                      | 20000321              |
| DECIMAL(2000-03-21, 10)                    | (10,0)                     | 20000321              |
| DECIMAL(2000-03-21, 12, 2)                 | (12,2)                     | 20000321.00           |
| DECIMAL(12:02:21)                          | (6,0)                      | 120221                |
| DECIMAL(12:02:21, 10)                      | (10,0)                     | 120221                |
| DECIMAL(12:02:21, 10, 2)                   | (10,2)                     | 120221.00             |
| DECIMAL(2000-03-21-12.02.21.123456)        | (20, 6)                    | 20000321120221.123456 |
| DECIMAL(2000-03-21-12.02.21.123456, 23)    | (23, 6)                    | 20000321120221.123456 |
| DECIMAL(2000-03-21-12.02.21.123456, 23, 4) | (23, 4)                    | 20000321120221.1234   |

## DECODE

→→ **DECODE** ( *expression1* , *expression2* , *result-expression* [ , *else-expression* ] ) →→

The schema is SYSIBM.

The DECODE function compares each *expression2* to *expression1*. If *expression1* is equal to *expression2*, or both *expression1* and *expression2* are null, the value of the following *result-expression* is returned. If no *expression2* matches *expression1*, the value of *else-expression* is returned; otherwise a null value is returned.

The DECODE function is similar to the CASE expression except for the handling of null values:

- A null value of *expression1* will match a corresponding null value of *expression2*.
- If the NULL keyword is used as an argument in the DECODE function, it must be cast to an appropriate data type.

The rules for determining the result type of a DECODE expression are based on the corresponding CASE expression.

Examples:

The DECODE expression:

```
DECODE (c1, 7, 'a', 6, 'b', 'c')
```

achieves the same result as the following CASE expression:

```
CASE c1
 WHEN 7 THEN 'a'
 WHEN 6 THEN 'b'
 ELSE 'c'
END
```

Similarly, the DECODE expression:

```
DECODE (c1, var1, 'a', var2, 'b')
```

where the values of c1, var1, and var2 could be null values, achieves the same result as the following CASE expression:

```
CASE
 WHEN c1 = var1 OR (c1 IS NULL AND var1 IS NULL) THEN 'a'
 WHEN c1 = var2 OR (c1 IS NULL AND var2 IS NULL) THEN 'b'
 ELSE NULL
END
```

Consider also the following query:

```
SELECT ID, DECODE(STATUS, 'A', 'Accepted',
 'D', 'Denied',
 CAST(NULL AS VARCHAR(1)), 'Unknown',
 'Other')
FROM CONTRACTS
```

Here is the same statement using a CASE expression:

```
SELECT ID,
 CASE
 WHEN STATUS = 'A' THEN 'Accepted'
 WHEN STATUS = 'D' THEN 'Denied'
 WHEN STATUS IS NULL THEN 'Unknown'
 ELSE 'Other'
 END
FROM CONTRACTS
```

## DECRYPT\_BIN and DECRYPT\_CHAR

```

 ──┬── DECRYPT_BIN ──┬── (—encrypted-data—)──┬──
 └── DECRYPT_CHAR ─┘ └── ,—password-string-expression ─┘

```

The schema is SYSIBM.

The DECRYPT\_BIN and DECRYPT\_CHAR functions both return a value that is the result of decrypting *encrypted-data*. The password used for decryption is either the *password-string-expression* value or the encryption password value that was assigned by the SET ENCRYPTION PASSWORD statement. The DECRYPT\_BIN and DECRYPT\_CHAR functions can only decrypt values that are encrypted using the ENCRYPT function (SQLSTATE 428FE).

*encrypted-data*

An expression that returns a CHAR FOR BIT DATA or VARCHAR FOR BIT

## DECRYPT\_BIN and DECRYPT\_CHAR

DATA value as a complete, encrypted data string. The data string must have been encrypted using the ENCRYPT function.

*password-string-expression*

An expression that returns a CHAR or VARCHAR value with at least 6 bytes and no more than 127 bytes (SQLSTATE 428FC). This expression must be the same password used to encrypt the data (SQLSTATE 428FD). If the value of the password argument is null or not provided, the data will be decrypted using the encryption password value that was assigned for the session by the SET ENCRYPTION PASSWORD statement (SQLSTATE 51039).

The result of the DECRYPT\_BIN function is VARCHAR FOR BIT DATA. The result of the DECRYPT\_CHAR function is VARCHAR. If *encrypted-data* included a hint, the hint is not returned by the function. The length attribute of the result is the length of the data type of *encrypted-data* minus 8 bytes. The actual length of the value returned by the function will match the length of the original string that was encrypted. If *encrypted-data* includes bytes beyond the encrypted string, these bytes are not returned by the function.

If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

If the data is decrypted on a different system, which uses a code page that is different from the code page in which the data was encrypted, expansion might occur when converting the decrypted value to the database code page. In such situations, the *encrypted-data* value should be cast to a VARCHAR string with a larger number of bytes.

Examples:

- Use the SET ENCRYPTION PASSWORD statement to set an encryption password for the session.

```
CREATE TABLE EMP (SSN VARCHAR(24) FOR BIT DATA);
SET ENCRYPTION PASSWORD = 'Ben123';
INSERT INTO EMP (SSN) VALUES ENCRYPT('289-46-8832');
SELECT DECRYPT_CHAR(SSN)
FROM EMP;
```

This query returns the value '289-46-8832'.

- Pass the encryption password explicitly.

```
INSERT INTO EMP (SSN) VALUES ENCRYPT('289-46-8832', 'Ben123', '');
SELECT DECRYPT_CHAR(SSN, 'Ben123')
FROM EMP;
```

This query returns the value '289-46-8832'.

## DEGREES

►►—DEGREES—(—*expression*—)—————►►

The schema is SYSIBM. (The SYSFUN version of the DEGREES function continues to be available.)

The DEGREES function returns the number of degrees of the argument, which is an angle expressed in radians.



The argument can be any built-in numeric data type. If the argument is decimal floating-point, the operation is performed in decimal floating-point; otherwise, the argument is converted to double-precision floating-point for processing by the function.

If the argument is `DECFLOAT(n)`, the result is `DECFLOAT(n)`; otherwise, the result is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example:

- Assume that `RAD` is a `DECIMAL(4,3)` host variable with a value of 3.142.

```
VALUES DEGREES(:RAD)
```

Returns the approximate value 180.0.

## DEREF

►►—DEREF—(—*expression*—)—————►

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`).

Examples:

Assume that `EMPLOYEE` is a table of type `EMP`, and that its object identifier column is named `EMPID`. Then the following query returns an object of type `EMP` (or one of its subtypes), for each row of the `EMPLOYEE` table (and its subtables). This query requires `SELECT` privilege on `EMPLOYEE` and all its subtables.

```
SELECT DEREF(EMPID) FROM EMPLOYEE
```

## DIFFERENCE

►►—DIFFERENCE—(—*expression*—, —*expression*—)—————►

The schema is `SYSFUN`.

## DIFFERENCE

Returns a value from 0 to 4 representing the difference between the sounds of two strings based on applying the SOUNDEX function to the strings. A value of 4 is the best possible sound match.

The arguments can be character strings that are either CHAR or VARCHAR not exceeding 4000 bytes. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed. The function interprets data that is passed to it as if it were ASCII characters, even if it is encoded in UTF-8.

The result of the function is 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:

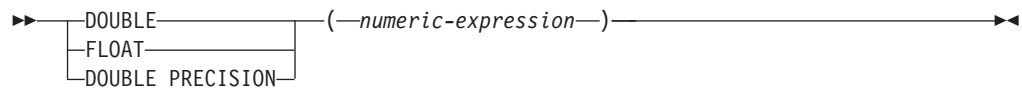
```
DIGITS(COLUMNX)
```

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.

## DOUBLE

### Numeric to Double:



### Character String to Double:



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.

In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

### 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.

## DOUBLE

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
```

## ENCRYPT

►►—ENCRYPT—►►  
►—(—*data-string-expression*— [ ,—*password-string-expression*— [ ,—*hint-string-expression*— ] ] )—►►

The schema is SYSIBM.

The ENCRYPT function returns a value that is the result of encrypting *data-string-expression*. The password used for encryption is either the *password-string-expression* value or the encryption password value that was assigned by the SET ENCRYPTION PASSWORD statement. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

*data-string-expression*

An expression that returns a CHAR or a VARCHAR value that is to be encrypted. The length attribute for the data type of *data-string-expression* is limited to 32663 without a *hint-string-expression* argument, and 32631 when the *hint-string-expression* argument is specified (SQLSTATE 42815).

*password-string-expression*

An expression that returns a CHAR or a VARCHAR value with at least 6 bytes and no more than 127 bytes (SQLSTATE 428FC). The value represents the password used to encrypt *data-string-expression*. If the value of the password argument is null or not provided, the data will be encrypted using the encryption password value that was assigned for the session by the SET ENCRYPTION PASSWORD statement (SQLSTATE 51039).

*hint-string-expression*

An expression that returns a CHAR or a VARCHAR value with at most 32 bytes that will help data owners remember passwords (for example, 'Ocean' as a hint to remember 'Pacific'). If a hint value is given, the hint is embedded into the result and can be retrieved using the GETHINT function. If this argument is null or not provided, no hint will be embedded in the result.

The result data type of the function is VARCHAR FOR BIT DATA.

- When the optional hint parameter is specified, the length attribute of the result is equal to the length attribute of the unencrypted data + 8 bytes + the number of bytes until the next 8-byte boundary + 32 bytes for the length of the hint.
- When the optional hint parameter is not specified, the length attribute of the result is equal to the length attribute of the unencrypted data + 8 bytes + the number of bytes until the next 8-byte boundary.

If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

Note that the encrypted result is longer than the *data-string-expression* value. Therefore, when assigning encrypted values, ensure that the target is declared with sufficient size to contain the entire encrypted value.

### Notes:

- **Encryption Algorithm:** The internal encryption algorithm is RC2 block cipher with padding; the 128-bit secret key is derived from the password using an MD5 message digest.
- **Encryption Passwords and Data:** Password management is the user's responsibility. Once the data is encrypted, only the password that was used when encrypting it can be used to decrypt it (SQLSTATE 428FD).  
The encrypted result might contain null terminator and other unprintable characters. Any assignment or cast to a length that is shorter than the suggested data length might result in failed decryption in the future, and lost data. Blanks are valid encrypted data values that might be truncated when stored in a column that is too short.
- **Administration of encrypted data:** Encrypted data can only be decrypted on servers that support the decryption functions corresponding to the ENCRYPT function. Therefore, replication of columns with encrypted data should only be done to servers that support the DECRYPT\_BIN or the DECRYPT\_CHAR function.

Examples:

- Use the SET ENCRYPTION PASSWORD statement to set an encryption password for the session.

```
CREATE TABLE EMP (SSN VARCHAR(24) FOR BIT DATA);
SET ENCRYPTION PASSWORD = 'Ben123';
INSERT INTO EMP(SSN) VALUES ENCRYPT('289-46-8832');
```

- Pass the encryption password explicitly.

```
INSERT INTO EMP(SSN) VALUES ENCRYPT('289-46-8832', 'Ben123');
```

- Define a password hint.

```
INSERT INTO EMP(SSN) VALUES ENCRYPT('289-46-8832', 'Pacific', 'Ocean');
```

## 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.

## EVENT\_MON\_STATE

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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result 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 SYSIBM. (The SYSFUN version of the EXP function continues to be available.)

The EXP function returns a value that is the base of the natural logarithm (e) raised to a power specified by the argument. The EXP and LN functions are inverse operations.

The argument must be an expression that returns a value of any built-in numeric data type. If the argument is decimal floating-point, the operation is performed in decimal floating-point; otherwise, the argument is converted to double-precision floating-point for processing by the function.

If the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*); otherwise, the result is a double-precision floating-point number. 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 E is a DECIMAL(10,9) host variable with a value of 3.453789832.

```
VALUES EXP(:E)
```

Returns the DOUBLE value +3.16200000069145E+001.

## 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.

## FLOOR

►►—FLOOR—(*—expression—*)—►►

The schema is SYSIBM. (The SYSFUN version of the FLOOR function continues to be available.)

Returns the largest integer value less than or equal to the argument.

The result of the function has the same data type and length attribute as the argument except that the scale is 0 if the argument is DECIMAL. For example, an argument with a data type of DECIMAL(5,5) returns DECIMAL(5,0).

The result can be null if the argument can be null or if the argument is not a decimal floating-point number and the database is configured with DFT\_SQLMATHWARN set to YES; the result is the null value if the argument is null.

Examples:

- Use the FLOOR function to truncate any digits to the right of the decimal point.

```
SELECT FLOOR(SALARY)
FROM EMPLOYEE
```

- Use the FLOOR function on both positive and negative numbers.

```
VALUES FLOOR(3.5), FLOOR(3.1),
 FLOOR(-3.1), FLOOR(-3.5)
```

This example returns 3., 3., -4., and -4., respectively.

## 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. (The system clock is used to generate the internal Universal Time, Coordinated (UTC) timestamp along with the database partition number on which the function executes. Adjustments that move the actual system clock backward could result in duplicate values.) The function is defined as not-deterministic.

There are no arguments to this function (the empty parentheses must be specified).

## GENERATE\_UNIQUE

The result of the function is a unique value that includes the internal form of the Universal Time, Coordinated (UTC) and the database partition number where the function was processed. The result cannot be null.

The result of this function can be used to provide unique values in a table. Each successive value will be greater than the previous value, providing a sequence that can be used within a table. The value includes the database partition number where the function executed so that a table partitioned across multiple database partitions also has unique values in some sequence. The sequence is based on the time the function was executed.

This function differs from using the special register CURRENT\_TIMESTAMP in that a unique value is generated for each row of a multiple row insert statement 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),
TEXT VARCHAR(1000))
INSERT INTO EMP_UPDATE
VALUES (GENERATE_UNIQUE(), '000020', 'Update entry...'),
(GENERATE_UNIQUE(), '000050', 'Update entry...')
```

This table will have a unique identifier for each row provided that the UNIQUE\_ID column is always set using GENERATE\_UNIQUE. This can be done by introducing a trigger on the table.

```
CREATE TRIGGER EMP_UPDATE_UNIQUE
NO CASCADE BEFORE INSERT ON EMP_UPDATE
REFERENCING NEW AS NEW_UPD
FOR EACH ROW
SNEW_UPD.UNIQUE_ID = GENERATE_UNIQUE()
```

With this trigger defined, the previous INSERT statement could be issued without the first column as follows.

```
INSERT INTO EMP_UPDATE (EMPNO, TEXT)
VALUES ('000020', 'Update entry 1...'),
('000050', 'Update entry 2...')
```

The timestamp (in UTC) for when a row was added to EMP\_UPDATE can be returned using:

```
SELECT TIMESTAMP (UNIQUE_ID), EMPNO, TEXT
FROM EMP_UPDATE
```

Therefore, there is no need to have a timestamp column in the table to record when a row is inserted.



## GETHINT

►► GETHINT (—*encrypted-data*—) ►►

The schema is SYSIBM.

The GETHINT function will return the password hint if one is found in the *encrypted-data*. A password hint is a phrase that will help data owners remember passwords; for example, 'Ocean' as a hint to remember 'Pacific'. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

*encrypted-data*

An expression that returns a CHAR FOR BIT DATA or VARCHAR FOR BIT DATA value that is a complete, encrypted data string. The data string must have been encrypted using the ENCRYPT function (SQLSTATE 428FE).

The result of the function is VARCHAR(32). The result can be null; if the hint parameter was not added to the *encrypted-data* by the ENCRYPT function or the first argument is null, the result is the null value.

Example:

In this example the hint 'Ocean' is stored to help the user remember the encryption password 'Pacific'.

```
INSERT INTO EMP (SSN) VALUES ENCRYPT('289-46-8832', 'Pacific','Ocean');
SELECT GETHINT(SSN)
FROM EMP;
```

The value returned is 'Ocean'.

## GRAPHIC

**Graphic to Graphic:**

►► GRAPHIC (—*graphic-expression*— [ , —*integer*— ] ) ►►

**Character to Graphic:**

►► GRAPHIC (—*character-expression*—) ►►

**Datetime to Graphic:**

►► GRAPHIC (—*datetime-expression*— [ , ISO | USA | EUR | JIS | LOCAL ] ) ►►

## GRAPHIC

The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature.

The GRAPHIC function returns a fixed-length graphic string representation of:

- A graphic string, if the first argument is any type of graphic string
- A character string, converting single-byte characters to double-byte characters, if the first argument is any type of character string
- A datetime value (Unicode database only), if the first argument is a date, time, or timestamp

In a Unicode database, if a supplied argument is a character string, it is first converted to a graphic string before the function is executed. When the output string is truncated, such that the last character is a high surrogate, that surrogate is converted to the blank character (X'0020'). Do not rely on this behavior, because it might change in a future release.

The result of the function is a fixed-length graphic string (GRAPHIC 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 Graphic

*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 a value is not specified, the length attribute of the result is the same as the length attribute of the first argument.

### Character to Graphic

*character-expression*

An expression whose value must be of a character string data type other than LONG VARCHAR or CLOB, and whose maximum length is 16 336 bytes.

The length attribute of the result is equal to the length attribute of the argument.

### Datetime to Graphic

*datetime-expression*

An expression that is of one of the following three data types:

**date** The result is the graphic string representation of the date in the format specified by the second argument. The length of the result is 10. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

**time** The result is the graphic string representation of the time in the format specified by the second argument. The length of the result is 8. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

**timestamp**

The result is the graphic string representation of the timestamp. The length of the result is 26. The second argument is not applicable and must not be specified (SQLSTATE 42815).

The code page of the string is the code page of the database at the application server.

## GREATEST

►► GREATEST (—*expression*—, —*expression*—) ◄◄

The schema is SYSIBM.

The GREATEST function returns the maximum value in a set of values.

The arguments must be compatible and each argument must be an expression that returns a value of any data type other than ARRAY, LOB, LONG VARCHAR, LONG VARGRAPHIC, XML, a distinct type based on any of these types, or a structured type (SQLSTATE 42815). This function cannot be used as a source function when creating a user-defined function. Because this function accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by all the operands based on the rules for result data types.

The result of the function is the largest argument value. If at least one argument can be null, the result can be null; if either argument is null, the result is the null value.

The GREATEST scalar function is a synonym for the MAX scalar function.

Examples:

Assume that table T1 contains three columns C1, C2, and C3 with values 1, 7, and 4, respectively. The query:

```
SELECT GREATEST (C1, C2, C3) FROM T1
```

returns 7.

If column C3 has a value of NULL instead of 4, the same query returns NULL.

## HASHEDVALUE

►► HASHEDVALUE (—*column-name*—) ◄◄

The schema is SYSIBM.

The HASHEDVALUE function returns the distribution map index of the row obtained by applying the partitioning function on the distribution key value of the row. For example, if used in a SELECT clause, it returns the distribution map index for each row of the table that was used to form the result of the SELECT statement.

## HASHEDVALUE

The distribution map index returned on transition variables and tables is derived from the current transition values of the distribution key columns. For example, in a before insert trigger, the function will return the projected distribution map index given the current values of the new transition variables. However, the values of the distribution key columns may be modified by a subsequent before insert trigger. Thus, the final distribution map index of the row when it is inserted into the database may differ from the projected value.

The argument must be the qualified or unqualified name of a column in a table. The column can have any data type. (This function cannot be used as a source function when creating a user-defined function. Because it accepts any data type as an argument, it is not necessary to create additional signatures to support user-defined distinct types.) 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.

The specific row (and table) for which the distribution map index is returned by the HASHEDVALUE function is determined from the context of the SQL statement that uses the function.

The data type of the result is INTEGER in the range 0 to 4095. For a table with no distribution key, the result is always 0. A null value is never returned. Since row-level information is returned, the results are the same, regardless of which column is specified for the table.

The HASHEDVALUE function cannot be used on replicated tables, within check constraints, or in the definition of generated columns (SQLSTATE 42881).

For compatibility with versions earlier than Version 8, the function name PARTITION can be substituted for HASHEDVALUE.

Example:

- List the employee numbers (EMPNO) from the EMPLOYEE table for all rows with a distribution map index of 100.

```
SELECT EMPNO FROM EMPLOYEE
WHERE HASHEDVALUE(PHONENO) = 100
```

- Log the employee number and the projected distribution map index of the new row into a table called EMPINSERTLOG2 for any insertion of employees by creating a before trigger on the table EMPLOYEE.

```
CREATE TRIGGER EMPINSLOGTRIG2
BEFORE INSERT ON EMPLOYEE
REFERENCING NEW AS NEWTABLE
FOR EACH ROW
INSERT INTO EMPINSERTLOG2
VALUES(NEWTABLE.EMPNO, HASHEDVALUE(NEWTABLE.EMPNO))
```

## 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.
- 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).

## HEX

```
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 byte long string '42'.
- Suppose COL\_3 is a column with a data type of decimal(6,2) and a value of 40.1. An eight byte long string '0004010C' is the result of applying the HEX function to the internal representation of the decimal value, 40.1.

## 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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- 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
```

## IDENTITY\_VAL\_LOCAL

►►—**IDENTITY\_VAL\_LOCAL**—(—)—————►►

The schema is SYSIBM.

The IDENTITY\_VAL\_LOCAL function is a non-deterministic function that returns the most recently assigned value for an identity column, where the assignment occurred as a result of a single INSERT statement using a VALUES clause. The function has no input parameters.

The result is a DECIMAL(31,0), regardless of the actual data type of the corresponding identity column.

The value returned by the function is the value assigned to the identity column of the table identified in the most recent single row insert operation. The INSERT statement must contain a VALUES clause on a table containing an identity column. The INSERT statement must also be issued at the same level; that is, the value must be available locally at the level it was assigned, until it is replaced by the next assigned value. (A new level is initiated each time a trigger or routine is invoked.)

The assigned value is either a value supplied by the user (if the identity column is defined as GENERATED BY DEFAULT), or an identity value generated by the database manager.

The function returns a null value if a single row INSERT statement with a VALUES clause has not been issued at the current processing level against a table containing an identity column.

The result of the function is not affected by the following:

- A single row INSERT statement with a VALUES clause for a table without an identity column
- A multiple row INSERT statement with a VALUES clause
- An INSERT statement with a fullselect
- A ROLLBACK TO SAVEPOINT statement

Notes:

- Expressions in the VALUES clause of an INSERT statement are evaluated prior to the assignments for the target columns of the insert operation. Thus, an invocation of an IDENTITY\_VAL\_LOCAL function inside the VALUES clause of an INSERT statement will use the most recently assigned value for an identity column from a previous insert operation. The function returns the null value if no previous single row INSERT statement with a VALUES clause for a table containing an identity column has been executed within the same level as the IDENTITY\_VAL\_LOCAL function.
- The identity column value of the table for which the trigger is defined can be determined within a trigger by referencing the trigger transition variable for the identity column.
- The result of invoking the IDENTITY\_VAL\_LOCAL function from within the trigger condition of an insert trigger is a null value.
- It is possible that multiple before or after insert triggers exist for a table. In this case, each trigger is processed separately, and identity values assigned by one triggered action are not available to other triggered actions using the IDENTITY\_VAL\_LOCAL function. This is true even though the multiple triggered actions are conceptually defined at the same level.
- It is not generally recommended to use the IDENTITY\_VAL\_LOCAL function in the body of a before insert trigger. The result of invoking the IDENTITY\_VAL\_LOCAL function from within the triggered action of a before insert trigger is the null value. The value for the identity column of the table for

## IDENTITY\_VAL\_LOCAL

which the trigger is defined cannot be obtained by invoking the `IDENTITY_VAL_LOCAL` function within the triggered action of a before insert trigger. However, the value for the identity column can be obtained in the triggered action by referencing the trigger transition variable for the identity column.

- The result of invoking the `IDENTITY_VAL_LOCAL` function from within the triggered action of an after insert trigger is the value assigned to an identity column of the table identified in the most recent single row insert operation invoked in the same triggered action that had a `VALUES` clause for a table containing an identity column. (This applies to both `FOR EACH ROW` and `FOR EACH STATEMENT` after insert triggers.) If a single row `INSERT` statement with a `VALUES` clause for a table containing an identity column was not executed within the same triggered action, prior to the invocation of the `IDENTITY_VAL_LOCAL` function, the function returns a null value.
- Because `IDENTITY_VAL_LOCAL` is a non-deterministic function, the result of invoking this function within the `SELECT` statement of a cursor can vary for each `FETCH` statement.
- The assigned value is the value actually assigned to the identity column (that is, the value that would be returned on a subsequent `SELECT` statement). This value is not necessarily the value provided in the `VALUES` clause of the `INSERT` statement, or a value generated by the database manager. The assigned value could be a value specified in a `SET` transition variable statement, within the body of a before insert trigger, for a trigger transition variable associated with the identity column.
- The value returned by the function following a failed single row `INSERT` statement with a `VALUES` clause into a table with an identity column is unpredictable. It could be the value that would have been returned from the function had it been invoked prior to the failed insert operation, or it could be the value that would have been assigned had the insert operation succeeded. The actual value returned depends on the point of failure, and is therefore unpredictable.

Examples:

Example 1: Create two tables, T1 and T2, each with an identity column named C1. Start the identity sequence for table T2 at 10. Insert some values for C2 into T1.

```
CREATE TABLE T1
(C1 INTEGER GENERATED ALWAYS AS IDENTITY,
 C2 INTEGER)

CREATE TABLE T2
(C1 DECIMAL(15,0) GENERATED BY DEFAULT AS IDENTITY (START WITH 10),
 C2 INTEGER)

INSERT INTO T1 (C2) VALUES (5)

INSERT INTO T1 (C2) VALUES (6)

SELECT * FROM T1
```

This query returns:

| C1 | C2 |
|----|----|
| 1  | 5  |
| 2  | 6  |



Insert a single row into table T2, where column C2 gets its value from the IDENTITY\_VAL\_LOCAL function.

```
INSERT INTO T2 (C2) VALUES (IDENTITY_VAL_LOCAL())

SELECT * FROM T2
```

This query returns:

```
C1 C2

 10. 2
```

Example 2: In a nested environment involving a trigger, use the IDENTITY\_VAL\_LOCAL function to retrieve the identity value assigned at a particular level, even though there might have been identity values assigned at lower levels. Assume that there are three tables, EMPLOYEE, EMP\_ACT, and ACCT\_LOG. There is an after insert trigger defined on EMPLOYEE that results in additional inserts into the EMP\_ACT and ACCT\_LOG tables.

```
CREATE TABLE EMPLOYEE
 (EMPNO SMALLINT GENERATED ALWAYS AS IDENTITY (START WITH 1000),
 NAME CHAR(30),
 SALARY DECIMAL(5,2),
 DEPTNO SMALLINT)

CREATE TABLE EMP_ACT
 (ACNT_NUM SMALLINT GENERATED ALWAYS AS IDENTITY (START WITH 1),
 EMPNO SMALLINT)

CREATE TABLE ACCT_LOG
 (ID SMALLINT GENERATED ALWAYS AS IDENTITY (START WITH 100),
 ACNT_NUM SMALLINT,
 EMPNO SMALLINT)

CREATE TRIGGER NEW_HIRE
 AFTER INSERT ON EMPLOYEE
 REFERENCING NEW AS NEW_EMP
 FOR EACH ROW
 BEGIN ATOMIC
 INSERT INTO EMP_ACT (EMPNO) VALUES (NEW_EMP.EMPNO);
 INSERT INTO ACCT_LOG (ACNT_NUM, EMPNO)
 VALUES (IDENTITY_VAL_LOCAL(), NEW_EMP.EMPNO);
 END
```

The first triggered insert operation inserts a row into the EMP\_ACT table. The statement uses a trigger transition variable for the EMPNO column of the EMPLOYEE table to indicate that the identity value for the EMPNO column of the EMPLOYEE table is to be copied to the EMPNO column of the EMP\_ACT table. The IDENTITY\_VAL\_LOCAL function could not be used to obtain the value assigned to the EMPNO column of the EMPLOYEE table, because an INSERT statement has not been issued at this level of the nesting. If the IDENTITY\_VAL\_LOCAL function were invoked in the VALUES clause of the INSERT statement for the EMP\_ACT table, it would return a null value. The insert operation against the EMP\_ACT table also results in the generation of a new identity value for the ACNT\_NUM column.

The second triggered insert operation inserts a row into the ACCT\_LOG table. The statement invokes the IDENTITY\_VAL\_LOCAL function to indicate that the identity value assigned to the ACNT\_NUM column of the EMP\_ACT table in the previous insert operation in the triggered action is to be copied to the ACNT\_NUM column of the ACCT\_LOG table. The EMPNO column is assigned the same value as the EMPNO column of the EMPLOYEE table.

## IDENTITY\_VAL\_LOCAL

After the following INSERT statement and all of the triggered actions have been processed:

```
INSERT INTO EMPLOYEE (NAME, SALARY, DEPTNO)
VALUES ('Rupert', 989.99, 50)
```

the contents of the three tables are as follows:

```
SELECT EMPNO, SUBSTR(NAME,1,10) AS NAME, SALARY, DEPTNO
FROM EMPLOYEE
```

| EMPNO | NAME   | SALARY | DEPTNO |
|-------|--------|--------|--------|
| 1000  | Rupert | 989.99 | 50     |

```
SELECT ACNT_NUM, EMPNO
FROM EMP_ACT
```

| ACNT_NUM | EMPNO |
|----------|-------|
| 1        | 1000  |

```
SELECT * FROM ACCT_LOG
```

| ID  | ACNT_NUM | EMPNO |
|-----|----------|-------|
| 100 | 1        | 1000  |

The result of the IDENTITY\_VAL\_LOCAL function is the most recently assigned value for an identity column at the same nesting level. After processing the original INSERT statement and all of the triggered actions, the IDENTITY\_VAL\_LOCAL function returns a value of 1000, because this is the value that was assigned to the EMPNO column of the EMPLOYEE table.

## INSERT

```
►►—INSERT—(—source-string—,—start—,—length—,—insert-string—,—)————►►
```

```
CODEUNITS16
CODEUNITS32
OCTETS
```

The schema is SYSIBM. The SYSFUN version of the INSERT function continues to be available.

The INSERT function returns a string where, beginning at *start* in *source-string*, *length* bytes have been deleted and *insert-string* has been inserted.

The INSERT function is identical to the OVERLAY function, except that the length argument is mandatory.

### *source-string*

An expression that specifies the source string. The expression must return a value that is a built-in character or graphic string data type (SQLSTATE 22011).

### *start*

An expression that returns an integer value. The integer value specifies the starting point within the source string where the deletion of bytes and the insertion of another string is to begin. The integer value must be between 1 and the length of *source-string* plus one (SQLSTATE 42815). If OCTETS is specified and the result is graphic data, the value must be an odd number between 1 and twice the length attribute of *source-string* plus one (SQLSTATE 428GC).

*length*

An expression that specifies the number of code units (in the specified string units) that are to be deleted from the source string, starting at the position identified by *start*. The value must be between 0 and the length of *source-string*, expressed in units that are either implicitly or explicitly specified (SQLSTATE 22011). If OCTETS is specified and the result is graphic data, the value must be an even number between 0 and twice the length attribute of *source-string* (SQLSTATE 428GC).

*insert-string*

An expression that specifies the string to be inserted into *source-string*, starting at the position identified by *start*. The expression must return a value that is a built-in character or graphic string data type compatible with *source-string* (SQLSTATE 22011).

**CODEUNITS16, CODEUNITS32, or OCTETS**

Specifies the string unit of *start* and *length*.

CODEUNITS16 specifies that *start* and *length* are expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *start* and *length* are expressed in 32-bit UTF-32 code units. OCTETS specifies that *start* and *length* are expressed in bytes.

If the string unit is specified as CODEUNITS16 or CODEUNITS32, and the result is a binary string or bit data, an error is returned (SQLSTATE 428GC). If the string unit is specified as OCTETS, and *insert-string* and *source-string* are binary strings, an error is returned (SQLSTATE 42815). If the string unit is specified as OCTETS, the operation is performed in the code page of the *source-string*. If a string unit is not explicitly specified, the data type of the result determines the unit that is used. If the result is graphic data, *start* and *length* are expressed in two-byte units; otherwise, they are expressed in bytes. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

The data type of the result depends on the data types of *source-string* and *insert-string*, as shown in the following table of supported type combinations.

*Table 33. Data type of the result as a function of the data types of source-string and insert-string*

| <i>source-string</i>                      | <i>insert-string</i>                                      | <b>Result</b>        |
|-------------------------------------------|-----------------------------------------------------------|----------------------|
| CHAR or VARCHAR                           | CHAR or VARCHAR                                           | VARCHAR              |
| GRAPHIC or VARGRAPHIC                     | GRAPHIC or VARGRAPHIC                                     | VARGRAPHIC           |
| CLOB                                      | CHAR, VARCHAR, or CLOB                                    | CLOB                 |
| DBCLOB                                    | GRAPHIC, VARGRAPHIC, or DBCLOB                            | DBCLOB               |
| CHAR or VARCHAR                           | CHAR FOR BIT DATA or VARCHAR FOR BIT DATA                 | VARCHAR FOR BIT DATA |
| CHAR FOR BIT DATA or VARCHAR FOR BIT DATA | CHAR, VARCHAR, CHAR FOR BIT DATA, or VARCHAR FOR BIT DATA | VARCHAR FOR BIT DATA |
| <b>For Unicode databases only:</b>        |                                                           |                      |
| CHAR or VARCHAR                           | GRAPHIC or VARGRAPHIC                                     | VARCHAR              |
| GRAPHIC or VARGRAPHIC                     | CHAR or VARCHAR                                           | VARGRAPHIC           |
| CLOB                                      | GRAPHIC, VARGRAPHIC, or DBCLOB                            | CLOB                 |

## INSERT

Table 33. Data type of the result as a function of the data types of source-string and insert-string (continued)

| source-string | insert-string          | Result |
|---------------|------------------------|--------|
| DBCLOB        | CHAR, VARCHAR, or CLOB | DBCLOB |

A *source-string* can have a length of 0; in this case, *start* must be 1 and *length* must be 0 (as implied by the bounds for *start* and *length* described above), and the result of the function is a copy of the *insert-string*.

An *insert-string* can also have a length of 0. This has the effect of deleting the code units from positions *start* to *start* + *length* - 1 from the *source-string*.

The length attribute of the result is the length attribute of *source-string* plus the length attribute of *insert-string*. The actual length of the result is  $A1 - \text{MIN}((A1 - V2 + 1), V3) + A4$ , where:

- A1 is the actual length of *source-string*
- V2 is the value of *start*
- V3 is the value of *length*
- A4 is the actual length of *insert-string*

If the actual length of the result string exceeds the maximum for the return data type, an error is returned (SQLSTATE 54006).

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples:

- Create the strings 'INSISTING', 'INSISERTING', and 'INSTING' from the string 'INSERTING' by inserting text into the middle of the existing text.

```
SELECT INSERT('INSERTING',4,2,'IS'),
 INSERT('INSERTING',4,0,'IS'),
 INSERT('INSERTING',4,2,'')
FROM SYSIBM.SYSDUMMY1
```

- Create the strings 'XXINSERTING', 'XXNSERTING', 'XXSERTING', and 'XXERTING' from the string 'INSERTING' by inserting text before the existing text, using 1 as the starting point.

```
SELECT INSERT('INSERTING',1,0,'XX'),
 INSERT('INSERTING',1,1,'XX'),
 INSERT('INSERTING',1,2,'XX'),
 INSERT('INSERTING',1,3,'XX')
FROM SYSIBM.SYSDUMMY1
```

- Create the string 'ABCABCXX' from the string 'ABCABC' by inserting text after the existing text. Because the source string is 6 characters long, set the starting position to 7 (one plus the length of the source string).

```
SELECT INSERT('ABCABC',7,0,'XX')
FROM SYSIBM.SYSDUMMY1
```

- Change the string 'Hegelstraße' to 'Hegelstrasse'.

```
SELECT INSERT('Hegelstraße',10,1,'ss',CODEUNITS16)
FROM SYSIBM.SYSDUMMY1
```

- The following example works with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown below in different Unicode encoding forms:



## INTEGER

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.

### *date-expression*

An expression that returns a value of the DATE data type. The result is an INTEGER value representing the date as *yyyymmdd*.

### *time-expression*

An expression that returns a value of the TIME data type. The result is an INTEGER value representing the time as *hhmmss*.

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
```

- Assume that the column BIRTHDATE (date) has an internal value equivalent to '1964-07-20'.

```
INTEGER(BIRTHDATE)
```

results in the value 19 640 720.

- Assume that the column STARTTIME (time) has an internal value equivalent to '12:03:04'.

```
INTEGER(STARTTIME)
```

results in the value 120 304.

## JULIAN\_DAY

►►—JULIAN\_DAY—(—*expression*—)—————►►

The schema is SYSFUN.

Returns an integer value representing the number of days from January 1, 4713 B.C. (the start of the Julian date calendar) to the date value specified in the argument.

The 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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

## LCASE

►► LCASE (—string-expression—) ◀◀

The schema is SYSIBM.

The LCASE function returns a string in which all the SBCS characters have been converted to lowercase characters.

LCASE is a synonym for LOWER.

## LCASE (Locale sensitive)

►► LCASE (—string-expression—, —locale-name—, —code-units—) ◀◀

CODEUNITS16  
CODEUNITS32  
OCTETS

The schema is SYSIBM.

The LCASE function returns a string in which all characters have been converted to lowercase characters using the rules associated with the specified locale.

LCASE is a synonym for LOWER.

## LEAST

►► LEAST (—expression—, —expression—) ◀◀

The schema is SYSIBM.

The LEAST function returns the minimum value in a set of values.

The arguments must be compatible and each argument must be an expression that returns a value of any data type other than ARRAY, LOB, LONG VARCHAR, LONG VARGRAPHIC, XML, a distinct type based on any of these types, or a structured type (SQLSTATE 42815). This function cannot be used as a source function when creating a user-defined function. Because this function accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by all the operands based on the rules for result data types.

The result of the function is the smallest argument value. If at least one argument can be null, the result can be null; if either argument is null, the result is the null value.





The *string-expression* is padded on the right with the necessary number of padding characters so that the specified substring of *string-expression* always exists. The character used for padding is the same character that is used to pad the string in contexts where padding would occur. For more information on padding, see “String assignments” in “Assignments and comparisons”.

The result of the function is a varying-length string with a length attribute that is the same as the length attribute of *string-expression* and a data type that depends on the data type of *string-expression*:

- VARCHAR if *string-expression* is CHAR or VARCHAR
- CLOB if *string-expression* is CLOB
- VARGRAPHIC if *string-expression* is GRAPHIC or VARGRAPHIC
- DBCLOB if *string-expression* is DBCLOB

The actual length of the result (in string units) is *length*.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples:

- Assume that variable ALPHA has a value of 'ABCDEF'. The following statement:

```
SELECT LEFT(ALPHA,3)
FROM SYSIBM.SYSDUMMY1
```

returns 'ABC', which are the three leftmost characters in ALPHA.

- Assume that variable NAME, which is defined as VARCHAR(50), has a value of 'KATIE AUSTIN', and that the integer variable FIRSTNAME\_LEN has a value of 5. The following statement:

```
SELECT LEFT(NAME,FIRSTNAME_LEN)
FROM SYSIBM.SYSDUMMY1
```

returns the value 'KATIE'.

- The following statement returns a zero-length string.

```
SELECT LEFT('ABCABC',0)
FROM SYSIBM.SYSDUMMY1
```

- The FIRSTNAME column in the EMPLOYEE table is defined as VARCHAR(12). Find the first name of an employee whose last name is 'BROWN' and return the first name in a 10-byte string.

```
SELECT LEFT(FIRSTNAME, 10)
FROM EMPLOYEE
WHERE LASTNAME = 'BROWN'
```

returns a VARCHAR(12) string that has the value 'DAVID' followed by five blank characters.

- In a Unicode database, FIRSTNAME is a VARCHAR(12) column. One of its values is the 6-character string 'Jürgen'. When FIRSTNAME has this value:

| Function...                                | Returns...                         |
|--------------------------------------------|------------------------------------|
| <code>LEFT(FIRSTNAME,2,CODEUNITS32)</code> | 'Jü' -- x'4AC3BC'                  |
| <code>LEFT(FIRSTNAME,2,CODEUNITS16)</code> | 'Jü' -- x'4AC3BC'                  |
| <code>LEFT(FIRSTNAME,2,OCTETS)</code>      | 'J' -- x'4A20', a truncated string |

- The following example works with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown below in different Unicode encoding forms:

## LEFT

|          | '&'         | 'N'     | '~'     | 'A'     | 'B'     |
|----------|-------------|---------|---------|---------|---------|
| UTF-8    | X'F09D849E' | X'4E'   | X'CC83' | X'41'   | X'42'   |
| UTF-16BE | X'D834DD1E' | X'004E' | X'0303' | X'0041' | X'0042' |

Assume that the variable UTF8\_VAR, with a length attribute of 20 bytes, contains the UTF-8 representation of the string.

```
SELECT LEFT(UTF8_VAR, 2, CODEUNITS16),
 LEFT(UTF8_VAR, 2, CODEUNITS32),
 LEFT(UTF8_VAR, 2, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&', '&N', and 'bb', respectively, where 'b' represents the blank character.

```
SELECT LEFT(UTF8_VAR, 5, CODEUNITS16),
 LEFT(UTF8_VAR, 5, CODEUNITS32),
 LEFT(UTF8_VAR, 5, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~A', '&N~AB', and '&N', respectively.

```
SELECT LEFT(UTF8_VAR, 10, CODEUNITS16),
 LEFT(UTF8_VAR, 10, CODEUNITS32),
 LEFT(UTF8_VAR, 10, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~ABbbb', '&N~ABbbbb', and '&N~ABb', respectively, where 'b' represents the blank character.

Assume that the variable UTF16\_VAR, with a length attribute of 20 code units, contains the UTF-16BE representation of the string.

```
SELECT LEFT(UTF16_VAR, 2, CODEUNITS16),
 LEFT(UTF16_VAR, 2, CODEUNITS32),
 HEX (LEFT(UTF16_VAR, 2, OCTETS))
FROM SYSIBM.SYSDUMMY1
```

returns the values '&', '&N', and X'D834', respectively, where X'D834' is an unmatched high surrogate.

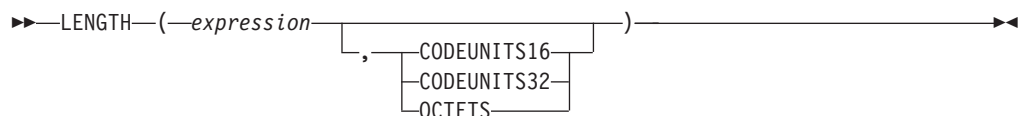
```
SELECT LEFT(UTF16_VAR, 5, CODEUNITS16),
 LEFT(UTF16_VAR, 5, CODEUNITS32),
 LEFT(UTF16_VAR, 6, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~A', '&N~AB', and '&N', respectively.

```
SELECT LEFT(UTF16_VAR, 10, CODEUNITS16),
 LEFT(UTF16_VAR, 10, CODEUNITS32),
 LEFT(UTF16_VAR, 10, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~ABbbb', '&N~ABbbbb', and '&N~A', respectively, where 'b' represents the blank character.

## LENGTH



The schema is SYSIBM.

The LENGTH function returns the length of *expression* in the implicit or explicit string unit.

*expression*

An expression that returns a value that is a built-in data type. If *expression* can be null, the result can be null; if *expression* is null, the result is the null value.

#### CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the string unit of the result. CODEUNITS16 specifies that the result is to be expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that the result is to be expressed in 32-bit UTF-32 code units. OCTETS specifies that the result is to be expressed in bytes.

If a string unit is explicitly specified, and if *expression* is not string data, an error is returned (SQLSTATE 428GC). If a string unit is specified as CODEUNITS16 or CODEUNITS32, and *expression* is a binary string or bit data, an error is returned (SQLSTATE 428GC). If a string unit is specified as OCTETS and *expression* is a binary string, an error is returned (SQLSTATE 42815). For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

If a string unit is not explicitly specified, the data type of the result determines the unit that is used. If the result is graphic data, the value returned specifies the length in 2-byte units. Otherwise, the value returned specifies the length in bytes.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The length of character and graphic strings includes trailing blanks. The length of binary strings includes binary zeroes. The length of varying-length strings is the actual length and not the maximum length. The length of all other values is the number of bytes used to represent the value:

- 2 for small integer
- 4 for large integer
- $(p/2)+1$  for decimal numbers with precision  $p$
- 8 for DECFLOAT(16)
- 16 for DECFLOAT(34)
- The length of the string for binary strings
- The length of the string for character strings
- 4 for single-precision floating-point
- 8 for double-precision floating-point
- 4 for date
- 3 for time
- 10 for timestamp

Examples:

- Assume that the host variable ADDRESS is a varying-length character string with a value of '895 Don Mills Road'.

```
LENGTH(:ADDRESS)
```

returns the value 18.

- Assume that START\_DATE is a column of type DATE.

## LENGTH

```
LENGTH(START_DATE)
```

returns the value 4.

•

```
LENGTH(CHAR(START_DATE, EUR))
```

returns the value 10.

- The following examples work with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown below in different Unicode encoding forms:

|          | '&'         | 'N'         | '~'         | 'A'         | 'B'         |
|----------|-------------|-------------|-------------|-------------|-------------|
| UTF-8    | X'F09D849E' | X'4E'       | X'CC83'     | X'41'       | X'42'       |
| UTF-16BE | X'D834DD1E' | X'004E'     | X'0303'     | X'0041'     | X'0042'     |
| UTF-32BE | X'0001D11E' | X'0000004E' | X'00000303' | X'00000041' | X'00000042' |

Assume that the variable UTF8\_VAR contains the UTF-8 representation of the string.

```
SELECT LENGTH(UTF8_VAR, CODEUNITS16),
 LENGTH(UTF8_VAR, CODEUNITS32),
 LENGTH(UTF8_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 6, 5, and 9, respectively.

Assume that the variable UTF16\_VAR contains the UTF-16BE representation of the string.

```
SELECT LENGTH(UTF16_VAR, CODEUNITS16),
 LENGTH(UTF16_VAR, CODEUNITS32),
 LENGTH(UTF16_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 6, 5, and 12, respectively.

## LN

►►—LN—(—expression—)—————►►

The schema is SYSIBM. (The SYSFUN version of the LN function continues to be available.)

The LN function returns the natural logarithm of a number. The LN and EXP functions are inverse operations.

The argument must be an expression that returns a value of any built-in numeric data type. If the argument is decimal floating-point, the operation is performed in decimal floating-point; otherwise, the argument is converted to double-precision floating-point for processing by the function. The value of the argument must be greater than zero (SQLSTATE 22003).

If the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*); otherwise, the result is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

**Syntax alternatives:** LOG can be specified in place of LN. It is supported only for compatibility with previous versions of DB2. LN should be used instead of LOG, because some database managers and applications implement LOG as the common logarithm of a number instead of the natural logarithm of a number.

Example:

- Assume that NATLOG is a DECIMAL(4,2) host variable with a value of 31.62.

```
VALUES LN(:NATLOG)
```

Returns the approximate value 3.45.

## LOCATE scalar function

```

▶▶—LOCATE—(—search-string—,—source-string—,—start—,—CODEUNITS16—,—CODEUNITS32—,—OCTETS—)

```

The schema is SYSIBM. The SYSFUN version of the LOCATE function continues to be available, but it is not sensitive to the database collation.

The LOCATE function returns the starting position of the first occurrence of one string (called the *search-string*) within another string (called the *source-string*). If the *search-string* is not found and neither argument is null, the result is zero. If the *search-string* is found, the result is a number from 1 to the actual length of the *source-string*. The search is done using the collation of the database, unless *search-string* or *source-string* is defined as FOR BIT DATA, in which case the search is done using a binary comparison.

If the optional *start* is specified, it indicates the character position in the *source-string* at which the search is to begin. An optional string unit can be specified to indicate in what units the *start* and result of the function are expressed.

If the *search-string* has a length of zero, the result returned by the function is 1. Otherwise, if the *source-string* has a length of zero, the result returned by the function is 0. Otherwise:

- If the value of *search-string* is equal to an identical length of substring of contiguous positions within the value of *source-string*, the result returned by the function is the starting position of the first such substring within the *source-string* value.
- Otherwise, the result returned by the function is 0.

*search-string*

An expression that specifies the string that is the object of the search. The expression must return a value that is a built-in character string data type, graphic string data type, or binary string data type with an actual length that is no greater than 4000 bytes. No element of the expression can be of type LONG VARCHAR, CLOB, LONG VARGRAPHIC, or DBCLOB. In addition, it cannot be a BLOB file reference variable. The expression can be specified by any of the following:

- A constant
- A special register
- A global variable
- A host variable

## LOCATE scalar function

- A scalar function whose operands are any of the above
- An expression that concatenates (using CONCAT or ||) any of the above
- An SQL procedure parameter

These rules are similar to those that are described for *pattern-expression* for the LIKE predicate.

### *source-string*

An expression that specifies the string in which the search is to take place. The expression must return a value that is a built-in character string data type, graphic string data type, or binary string data type. The expression can be specified by any of the following:

- A constant
- A special register
- A global variable
- A host variable (including a locator variable or a file reference variable)
- A scalar function
- A large object locator
- A column name
- An expression that concatenates (using CONCAT or ||) any of the above

### *start*

An expression that specifies the position within *source-string* at which the search is to start. The expression must be an integer that is greater than or equal to zero. If *start* is specified, the LOCATE function is similar to:

```
POSITION(search-string,
SUBSTRING(source-string, start, string-unit),
string-unit) + start - 1
```

where *string-unit* is either CODEUNITS16, CODEUNITS32, or OCTETS.

If *start* is not specified, the search begins at the first position of the source string, and the LOCATE function is similar to:

```
POSITION(search-string, source-string, string-unit)
```

### **CODEUNITS16, CODEUNITS32, or OCTETS**

Specifies the string unit of *start* and the result. CODEUNITS16 specifies that *start* and the result are to be expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *start* and the result are to be expressed in 32-bit UTF-32 code units. OCTETS specifies that *start* and the result are to be expressed in bytes.

If a string unit is specified as CODEUNITS16 or CODEUNITS32, and *search-string* or *source-string* is a binary string or bit data, an error is returned (SQLSTATE 428GC). If a string unit is specified as OCTETS and *search-string* and *source-string* are binary strings, an error is returned (SQLSTATE 42815).

If a string unit is not explicitly specified, the data type of the result determines the unit that is used. If the result is graphic data, *start* and the returned position are expressed in two-byte units; otherwise, they are expressed in bytes.

If a locale-sensitive UCA-based collation is used for this function, then the CODEUNITS16 option offers the best performance characteristics.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

The first and second arguments must have compatible string types. For more information on compatibility, see “Rules for string conversions”. In a Unicode database, if one string argument is character (not FOR BIT DATA) and the other string argument is graphic, then the *search-string* is converted to the data type of the *source-string* for processing. If one argument is character FOR BIT DATA, the other argument must not be graphic (SQLSTATE 42846).

The result of the function is a large integer. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples:

- Find the location of the first occurrence of the character 'N' in the string 'DINING'.

```
SELECT LOCATE('N', 'DINING')
FROM SYSIBM.SYSDUMMY1
```

The result is the value 3.

- For all the rows in the table named IN\_TRAY, select the RECEIVED column, the SUBJECT column, and the starting position of the string 'GOOD' within the NOTE\_TEXT column.

```
SELECT RECEIVED, SUBJECT, LOCATE('GOOD', NOTE_TEXT)
FROM IN_TRAY
WHERE LOCATE('GOOD', NOTE_TEXT) <> 0
```

- Locate the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable LOCATION with the position, as measured in CODEUNITS32 units, within the string.

```
SET :LOCATION = LOCATE('ß', 'Jürgen lives on Hegelstraße', 1, CODEUNITS32)
```

The value of host variable LOCATION is set to 26.

- Locate the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable LOCATION with the position, as measured in CODEUNITS16 units, within the string.

```
SET :LOCATION = LOCATE('ß', 'Jürgen lives on Hegelstraße', 1, CODEUNITS16)
```

The value of host variable LOCATION is set to 26.

- Locate the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable LOCATION with the position, as measured in OCTETS, within the string.

```
SET :LOCATION = LOCATE('ß', 'Jürgen lives on Hegelstraße', 1, OCTETS)
```

The value of host variable LOCATION is set to 27.

- The following examples work with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the non-spacing combining tilde character. This string is shown below in different Unicode encoding forms:

|          | '&'         | 'N'     | '~'     | 'A'     | 'B'     |
|----------|-------------|---------|---------|---------|---------|
| UTF-8    | X'F09D849E' | X'4E'   | X'CC83' | X'41'   | X'42'   |
| UTF-16BE | X'D834DD1E' | X'004E' | X'0303' | X'0041' | X'0042' |

Assume that the variable UTF8\_VAR contains the UTF-8 representation of the string.

## LOCATE scalar function

```
SELECT LOCATE('~', UTF8_VAR, CODEUNITS16),
 LOCATE('~', UTF8_VAR, CODEUNITS32),
 LOCATE('~', UTF8_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 4, 3, and 6, respectively.

Assume that the variable UTF16\_VAR contains the UTF-16BE representation of the string.

```
SELECT LOCATE('~', UTF16_VAR, CODEUNITS16),
 LOCATE('~', UTF16_VAR, CODEUNITS32),
 LOCATE('~', UTF16_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 4, 3, and 7, respectively.

- In a Unicode database created with the case insensitive collation UCA500R1\_LEN\_S1, find the position of the word 'Brown' in the phrase 'The quick brown fox'.

```
SET :LOCATION = LOCATE('Brown', 'The quick brown fox', CODEUNITS16)
```

The value of the host variable LOCATION is set to 11.

## LOG10

►►—LOG10—(—*expression*—)—————►►

The schema is SYSIBM. (The SYSFUN version of the LOG10 function continues to be available.)

The LOG10 function returns the common logarithm (base 10) of a number.

The argument must be an expression that returns a value of any built-in numeric data type. If the argument is decimal floating-point, the operation is performed in decimal floating-point; otherwise, the argument is converted to double-precision floating-point for processing by the function. The value of the argument must be greater than zero (SQLSTATE 22003).

If the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*); otherwise, the result is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example:

- Assume that L is a DECIMAL(4,2) host variable with a value of 31.62.

```
VALUES LOG10(:L)
```

Returns the DOUBLE value +1.49996186559619E+000.

## LONG\_VARCHAR

►►—LONG\_VARCHAR—(—*character-string-expression*—)—————►►

The LONG\_VARCHAR function is deprecated and might be removed in a future release. The function is compatible with earlier DB2 versions.



## LONG\_VARGRAPHIC

▶▶—LONG\_VARGRAPHIC—(—*graphic-expression*—)————▶▶

The LONG\_VARGRAPHIC function is deprecated and might be removed in a future release. The function is compatible with earlier DB2 versions.

## LOWER

▶▶—LOWER—(—*string-expression*—)————▶▶

The schema is SYSIBM. (The SYSFUN version of this function continues to be available with support for LONG VARCHAR and CLOB arguments.)

The 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 converted to the characters a-z, and other characters will be converted to their lowercase equivalents, if they exist. For example, in code page 850, É maps to é. If the code point length of the result character is not the same as the code point length of the source character, the source character is not converted. Because not all characters are converted, LOWER(UPPER(*string-expression*)) does not necessarily return the same result as LOWER(*string-expression*).

The 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 as the argument. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

Example:

Ensure that the characters in the value of column JOB in the EMPLOYEE table are returned in lowercase characters.

```
SELECT LOWER(JOB)
FROM EMPLOYEE
WHERE EMPNO = '000020';
```

The result is the value 'manager'.

## LOWER (Locale sensitive)

▶▶—LOWER—(—*string-expression*—, —*locale-name*—)————▶▶

└─, —*code-units*—┘ └─, ┌─CODEUNITS16─┐

└─CODEUNITS32─┘

└─OCTETS─┘

The schema is SYSIBM.

## LOWER (Locale sensitive)

The LOWER function returns a string in which all characters have been converted to lowercase characters using the rules associated with the specified locale.

### *string-expression*

An expression that returns a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC string. If *string-expression* is CHAR or VARCHAR, the expression must not be FOR BIT DATA (SQLSTATE 42815).

### *locale-name*

A character constant that specifies the locale that defines the rules for conversion to lowercase characters. The value of *locale-name* is not case sensitive and must be a valid locale (SQLSTATE 42815). For information on valid locales and their naming, see “Locale names for SQL and XQuery”.

### *code-units*

An integer constant that specifies the number of code units in the result. If specified, *code-units* must be an integer between 1 and 32 672 if the result is character data, or between 1 and 16 336 if the result is graphic data (SQLSTATE 42815). If *code-units* is not explicitly specified, it is implicitly the length attribute of *string-expression*. If OCTETS is specified and the result is graphic data, the value of *code-units* must be even (SQLSTATE 428GC).

### CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the string unit of *code-units*.

CODEUNITS16 specifies that *code-units* is expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *code-units* is expressed in 32-bit UTF-32 code units. OCTETS specifies that *code-units* is expressed in bytes.

If a string unit is not explicitly specified, the data type of the result determines the unit that is used. If the result is graphic data, *code-units* is expressed in two-byte units; otherwise, it is expressed in bytes. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

The result of the function is VARCHAR if *string-expression* is CHAR or VARCHAR, and VARGRAPHIC if *string-expression* is GRAPHIC or VARGRAPHIC.

The length attribute of the result is determined by the implicit or explicit value of *code-units*, the implicit or explicit string unit, and the result data type, as shown in the following table:

Table 34. Length attribute of the result of LOWER as a function of string unit and result type

| String unit | Character result type                      | Graphic result type                        |
|-------------|--------------------------------------------|--------------------------------------------|
| CODEUNITS16 | $\text{MIN}(\text{code-units} * 3, 32672)$ | <i>code-units</i>                          |
| CODEUNITS32 | $\text{MIN}(\text{code-units} * 4, 32672)$ | $\text{MIN}(\text{code-units} * 2, 16336)$ |
| OCTETS      | <i>code-units</i>                          | $\text{MIN}(\text{code-units} / 2, 16336)$ |

The actual length of the result might be greater than the length of *string-expression*. If the actual length of the result is greater than the length attribute of the result, an error is returned (SQLSTATE 42815). If the number of code units in the result exceeds the value of *code-units*, an error is returned (SQLSTATE 42815).

If *string-expression* is not in UTF-16, this function performs code page conversion of *string-expression* to UTF-16, and of the result from UTF-16 to the code page of *string-expression*. If either code page conversion results in at least one substitution

character, the result includes the substitution character, a warning is returned (SQLSTATE 01517), and the warning flag SQLWARN8 in the SQLCA is set to 'W'.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples:

- Ensure that the characters in the value of column JOB in the EMPLOYEE table are returned in lowercase characters.

```
SELECT LOWER(JOB, 'en_US')
FROM EMPLOYEE
WHERE EMPNO = '000020'
```

The result is the value 'manager'.

- Find the lowercase characters for all the 'İ' characters in a Turkish string.

```
VALUES LOWER('İİİİ', 'tr_TR', CODEUNITS16)
```

The result is the string 'iii'.

## LTRIM

►►—LTRIM—(—*string-expression*—)—————►►

The schema is SYSIBM. (The SYSFUN version of this function continues to be available with support for LONG VARCHAR and CLOB arguments.)

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.

## LTRIM

Example:

Assume that host variable HELLO is defined as CHAR(9) and has a value of 'Hello'.

```
VALUES LTRIM(:HELLO)
```

The result is 'Hello'.

## MAX

```
▶▶ MAX(-expression, -expression) ▶▶
```

The schema is SYSIBM.

The MAX function returns the maximum value in a set of values.

The arguments must be compatible and each argument must be an expression that returns a value of any data type other than ARRAY, LOB, LONG VARCHAR, LONG VARGRAPHIC, XML, a distinct type based on any of these types, or a structured type (SQLSTATE 42815). This function cannot be used as a source function when creating a user-defined function. Because this function accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by all the operands based on the rules for result data types.

The result of the function is the largest argument value. If at least one argument can be null, the result can be null; if either argument is null, the result is the null value.

The MAX scalar function is a synonym for the GREATEST scalar function.

Example:

Return the bonus for an employee, the greater of 500 and 5% of the employee's salary.

```
SELECT EMPNO, MAX(SALARY * 0.05, 500)
FROM EMPLOYEE
```

## MAX\_CARDINALITY

```
▶▶ MAX_CARDINALITY(-array-expression) ▶▶
```

The schema is SYSIBM.

The MAX\_CARDINALITY function returns a value of type BIGINT representing the maximum number of elements that an array can contain. This is the cardinality that was specified in the CREATE TYPE statement for the user-defined array type.

The argument can be either an SQL procedure variable or parameter of an array data type or a cast specification of a parameter marker to an array data type.

Example:

Assume that type PHONE\_LIST is defined as:

```
CREATE TYPE PHONE_LIST
AS INTEGER ARRAY[100]
```

The array NUMBERS is of type PHONE\_LIST. The following SET statement assigns variable CARD the value 100, in accordance with the definition of PHONE\_LIST:

```
SET CARD = MAX_CARDINALITY(NUMBERS)
```

## 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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- 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.

## MIDNIGHT\_SECONDS

Returns an integer value in the range 0 to 86 400, representing the number of seconds between midnight and the time value specified in the argument.

The 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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

Examples:

- Find the number of seconds between midnight and 00:10:10, and midnight and 13:10:10.

```
VALUES (MIDNIGHT_SECONDS('00:10:10'), MIDNIGHT_SECONDS('13:10:10'))
```

This example returns the following:

```
1 2

 610 47410
```

Since a minute is 60 seconds, there are 610 seconds between midnight and the specified time. The same follows for the second example. There are 3600 seconds in an hour, and 60 seconds in a minute, resulting in 47 410 seconds between the specified time and midnight.

- Find the number of seconds between midnight and 24:00:00, and midnight and 00:00:00.

```
VALUES (MIDNIGHT_SECONDS('24:00:00'), MIDNIGHT_SECONDS('00:00:00'))
```

This example returns the following:

```
1 2

 86400 0
```

Note that these two values represent the same point in time, but return different MIDNIGHT\_SECONDS values.

## MIN

A diagram showing the syntax of the MIN function. It consists of the text 'MIN' followed by an opening parenthesis '(', then two placeholders for expressions: '-expression' and ', -expression', followed by a closing parenthesis ')'. A horizontal arrow points from the opening parenthesis to the right. A vertical line descends from the first '-expression', and another vertical line descends from the second '-expression'. These two vertical lines are connected by a horizontal line at the top, forming a rectangular box that encloses the two expressions.

The schema is SYSIBM.

The MIN function returns the minimum value in a set of values.

The arguments must be compatible and each argument must be an expression that returns a value of any data type other than ARRAY, LOB, LONG VARCHAR, LONG VARGRAPHIC, XML, a distinct type based on any of these types, or a structured type (SQLSTATE 42815). This function cannot be used as a source function when creating a user-defined function. Because this function accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by all the operands based on the rules for result data types.

The result of the function is the smallest argument value. If at least one argument can be null, the result can be null; if either argument is null, the result is the null value.

The MIN scalar function is a synonym for the LEAST scalar function.

Example:

Return the bonus for an employee, the LESSER of 5000 and 5% of the employee's salary.

```
SELECT EMPNO, MIN(SALARY * 0.05, 5000)
FROM EMPLOYEE
```

## MINUTE

►►—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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- 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** (*—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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- 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—(—*expression*—)—————►◄

The schema is SYSFUN.

Returns a mixed case character string containing the name of the month (for example, 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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is VARCHAR(100). The result can be null; if the argument is null, the result is the null value.

## MULTIPLY\_ALT

►►—MULTIPLY\_ALT—(—*exact\_numeric\_expression*—,—*exact\_numeric\_expression*—)—————►◄

The schema is SYSIBM.

The MULTIPLY\_ALT scalar function returns the product of the two arguments as a decimal value. It is provided as an alternative to the multiplication operator, especially when the sum of the precisions of the arguments exceeds 31.

The arguments can be any built-in exact numeric data type (DECIMAL, BIGINT, INTEGER, or SMALLINT).

The result of the function is a DECIMAL. The precision and scale of the result are determined as follows, using the symbols  $p$  and  $s$  to denote the precision and scale of the first argument, and the symbols  $p'$  and  $s'$  to denote the precision and scale of the second argument.

- The precision is  $\text{MIN}(31, p + p')$
- The scale is:
  - 0 if the scale of both arguments is 0
  - $\text{MIN}(31, s + s')$  if  $p + p'$  is less than or equal to 31
  - $\text{MAX}(\text{MIN}(3, s + s'), 31 - (p - s + p' - s'))$  if  $p + p'$  is greater than 31.

The result can be null if at least one argument can be null, or if the database is configured with DFT\_SQLMATHWARN set to YES; the result is the null value if one of the arguments is null.

The MULTIPLY\_ALT function is a preferable choice to the multiplication operator when performing decimal arithmetic where a scale of at least 3 is required and the sum of the precisions exceeds 31. In these cases, the internal computation is performed so that overflows are avoided. The final result is then assigned to the result data type, using truncation where necessary to match the scale. Note that overflow of the final result is still possible when the scale is 3.

## MULTIPLY\_ALT

The following is a sample comparing the result types using MULTIPLY\_ALT and the multiplication operator.

| Type of argument 1 | Type of argument 2 | Result using MULTIPLY_ALT | Result using multiplication operator |
|--------------------|--------------------|---------------------------|--------------------------------------|
| DECIMAL(31,3)      | DECIMAL(15,8)      | DECIMAL(31,3)             | DECIMAL(31,11)                       |
| DECIMAL(26,23)     | DECIMAL(10,1)      | DECIMAL(31,19)            | DECIMAL(31,24)                       |
| DECIMAL(18,17)     | DECIMAL(20,19)     | DECIMAL(31,29)            | DECIMAL(31,31)                       |
| DECIMAL(16,3)      | DECIMAL(17,8)      | DECIMAL(31,9)             | DECIMAL(31,11)                       |
| DECIMAL(26,5)      | DECIMAL(11,0)      | DECIMAL(31,3)             | DECIMAL(31,5)                        |
| DECIMAL(21,1)      | DECIMAL(15,1)      | DECIMAL(31,2)             | DECIMAL(31,2)                        |

Example:

Multiply two values where the data type of the first argument is DECIMAL(26,3) and the data type of the second argument is DECIMAL(9,8). The data type of the result is DECIMAL(31,7).

```
values multiply_alt(98765432109876543210987.654,5.43210987)
```

```
1
```

```

536504678578875294857887.5277415
```

Note that the complete product of these two numbers is 536504678578875294857887.52774154498, but the last 4 digits are truncated to match the scale of the result data type. Using the multiplication operator with the same values will cause an arithmetic overflow, since the result data type is DECIMAL(31,11) and the result value has 24 digits left of the decimal, but the result data type only supports 20 digits.

## NORMALIZE\_DECFLOAT

►►—NORMALIZE\_DECFLOAT—(—*expression*—)—————►►

The schema is SYSIBM.

The NORMALIZE\_DECFLOAT function returns a decimal floating-point value equal to the input argument set to its simplest form; that is, a non-zero number with trailing zeros in the coefficient has those zeros removed. This may require representing the number in normalized form by dividing the coefficient by the appropriate power of ten and adjusting the exponent accordingly. A zero value has its exponent set to 0.

*expression*

An expression that returns a value of any built-in numeric data type. Arguments of type SMALLINT, INTEGER, REAL, DOUBLE, or DECIMAL(*p,s*), where *p* ≤ 16, are converted to DECFLOAT(16) for processing. Arguments of type BIGINT or DECIMAL(*p,s*), where *p* > 16, are converted to DECFLOAT(34) for processing.

The result of the function is a DECFLOAT(16) value if the data type of expression after conversion to decimal floating-point is DECFLOAT(16). Otherwise, the result of the function is a DECFLOAT(34) value. If the argument is a special decimal

floating-point value, the result is the same special decimal floating-point value. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples:

- The following examples show the values that are returned by the NORMALIZE\_DECFLOAT function, given a variety of input decimal floating-point values:

```

NORMALIZE_DECFLOAT(DECFLOAT(2.1)) = 2.1
NORMALIZE_DECFLOAT(DECFLOAT(-2.0)) = -2
NORMALIZE_DECFLOAT(DECFLOAT(1.200)) = 1.2
NORMALIZE_DECFLOAT(DECFLOAT(-120)) = -1.2E+2
NORMALIZE_DECFLOAT(DECFLOAT(120.00)) = 1.2E+2
NORMALIZE_DECFLOAT(DECFLOAT(0.00)) = 0
NORMALIZE_DECFLOAT(-NAN) = -NaN
NORMALIZE_DECFLOAT(-INFINITY) = -Infinity

```

## 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. They can be of either a built-in (other than a long string) or distinct data type (other than based on a long string). (This function cannot be used as a source function when creating a user-defined function. Because this function accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.) 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.

## NVL

►► **NVL** (—*expression*—, —*expression*—) ◀◀

The schema is SYSIBM.

The NVL function returns the first argument that is not null.

NVL is a synonym for COALESCE.

## OCTET\_LENGTH

►►—OCTET\_LENGTH—(—*expression*—)—————►

The schema is SYSIBM.

The OCTET\_LENGTH function returns the length of *expression* in octets (bytes).

*expression*

An expression that returns a value that is a built-in string data type.

The result of the function is INTEGER. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The length of character or graphic strings includes trailing blanks. The length of binary strings includes binary zeroes. The length of varying-length strings is the actual length and not the maximum length.

For greater portability, code your application to be able to accept a result of data type DECIMAL(31).

Examples:

- Assume that table T1 has a GRAPHIC(10) column named C1.

```
SELECT OCTET_LENGTH(C1) FROM T1
```

returns the value 20.

- The following example works with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown below in different Unicode encoding forms:

|          | '&'         | 'N'     | '~'     | 'A'     | 'B'     |
|----------|-------------|---------|---------|---------|---------|
| UTF-8    | X'F09D849E' | X'4E'   | X'CC83' | X'41'   | X'42'   |
| UTF-16BE | X'D834DD1E' | X'004E' | X'0303' | X'0041' | X'0042' |

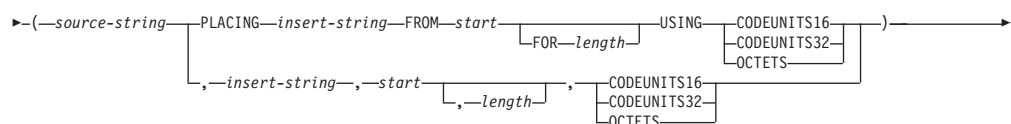
Assume that the variables UTF8\_VAR and UTF16\_VAR contain the UTF-8 and the UTF-16BE representations of the string, respectively.

```
SELECT OCTET_LENGTH(UTF8_VAR),
 OCTET_LENGTH(UTF16_VAR)
FROM SYSIBM.SYSDUMMY1
```

returns the values 9 and 12, respectively.

## OVERLAY

►►—OVERLAY—————►



The schema is SYSIBM.

The OVERLAY function returns a string in which, beginning at *start* in *source-string*, *length* of the specified code units have been deleted and *insert-string* has been inserted.

#### *source-string*

An expression that specifies the source string. The expression must return a value that is a built-in character or graphic string data type (SQLSTATE 42815).

#### *insert-string*

An expression that specifies the string to be inserted into *source-string*, starting at the position identified by *start*. The expression must return a value that is a built-in character or graphic string data type compatible with *source-string* (SQLSTATE 22011). If the code page of the *insert-string* differs from that of the *source-string*, *insert-string* is converted to the code page of the *source-string*.

#### *start*

An expression that returns an integer value. The integer value specifies the starting point within the source string where the deletion of bytes and the insertion of another string is to begin. The integer value must be between 1 and the length of *source-string* plus one (SQLSTATE 42815). If OCTETS is specified and the result is graphic data, the value must be an odd number between 1 and twice the length attribute of *source-string* plus one (SQLSTATE 428GC).

#### *length*

An expression that specifies the number of code units (in the specified string units) that are to be deleted from the source string, starting at the position identified by *start*. The value must be between 0 and the length of *source-string*, expressed in units that are either implicitly or explicitly specified (SQLSTATE 22011). If OCTETS is specified and the result is graphic data, the value must be an even number between 0 and twice the length attribute of *source-string* (SQLSTATE 428GC).

Not specifying *length* is equivalent to specifying a value of 1, except when OCTETS is specified and the result is graphic data, in which case, not specifying *length* is equivalent to specifying a value of 2.

### **CODEUNITS16, CODEUNITS32, or OCTETS**

Specifies the string unit of *start* and *length*.

CODEUNITS16 specifies that *start* and *length* are expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *start* and *length* are expressed in 32-bit UTF-32 code units. OCTETS specifies that *start* and *length* are expressed in bytes.

If the string unit is specified as CODEUNITS16 or CODEUNITS32, and the result is a binary string or bit data, an error is returned (SQLSTATE 428GC). If the string unit is specified as OCTETS, and *insert-string* and *source-string* are binary strings, an error is returned (SQLSTATE 42815). If the string unit is specified as OCTETS, the operation is performed in the code page of the *source-string*. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

## OVERLAY

The data type of the result depends on the data types of *source-string* and *insert-string*, as shown in the following table of supported type combinations.

*Table 35. Data type of the result as a function of the data types of source-string and insert-string*

| <i>source-string</i>                      | <i>insert-string</i>                                      | <b>Result</b>        |
|-------------------------------------------|-----------------------------------------------------------|----------------------|
| CHAR or VARCHAR                           | CHAR or VARCHAR                                           | VARCHAR              |
| GRAPHIC or VARGRAPHIC                     | GRAPHIC or VARGRAPHIC                                     | VARGRAPHIC           |
| CLOB                                      | CHAR, VARCHAR, or CLOB                                    | CLOB                 |
| DBCLOB                                    | GRAPHIC, VARGRAPHIC, or DBCLOB                            | DBCLOB               |
| CHAR or VARCHAR                           | CHAR FOR BIT DATA or VARCHAR FOR BIT DATA                 | VARCHAR FOR BIT DATA |
| CHAR FOR BIT DATA or VARCHAR FOR BIT DATA | CHAR, VARCHAR, CHAR FOR BIT DATA, or VARCHAR FOR BIT DATA | VARCHAR FOR BIT DATA |
| <b>For Unicode databases only:</b>        |                                                           |                      |
| CHAR or VARCHAR                           | GRAPHIC or VARGRAPHIC                                     | VARCHAR              |
| GRAPHIC or VARGRAPHIC                     | CHAR or VARCHAR                                           | VARGRAPHIC           |
| CLOB                                      | GRAPHIC, VARGRAPHIC, or DBCLOB                            | CLOB                 |
| DBCLOB                                    | CHAR, VARCHAR, or CLOB                                    | DBCLOB               |

A *source-string* can have a length of 0; in this case, *start* must be 1 and *length* must be 0 (as implied by the bounds for *start* and *length* described above), and the result of the function is a copy of the *insert-string*. If *length* is not explicitly specified in this case, an error is returned because the assumed length is non-zero (SQLSTATE 22011).

An *insert-string* can also have a length of 0. This has the effect of deleting the code units from positions *start* to *start* + *length* - 1 from the *source-string*.

The length attribute of the result is the length attribute of *source-string* plus the length attribute of *insert-string*. The actual length of the result is  $A1 - \text{MIN}((A1 - V2 + 1), V3) + A4$ , where:

- A1 is the actual length of *source-string*
- V2 is the value of *start*
- V3 is the value of *length*
- A4 is the actual length of *insert-string*

If the actual length of the result string exceeds the maximum for the return data type, an error is returned (SQLSTATE 54006).

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples:

- Create the strings 'INSISTING', 'INSISERTING', and 'INSTING' from the string 'INSERTING' by inserting text into the middle of the existing text.

```
SELECT OVERLAY('INSERTING','IS',4,2,OCTETS),
OVERLAY('INSERTING','IS',4,0,OCTETS),
OVERLAY('INSERTING','',4,2,OCTETS)
FROM SYSIBM.SYSDUMMY1
```

- Create the strings 'XXINSERTING', 'XXNSERTING', 'XXSERTING', and 'XXERTING' from the string 'INSERTING' by inserting text before the existing text, using 1 as the starting point.

```
SELECT OVERLAY('INSERTING','XX',1,0,CODEUNITS16)),
OVERLAY('INSERTING','XX',1,1,CODEUNITS16)),
OVERLAY('INSERTING','XX',1,2,CODEUNITS16)),
OVERLAY('INSERTING','XX',1,3,CODEUNITS16))
FROM SYSIBM.SYSDUMMY1
```

- Create the string 'ABCABCXX' from the string 'ABCABC' by inserting text after the existing text. Because the source string is 6 characters long, set the starting position to 7 (one plus the length of the source string).

```
SELECT OVERLAY('ABCABC','XX',7,0,CODEUNITS16))
FROM SYSIBM.SYSDUMMY1
```

- Change the string 'Hegelstraße' to 'Hegelstrasse'.

```
SELECT OVERLAY('Hegelstraße','ss',10,1,CODEUNITS16))
FROM SYSIBM.SYSDUMMY1
```

- The following example works with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown below in different Unicode encoding forms:

|          | '&'         | 'N'     | '~'     | 'A'     | 'B'     |
|----------|-------------|---------|---------|---------|---------|
| UTF-8    | X'F09D849E' | X'4E'   | X'CC83' | X'41'   | X'42'   |
| UTF-16BE | X'D834DD1E' | X'004E' | X'0303' | X'0041' | X'0042' |

Assume that the variables UTF8\_VAR and UTF16\_VAR contain the UTF-8 and the UTF-16BE representations of the string, respectively. Use the OVERLAY function to insert a 'C' into the Unicode string '&N~AB'.

```
SELECT OVERLAY(UTF8_VAR, 'C', 1, CODEUNITS16),
OVERLAY(UTF8_VAR, 'C', 1, CODEUNITS32),
OVERLAY(UTF8_VAR, 'C', 1, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 'C?N~AB', 'CN~AB', and 'CbbbN~AB', respectively, where '?' represents X'EDB49E', which corresponds to the X'DD1E' in the intermediate UTF-16 form, and 'bbb' replaces the UTF-8 incomplete characters X'9D849E'.

```
SELECT OVERLAY(UTF8_VAR, 'C', 5, CODEUNITS16),
OVERLAY(UTF8_VAR, 'C', 5, CODEUNITS32),
OVERLAY(UTF8_VAR, 'C', 5, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~CB', '&N~AC', and '&N~AB', respectively.

```
SELECT OVERLAY(UTF16_VAR, 'C', 1, CODEUNITS16),
OVERLAY(UTF16_VAR, 'C', 1, CODEUNITS32)
FROM SYSIBM.SYSDUMMY1
```

returns the values 'C?N~AB' and 'CN~AB', respectively, where '?' represents the unmatched low surrogate U+DD1E.

```
SELECT OVERLAY(UTF16_VAR, 'C', 5, CODEUNITS16),
OVERLAY(UTF16_VAR, 'C', 5, CODEUNITS32)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~CB' and '&N~AC', respectively.

## PARAMETER

### PARAMETER

The PARAMETER function represents a position in an SQL statement where the value is provided dynamically by XQuery as part of the invocation of the db2-fn:sqlquery function.

►►—PARAMETER—(*integer-constant*)—►►

The schema is SYSIBM.

The *integer-constant* is a position index to a value in the arguments of db2-fn:sqlquery. The value must be between 1 and the total number of the arguments specified in the db2-fn:sqlquery SQL statement (SQLSTATE 42815).

The PARAMETER function represents a position in an SQL statement where the value is provided dynamically by XQuery as part of the invocation of the db2-fn:sqlquery function. The argument of the PARAMETER function determines which value is substituted for the PARAMETER function when the db2-fn:sqlquery function is executed. The value supplied by the PARAMETER function can be referenced multiple times within the same SQL statement.

This function can only be used in a fullselect contained in the string literal argument of the db2-fn:sqlquery function in an XQuery expression (SQLSTATE 42887).

Example:

In the following example, the db2-fn:sqlquery function call uses one PARAMETER function call and the XQuery expression \$po/@OrderDate, the order date attribute. The PARAMETER function call returns the value of order date attribute:

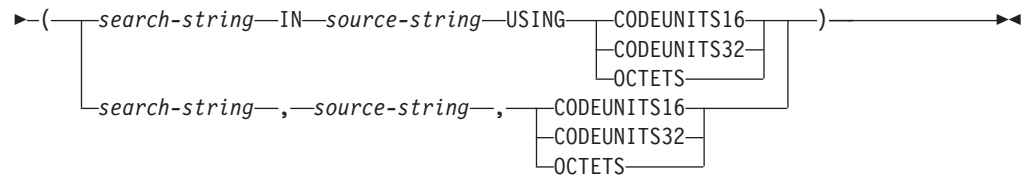
```
xquery
declare default element namespace "http://posample.org";
for $po in db2-fn:xmlcolumn('PURCHASEORDER.PORDER')/PurchaseOrder,
 $item in $po/item/partid
for $p in db2-fn:sqlquery(
 "select description from product where promostart < PARAMETER(1)",
 $po/@OrderDate)
where $p//@pid = $item
return
<RESULT>
 <PoNum>{data($po/@PoNum)}</PoNum>
 <PartID>{data($item)} </PartID>
 <PoDate>{data($po/@OrderDate)}</PoDate>
</RESULT>
```

The example returns the purchase ID, part ID, and the purchase date for all the parts sold after the promotional start date.

### POSITION scalar function

►►—POSITION—►►





The schema is SYSIBM.

The POSITION function returns the starting position of the first occurrence of one string (called the *search-string*) within another string (called the *source-string*). If *search-string* is not found and neither argument is null, the result is zero. If the *search-string* is found, the result is a number from 1 to the actual length of *source-string*, expressed in the string unit that is explicitly specified. The search is done using the collation of the database, unless *search-string* or *source-string* is defined as FOR BIT DATA, in which case the search is done using a binary comparison.

If *source-string* has an actual length of 0, the result of the function is 0. If *search-string* has an actual length of 0 and *source-string* is not null, the result of the function is 1.

*search-string*

An expression that specifies the string that is the object of the search. The expression must return a value that is a built-in character string data type, graphic string data type, or binary string data type with an actual length that is no greater than 4000 bytes. No element of the expression can be of type LONG VARCHAR, CLOB, LONG VARGRAPHIC, or DBCLOB. In addition, it cannot be a BLOB file reference variable. The expression can be specified by any of the following:

- A constant
- A special register
- A host variable
- A scalar function whose operands are any of the above
- An expression that concatenates (using CONCAT or ||) any of the above
- An SQL procedure parameter

These rules are similar to those that are described for *pattern-expression* for the LIKE predicate.

*source-string*

An expression that specifies the string in which the search is to take place. The expression must return a value that is a built-in character string data type, graphic string data type, or binary string data type. The expression can be specified by any of the following:

- 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 that concatenates (using CONCAT or ||) any of the above

## POSITION scalar function

### CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the string unit of the result. CODEUNITS16 specifies that the result is to be expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that the result is to be expressed in 32-bit UTF-32 code units. OCTETS specifies that the result is to be expressed in bytes.

If a string unit is specified as CODEUNITS16 or CODEUNITS32, and *search-string* or *source-string* is a binary string or bit data, an error is returned (SQLSTATE 428GC). If a string unit is specified as OCTETS and *search-string* and *source-string* are binary strings, an error is returned (SQLSTATE 42815).

If a locale-sensitive UCA-based collation is used for this function, then the CODEUNITS16 option offers the best performance characteristics.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

The first and second arguments must have compatible string types. For more information on compatibility, see “Rules for string conversions”. In a Unicode database, if one string argument is character (not FOR BIT DATA) and the other string argument is graphic, then the *search-string* is converted to the data type of the *source-string* for processing. If one argument is character FOR BIT DATA, the other argument must not be graphic (SQLSTATE 42846).

The result of the function is a large integer. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples:

- Select the RECEIVED column, the SUBJECT column, and the starting position of the string ‘GOOD BEER’ within the NOTE\_TEXT column for all rows in the IN\_TRAY table that contain that string.

```
SELECT RECEIVED, SUBJECT, POSITION('GOOD BEER', NOTE_TEXT, OCTETS)
FROM IN_TRAY
WHERE POSITION('GOOD BEER', NOTE_TEXT, OCTETS) <> 0
```

- Find the position of the character ‘ß’ in the string ‘Jürgen lives on Hegelstraße’, and set the host variable LOCATION with the position, as measured in CODEUNITS32 units, within the string.

```
SET :LOCATION = POSITION(
 'ß', 'Jürgen lives on Hegelstraße', CODEUNITS32
)
```

The value of host variable LOCATION is set to 26.

- Find the position of the character ‘ß’ in the string ‘Jürgen lives on Hegelstraße’, and set the host variable LOCATION with the position, as measured in OCTETS, within the string.

```
SET :LOCATION = POSITION(
 'ß', 'Jürgen lives on Hegelstraße', OCTETS
)
```

The value of host variable LOCATION is set to 27.

- The following examples work with the Unicode string ‘&N~AB’, where ‘&’ is the musical symbol G clef character, and ‘~’ is the non-spacing combining tilde character. This string is shown below in different Unicode encoding forms:

|       | ‘&’         | ‘N’   | ‘~’     | ‘A’   | ‘B’   |
|-------|-------------|-------|---------|-------|-------|
| UTF-8 | X'F09D849E' | X'4E' | X'CC83' | X'41' | X'42' |

|          | '&'         | 'N'     | '~'     | 'A'     | 'B'     |
|----------|-------------|---------|---------|---------|---------|
| UTF-16BE | X'D834DD1E' | X'004E' | X'0303' | X'0041' | X'0042' |

Assume that the variable UTF8\_VAR contains the UTF-8 representation of the string.

```
SELECT POSITION('N', UTF8_VAR, CODEUNITS16),
 POSITION('N', UTF8_VAR, CODEUNITS32),
 POSITION('N', UTF8_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 3, 2, and 5, respectively.

Assume that the variable UTF16\_VAR contains the UTF-16BE representation of the string.

```
SELECT POSITION('B', UTF16_VAR, CODEUNITS16),
 POSITION('B', UTF16_VAR, CODEUNITS32),
 POSITION('B', UTF16_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

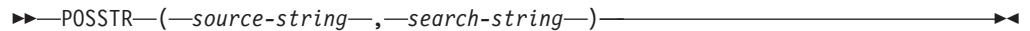
returns the values 6, 5, and 11, respectively.

- In a Unicode database created with the case insensitive collation UCA500R1\_LEN\_S1, find the position of the word 'Brown' in the phrase 'The quick brown fox'.

```
SET :LOCATION = POSITION('Brown', 'The quick brown fox', CODEUNITS16)
```

The value of the host variable LOCATION is set to 11.

## POSSTR scalar function



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 global variable
- 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

## POSSTR scalar function

### *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 global variable
- A host variable
- A scalar function whose operands are any of the above
- An expression concatenating any of the above
- An SQL procedure parameter

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.

The following are examples of invalid string expressions or strings:

- SQL user-defined function parameters
- Trigger transition variables
- Local variables in dynamic compound statements

In a Unicode database, if one argument is character (not FOR BIT DATA) and the other argument is graphic, then the *search-string* is converted to the data type of the *source-string* for processing. If one argument is character FOR BIT DATA, the other argument must not be graphic (SQLSTATE 42846).

Both *search-string* and *source-string* have zero or more contiguous positions. If the strings are character or binary strings, a position is a byte. If the strings are graphic strings, a position is a graphic (DBCS) character.

The POSSTR function accepts mixed data strings. However, POSSTR operates on a strict byte-count basis, oblivious to the database collation and 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—(—*expression1*—,—*expression2*—)—◄◄

The schema is SYSIBM. (The SYSFUN version of the POWER function continues to be available.)

The POWER function returns the result of raising the first argument to the power of the second argument.

The arguments can be of any built-in numeric data type. DECIMAL and REAL arguments are converted to a double-precision floating-point number. If either argument is decimal floating-point, the arguments are converted to DECFLOAT(34) for processing by the function.

The result of the function is:

- INTEGER if both arguments are INTEGER or SMALLINT
- BIGINT if one argument is BIGINT and the other argument is BIGINT, INTEGER or SMALLINT
- DECFLOAT(34) if one of the arguments is decimal floating-point. If either argument is a DECFLOAT and one of the following statements is true, the result is NAN and an invalid operation condition:
  - Both arguments are zero
  - The second argument has a non-zero fractional part
  - The second argument has more than 9 digits
  - The second argument is INFINITY
- DOUBLE otherwise

If the argument is a special decimal floating-point value, the rules for general arithmetic operations for decimal floating-point apply. See “General arithmetic operation rules for decimal floating-point” in “General arithmetic operation rules for decimal floating-point” on page 156.

The result can be null; if any argument is null, the result is the null value.

Example:

- Assume that the host variable HPOWER is an integer with a value of 3.

```
VALUES POWER(2, :HPOWER)
```

Returns the value 8.

## QUANTIZE

►►—QUANTIZE—(*numeric-expression*—,—*exp-expression*—)—◄◄

The schema is SYSIBM.

The QUANTIZE function returns a decimal floating-point value that is equal in value (except for any rounding) and sign to *numeric-expression* and that has an exponent equal to the exponent of *exp-expression*. The number of digits (16 or 34) is the same as the number of digits in *numeric-expression*.

### *numeric-expression*

An expression that returns a value of any built-in numeric data type. If the argument is not a decimal floating-point value, it is converted to DECFLOAT(34) for processing.

### *exp-expression*

An expression that returns a value of any built-in numeric data type. If the argument is not a decimal floating-point value, it is converted to DECFLOAT(34) for processing. The *exp-expression* is used as an example pattern for rescaling *numeric-expression*. The sign and coefficient of *exp-expression* are ignored.

The coefficient of the result is derived from that of *numeric-expression*. It is rounded, if necessary (if the exponent is being increased), multiplied by a power of ten (if the exponent is being decreased), or remains unchanged (if the exponent is already equal to that of *exp-expression*).

The CURRENT DECFLOAT ROUNDING MODE special register determines the rounding mode.

Unlike other arithmetic operations on the decimal floating-point data type, if the length of the coefficient after the quantize operation is greater than the precision specified by *exp-expression*, the result is NaN and a warning is returned (SQLSTATE 0168D). This ensures that, unless there is a warning condition, the exponent of the result of QUANTIZE is always equal to that of *exp-expression*.

- if either argument is NaN, NaN is returned
- if either argument is sNaN, NaN is returned and a warning is returned (SQLSTATE 01565)
- if both arguments are infinity (positive or negative), infinity with the same sign as the first argument is returned
- if one argument is infinity (positive or negative) and the other argument is not infinity, NaN is returned and a warning is returned (SQLSTATE 0168D)

The result of the function is a DECFLOAT(16) value if both arguments are DECFLOAT(16). Otherwise, the result of the function is a DECFLOAT(34) value. The result can be null; if any argument is null, the result is the null value.

### Examples:

- The following examples show the values that are returned by the QUANTIZE function given a variety of input decimal floating-point values and assuming a rounding mode of ROUND\_HALF\_UP:

```

QUANTIZE(2.17, DECFLOAT(0.001)) = 2.170
QUANTIZE(2.17, DECFLOAT(0.01)) = 2.17
QUANTIZE(2.17, DECFLOAT(0.1)) = 2.2
QUANTIZE(2.17, DECFLOAT('1E+0')) = 2
QUANTIZE(2.17, DECFLOAT('1E+1')) = 0E+1
QUANTIZE(2, DECFLOAT(INFINITY)) = NaN -- warning
QUANTIZE(0, DECFLOAT('1E+5')) = 0E+5
QUANTIZE(217, DECFLOAT('1E-1')) = 217.0
QUANTIZE(217, DECFLOAT('1E+0')) = 217
QUANTIZE(217, DECFLOAT('1E+1')) = 2.2E+2
QUANTIZE(217, DECFLOAT('1E+2')) = 2E+2

```

- In the following example the value -0 is returned for the QUANTIZE function. The CHAR function is used to avoid the potential removal of the minus sign by a client program:

```
CHAR(QUANTIZE(-0.1, DECFLOAT(1))) = -0
```

## QUARTER

►►—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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

## RADIANS

►►—RADIANS—(—*expression*—)—————►►

The schema is SYSIBM. (The SYSFUN version of the RADIANS function continues to be available.)

The RADIANS function returns the number of radians for an argument that is expressed in degrees.

The argument can be any built-in numeric data type. If the argument is decimal floating-point, the operation is performed in decimal floating-point; otherwise, the argument is converted to double-precision floating-point for processing by the function.

If the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*); otherwise, the result is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example:

## RADIANS

- Assume that host variable HDEG is an INTEGER with a value of 180. The following statement:

```
VALUES RADIANS(:HDEG)
```

Returns the value +3.14159265358979E+000.

## RAISE\_ERROR

►—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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

### *sqlstate*

A character string containing exactly 5 bytes. It must be of type CHAR defined with a length of 5 or type VARCHAR defined with a length of 5 or greater. 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' through '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.

To use this function in a context where the 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.

```
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.

The RAND function returns a floating point value between 0 and 1.

If an expression is specified, it is used as the seed value. The expression must be a built-in SMALLINT or INTEGER data type with a value between 0 and 2 147 483 647.

The data type of the result is double-precision floating point. If the argument is null, the result is the null value.

A specific seed value will produce the same sequence of random numbers for a specific instance of a RAND function in a query each time the query is executed. If a seed value is not specified, a different sequence of random numbers is produced each time the query is executed within the same session. To produce a set of random numbers that varies from session to session, specify a random seed; for example, one that is based on the current time.

RAND is a non-deterministic function.

## 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.



►</row-tag-string→—————▶

Column names may or may not be valid XML attribute values. For column names which are not valid XML attribute values, character replacement is performed on the column name before it is included in the result string.

Column values may or may not be valid XML element names. If the *format-string* COLATTVAL is specified, then for the column names which are not valid XML element values, character replacement is performed on the column value before it is included in the result string. If the *format-string* COLATTVAL\_XML is specified, then character replacement is not performed on column values (although character replacement is still performed on column names).

#### *row-tag-string*

A string constant that specifies the tag used for each row. If an empty string is specified, then a value of 'row' is assumed.

If a string of one or more blank characters is specified, then no beginning *row-tag-string* or ending *row-tag-string* (including the angle bracket delimiters) will appear in the result string.

#### *column-name*

A qualified or unqualified name of a table column. The column must have one of the following data types (SQLSTATE 42815):

- numeric (SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, DOUBLE)
- character string (CHAR, VARCHAR; a character string with a subtype of BIT DATA is not allowed)
- datetime (DATE, TIME, TIMESTAMP)
- a user-defined type based on one of the above types

The same column name cannot be specified more than once (SQLSTATE 42734).

The result of the function is VARCHAR. The maximum length is 32 672 bytes (SQLSTATE 54006).

Consider the following invocation:

```
REC2XML (dc, fs, rt, c1, c2, ..., cn)
```

If the value of "fs" is either "COLATTVAL" or "COLATTVAL\_XML", then the result is the same as this expression:

```
'<' CONCAT rt CONCAT '>' CONCAT y1 CONCAT y2
CONCAT ... CONCAT yn CONCAT '</' CONCAT rt CONCAT '>'
```

where y<sub>n</sub> is equivalent to:

```
'<column name="' CONCAT xvcn CONCAT vn
```

and vn is equivalent to:

```
'">' CONCAT rn CONCAT '</column>'
```

if the column is not null, and

```
'" null="true"/>'
```

if the column value is null.

$xvc_n$  is equivalent to a string representation of the column name of  $c_n$ , where any characters appearing in Table 37 are replaced with the corresponding representation. This ensures that the resulting string is a valid XML attribute or element value token.

The  $r_n$  is equivalent to a string representation as indicated in Table 36

Table 36. Column Values String Result

| Data type of $c_n$                                        | $r_n$                                                                                                                                                                                                                                                                                   |
|-----------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CHAR, VARCHAR                                             | The value is a string. If the <i>format-string</i> does not end in the characters "_XML", then each character in $c_n$ is replaced with the corresponding replacement representation from Table 37, as indicated. The length attribute is: $dc * \text{the length attribute of } c_n$ . |
| SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC, REAL, DOUBLE | The value is $LTRIM(RTRIM(CHAR(c_n)))$ . The length attribute is the result length of $CHAR(c_n)$ . The decimal character is always the period (".") character.                                                                                                                         |
| DATE                                                      | The value is $CHAR(c_n,ISO)$ . The length attribute is the result length of $CHAR(c_n,ISO)$ .                                                                                                                                                                                           |
| TIME                                                      | The value is $CHAR(c_n,JIS)$ . The length attribute is the result length of $CHAR(c_n,JIS)$ .                                                                                                                                                                                           |
| TIMESTAMP                                                 | The value is $CHAR(c_n)$ . The length attribute is the result length of $CHAR(c_n)$ .                                                                                                                                                                                                   |

Character replacement:

Depending on the value specified for the *format-string*, certain characters in column names and column values will be replaced to ensure that the column names form valid XML attribute values and the column values form valid XML element values.

Table 37. Character Replacements for XML Attribute Values and Element Values

| Character | Replacement |
|-----------|-------------|
| <         | &lt;        |
| >         | &gt;        |
| "         | &quot;      |
| &         | &amp;       |
| '         | &apos;      |

Examples:

**Note:** REC2XML does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- Using the DEPARTMENT table in the sample database, format the department table row, except the DEPTNAME and LOCATION columns, for department 'D01' into an XML string. Since the data does not contain any of the characters which require replacement, the expansion factor will be 1.0 (no expansion). Also note that the MGRNO value is null for this row.

```
SELECT REC2XML (1.0, 'COLATTVAL', '', DEPTNO, MGRNO, ADMRDEPT)
FROM DEPARTMENT
WHERE DEPTNO = 'D01'
```

This example returns the following VARCHAR(117) string:

```
<row>
<column name="DEPTNO">D01</column>
<column name="MGRNO" null="true"/>
<column name="ADMRDEPT">A00</column>
</row>
```

- A 5-day university schedule introduces a class named '&43<FIE' to a table called CL\_SCHED, with a new format for the CLASS\_CODE column. Using the REC2XML function, this example formats an XML string with this new class data, except for the class end time.

The length attribute for the REC2XML call (see below) with an expansion factor of 1.0 would be 128 (11 for the '<row>' and '</row>' overhead, 21 for the column names, 75 for the '<column name=', '>', '</column>' and double quotes, 7 for the CLASS\_CODE data, 6 for the DAY data, and 8 for the STARTING data). Since the '&' and '<' characters will be replaced, an expansion factor of 1.0 will not be sufficient. The length attribute of the function will need to support an increase from 7 to 14 bytes for the new format CLASS\_CODE data.

However, since it is known that the DAY value will never be more than 1 digit long, an unused extra 5 units of length are added to the total. Therefore, the expansion only needs to handle an increase of 2. Since CLASS\_CODE is the only character string column in the argument list, this is the only column data to which the expansion factor applies. To get an increase of 2 for the length, an expansion factor of 9/7 (approximately 1.2857) would be needed. An expansion factor of 1.3 will be used.

```
SELECT REC2XML (1.3, 'COLATTVAL', 'record', CLASS_CODE, DAY, STARTING)
FROM CL_SCHED
WHERE CLASS_CODE = '&43<FIE'
```

This example returns the following VARCHAR(167) string:

```
<record>
<column name="CLASS_CODE">&43<FIE</column>
<column name="DAY">5</column>
<column name="STARTING">06:45:00</column>
</record>
```

- Assume that new rows have been added to the EMP\_RESUME table in the sample database. The new rows store the resumes as strings of valid XML. The COLATTVAL\_XML *format-string* is used so character replacement will not be carried out. None of the resumes are more than 3500 bytes in length. The following query is used to select the XML version of the resumes from the EMP\_RESUME table and format it into an XML document fragment.

```
SELECT REC2XML (1.0, 'COLATTVAL_XML', 'row', EMPNO, RESUME_XML)
FROM (SELECT EMPNO, CAST(RESUME AS VARCHAR(3500)) AS RESUME_XML
FROM EMP_RESUME
WHERE RESUME_FORMAT = 'XML')
AS EMP_RESUME_XML
```

This example returns a row for each employee who has a resume in XML format. Each returned row will be a string with the following format:

```
<row>
<column name="EMPNO">{employee number}</column>
<column name="RESUME_XML">{resume in XML}</column>
</row>
```

Where "{employee number}" is the actual EMPNO value for the column and "{resume in XML}" is the actual XML fragment string value that is the resume.

## 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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The first argument is a character string or binary string type. For a VARCHAR the maximum length is 4000 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 4000 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 this example, the CHAR function has been used to limit the output of REPEAT to 60 bytes.

## REPLACE

►►—REPLACE—(—*source-string*—,—*search-string*—,—*replace-string*—)—————►►

The schema is SYSIBM. The SYSFUN version of the REPLACE function continues to be available but it is not sensitive to the database collation.

Replaces all occurrences of *search-string* in *source-string* with *replace-string*. If *search-string* is not found in *source-string*, *search-string* is returned unchanged. The search is done using the collation of the database unless *source-string*, *search-string* or *replace-string* is defined as FOR BIT DATA, in which case the search is done using a binary comparison.

*source-string*

An expression that specifies the source string. The expression must return a value that is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type.

*search-string*

An expression that specifies the string to be removed from the source string. The expression must return a value that is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type.

*replace-string*

An expression that specifies the replacement string. The expression must return a value that is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the expression is an empty string, nothing replaces the string that is removed from the source string.

The actual length of each string must be 32 672 bytes or less for character strings, or 16 336 or less for graphic strings. All three arguments must have compatible data types.

If *source-string*, *search-string* or *replace-string* is defined as FOR BIT DATA, the result is VARCHAR FOR BIT DATA. If *source-string* is a character string, the result is VARCHAR. If *source-string* is a graphic string, the result is VARGRAPHIC. If one argument is character FOR BIT DATA, the other arguments must not be graphic (SQLSTATE 42846).

The length attribute of the result depends on the arguments:

- If the length attribute of *replace-string* is less than or equal to the length attribute of *search-string*, the length attribute of the result is the length attribute of *source-string*.
- If the length attribute of *replace-string* is greater than the length attribute of *search-string*, the length attribute of the result is determined as follows, depending on the data type of the result:
  - For VARCHAR:
    - If  $L1 \leq 4000$ , the length attribute of the result is  $\text{MIN}(4000, (L3 * (L1/L2)) + \text{MOD}(L1, L2))$
    - Otherwise, the length attribute of the result is  $\text{MIN}(32672, (L3 * (L1/L2)) + \text{MOD}(L1, L2))$
  - For VARGRAPHIC:
    - If  $L1 \leq 2000$ , the length attribute of the result is  $\text{MIN}(2000, (L3 * (L1/L2)) + \text{MOD}(L1, L2))$
    - Otherwise, the length attribute of the result is  $\text{MIN}(16336, (L3 * (L1/L2)) + \text{MOD}(L1, L2))$

where:

- L1 is the length attribute of *source-string*
- L2 is the length attribute of the *search-string* if the search string is a string constant. Otherwise, L2 is 1.
- L3 is the length attribute of *replace-string*

If the result is a character string, the length attribute of the result must not exceed 32 672. If the result is a graphic string, the length attribute of the result must not exceed 16 336.





- Return the RID and last name of employees in department 20 from the EMPLOYEE table.

```
SELECT RID_BIT (EMPLOYEE), ROW CHANGE TOKEN FOR EMPLOYEE, LASTNAME
FROM EMPLOYEE
WHERE DEPTNO = '20'
```

- Given table EMP1, which is defined as follows:

```
CREATE TABLE EMP1 (
EMPNO CHAR(6),
NAME CHAR(30),
SALARY DECIMAL(9,2),
PICTURE BLOB(250K),
RESUME CLOB(32K)
)
```

Set host variable HV\_EMP\_RID to the value of the RID\_BIT built-in scalar function, and HV\_EMP\_RCT to the value of the ROW CHANGE TOKEN expression for the row corresponding to employee number 3500.

```
SELECT RID_BIT(EMP1), ROW CHANGE TOKEN FOR EMP1
INTO :HV_EMP_RID, :HV_EMP_RCT FROM EMP1
WHERE EMPNO = '3500'
```

Using that RID value to identify the employee, and user-defined function UPDATE\_RESUME, increase the employee's salary by \$1000 and update the employee's resume.

```
UPDATE EMP1 SET
SALARY = SALARY + 1000,
RESUME = UPDATE_RESUME(:HV_RESUME)
WHERE RID_BIT(EMP1) = :HV_EMP_RID
AND ROW CHANGE TOKEN FOR EMP1 = :HV_EMP_RCT
```

## RIGHT

►► RIGHT (—*string-expression*—, —*length*—) —►►

, CODEUNITS16  
CODEUNITS32  
OCTETS

The schema is SYSIBM. The SYSFUN version of the RIGHT function continues to be available.

The RIGHT function returns the rightmost string of *string-expression* of length *length*, expressed in the specified string unit. If *string-expression* is a character string, the result is a character string. If *string-expression* is a graphic string, the result is a graphic string

### *string-expression*

An expression that specifies the string from which the result is derived. The expression must return a value that is a built-in character or graphic string data type (SQLSTATE 42815). A substring of *string-expression* is zero or more contiguous code points of *string-expression*.

### *length*

An expression that specifies the length of the result. The result must be a built-in integer data type (SQLSTATE 42815). The value must be between 0 and the length of *string-expression*, expressed in units that are either implicitly or explicitly specified (SQLSTATE 22011). If OCTETS is specified and the result is

graphic data, the value must be an even number between 0 and twice the length attribute of *string-expression* (SQLSTATE 428GC).

### CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the string unit of *length*.

CODEUNITS16 specifies that *length* is expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *length* is expressed in 32-bit UTF-32 code units. OCTETS specifies that *length* is expressed in bytes.

If the string unit is specified as CODEUNITS16 or CODEUNITS32, and *string-expression* is a binary string or bit data, an error is returned (SQLSTATE 428GC). If the string unit is specified as OCTETS and *string-expression* is a graphic string, *length* must be an even number; otherwise, an error is returned (SQLSTATE 428GC). If a string unit is not explicitly specified, the data type of the result determines the unit that is used. If the result is graphic data, *length* is expressed in two-byte units; otherwise, it is expressed in bytes. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

The *string-expression* is padded on the right with the necessary number of padding characters so that the specified substring of *string-expression* always exists. The character used for padding is the same character that is used to pad the string in contexts where padding would occur. For more information on padding, see “String assignments” in “Assignments and comparisons”.

The result of the function is a varying-length string with a length attribute that is the same as the length attribute of *string-expression* and a data type that depends on the data type of *string-expression*:

- VARCHAR if *string-expression* is CHAR or VARCHAR
- CLOB if *string-expression* is CLOB
- VARGRAPHIC if *string-expression* is GRAPHIC or VARGRAPHIC
- DBCLOB if *string-expression* is DBCLOB

The actual length of the result (in string units) is *length*.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples:

- Assume that variable ALPHA has a value of 'ABCDEF'. The following statement:

```
SELECT RIGHT(ALPHA,3)
FROM SYSIBM.SYSDUMMY1
```

returns 'DEF', which are the three rightmost characters in ALPHA.

- Assume that variable NAME, which is defined as VARCHAR(50), has a value of 'KATIE AUSTIN', and that the integer variable LASTNAME\_LEN has a value of 6. The following statement:

```
SELECT RIGHT(NAME, LASTNAME_LEN)
FROM SYSIBM.SYSDUMMY1
```

returns the value 'AUSTIN'.

- The following statement returns a zero-length string.

```
SELECT RIGHT('ABCABC',0)
FROM SYSIBM.SYSDUMMY1
```

- The FIRSTNAME column in the EMPLOYEE table is defined as VARCHAR(12). Find the first name of an employee whose last name is 'BROWN' and return the first name in a 10-byte string.

```
SELECT RIGHT(FIRSTNAME, 10)
FROM EMPLOYEE
WHERE LASTNAME = 'BROWN'
```

returns a VARCHAR(12) string that has the value 'DAVID' followed by five blank characters.

- In a Unicode database, FIRSTNAME is a VARCHAR(12) column. One of its values is the 6-character string 'Jürgen'. When FIRSTNAME has this value:

Function...	Returns...
<code>RIGHT(FIRSTNAME,5,CODEUNITS32)</code>	'ürgen' -- x'C3BC7267656E'
<code>RIGHT(FIRSTNAME,5,CODEUNITS16)</code>	'ürgen' -- x'C3BC7267656E'
<code>RIGHT(FIRSTNAME,5,OCTETS)</code>	'rgen' -- x'207267656E', a truncated string

- The following example works with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown below in different Unicode encoding forms:

	'&'	'N'	'~'	'A'	'B'
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'

Assume that the variable UTF8\_VAR, with a length attribute of 20 bytes, contains the UTF-8 representation of the string.

```
SELECT RIGHT(UTF8_VAR, 2, CODEUNITS16),
 RIGHT(UTF8_VAR, 2, CODEUNITS32),
 RIGHT(UTF8_VAR, 2, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 'AB', 'AB', and 'AB', respectively.

```
SELECT RIGHT(UTF8_VAR, 5, CODEUNITS16),
 RIGHT(UTF8_VAR, 5, CODEUNITS32),
 RIGHT(UTF8_VAR, 5, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '?N~AB', '&N~AB', and 'N~AB', respectively, where '?' is X'EDB49E'.

```
SELECT RIGHT(UTF8_VAR, 10, CODEUNITS16),
 RIGHT(UTF8_VAR, 10, CODEUNITS32),
 RIGHT(UTF8_VAR, 10, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~ABbbbb', '&N~AB**bbbbb**', and '&N~AB**b**', respectively, where '**b**' represents the blank character.

Assume that the variable UTF16\_VAR, with a length attribute of 20 code units, contains the UTF-16BE representation of the string.

```
SELECT RIGHT(UTF16_VAR, 2, CODEUNITS16),
 RIGHT(UTF16_VAR, 2, CODEUNITS32),
 RIGHT(UTF16_VAR, 2, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 'AB', 'AB', and 'B', respectively.

## RIGHT

```
SELECT RIGHT(UTF16_VAR, 5, CODEUNITS16),
 RIGHT(UTF16_VAR, 5, CODEUNITS32),
 RIGHT(UTF16_VAR, 6, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '?N~AB', '&N~AB', and '~AB', respectively, where '?' is the standalone low surrogate X'DD1E'.

```
SELECT RIGHT(UTF16_VAR, 10, CODEUNITS16),
 RIGHT(UTF16_VAR, 10, CODEUNITS32),
 RIGHT(UTF16_VAR, 10, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~AB**bbbb**', '&N~AB**bbbb**', and '?N~AB', respectively, where 'b' represents the blank character and '?' is X'DD1E'.

## ROUND

►►—ROUND—(—*expression1*—,—*expression2*—)—————►►

The schema is SYSIBM. (The SYSFUN version of the ROUND function continues to be available.)

The ROUND function returns *expression1* rounded to *expression2* places to the right of the decimal point if *expression2* is positive, or to the left of the decimal point if *expression2* is zero or negative.

If *expression1* is positive, a digit value of 5 or greater is an indication to round to the next higher positive number. For example, ROUND(3.5,0) = 4. If *expression1* is negative, a digit value of 5 or greater is an indication to round to the next lower negative number. For example, ROUND(-3.5,0) = -4.

*expression1*

An expression that returns a value of any built-in numeric data type.

*expression2*

An expression that returns a small or large integer. When the value of *expression2* is not negative, it specifies rounding to that number of places to the right of the decimal separator. When the value of *expression2* is negative, it specifies rounding to the absolute value of *expression2* places to the left of the decimal separator.

If *expression2* is not negative, *expression1* is rounded to the absolute value of *expression2* number of places to the right of the decimal point. If the value of *expression2* is greater than the scale of *expression1* then the value is unchanged except that the result value has a precision that is larger by 1. For example, ROUND(748.58,5) = 748.58 where the precision is now 6 and the scale remains 2.

If *expression2* is negative, *expression1* is rounded to the absolute value of *expression2*+1 number of places to the left of the decimal point.

If the absolute value of a negative *expression2* is larger than the number of digits to the left of the decimal point, the result is 0. For example, ROUND(748.58,-4) = 0.

The data type and length attribute of the result are the same as the data type and length attribute of the first argument, except that the precision is increased by one if the *expression1* is DECIMAL and the precision is less than 31.

For example, an argument with a data type of DECIMAL(5,2) results in DECIMAL(6,2). An argument with a data type of DECIMAL(31,2) results in DECIMAL(31,2). The scale is the same as the scale of the first argument.

If either argument can be null or if the argument is not a decimal floating-point number and the database is configured with DFT\_SQLMATHWARN set to YES, the result can be null. If either argument is null, the result is the null value.

This function is not affected by the setting of the CURRENT DECFLOAT ROUNDING MODE special register, even for decimal floating-point arguments. The rounding behavior of ROUND corresponds to a value of ROUND\_HALF\_UP. If you want behavior for a decimal floating-point value that conforms to the rounding mode specified by the CURRENT DECFLOAT ROUNDING MODE special register, use the QUANTIZE function instead.

Examples:

Calculate the value of 873.726, rounded to 2, 1, 0, -1, -2, -3, and -4 decimal places, respectively.

```
VALUES (
 ROUND(873.726, 2),
 ROUND(873.726, 1),
 ROUND(873.726, 0),
 ROUND(873.726,-1),
 ROUND(873.726,-2),
 ROUND(873.726,-3),
 ROUND(873.726,-4))
```

This example returns:

```
1 2 3 4 5 6 7

873.730 873.700 874.000 870.000 900.000 1000.000 0.000
```

Calculate using both positive and negative numbers.

```
VALUES (
 ROUND(3.5, 0),
 ROUND(3.1, 0),
 ROUND(-3.1, 0),
 ROUND(-3.5,0))
```

This example returns:

```
1 2 3 4

4.0 3.0 -3.0 -4.0
```

Calculate the decimal floating-point number 3.12350 rounded to three decimal places.

```
VALUES (
 ROUND(DECFLOAT('3.12350'), 3))
```

This example returns:

```
1

3.12400
```

## RTRIM

►► RTRIM(*(—string-expression—)*) ◀◀

The schema is SYSIBM. (The SYSFUN version of this function continues to be available with support for LONG VARCHAR and CLOB arguments.)

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 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'.

## SECLABEL

►► SECLABEL(*(—security-policy-name—, —security-label-string—)*) ◀◀

The schema is SYSIBM.

The SECLABEL function returns an unnamed security label with a data type of DB2SECURITYLABEL. Use the SECLABEL function to insert a security label with given component values without having to create a named security label.

*security-policy-name*

A string that specifies a security policy that exists at the current server (SQLSTATE 42704). The string must be a character or graphic string constant or host variable.

*security-label-string*

An expression that returns a valid representation of a security label for the security policy named by *security-policy-name* (SQLSTATE 4274I). The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type.

## Examples:

- The following statement inserts a row in table REGIONS which is protected by the security policy named CONTRIBUTIONS. The security label for the row to be inserted is given by the SECLABEL function. The security policy CONTRIBUTIONS has two components. The security label given has the element LIFE MEMBER for first component, the elements BLUE and YELLOW for the second component.

```
INSERT INTO REGIONS
VALUES (SECLABEL('CONTRIBUTIONS', 'LIFE MEMBER:(BLUE,YELLOW)'),
1, 'Northeast')
```

- The following statement inserts a row in table CASE\_IDS which is protected by the security policy named TS\_SECPOLICY, which has three components. The security label is provided by the SECLABEL function. The security label inserted has the element HIGH PROFILE for the first component, the empty value for the second component and the element G19 for the third component.

```
INSERT INTO CASE_IDS
VALUES (SECLABEL('TS_SECPOLICY', 'HIGH PROFILE:():G19') , 3, 'KLB')
```

**SECLABEL\_BY\_NAME**

►►—SECLABEL\_BY\_NAME—(—*security-policy-name*—,—*security-label-name*—)————►►

The schema is SYSIBM.

The SECLABEL\_BY\_NAME function returns the specified security label. The security label returned has a data type of DB2SECURITYLABEL. Use this function to insert a named security label.

*security-policy-name*

A string that specifies a security policy that exists at the current server (SQLSTATE 42704). The string must be a character or graphic string constant or host variable.

*security-label-name*

An expression that returns the name of a security label that exists at the current server for the security policy named by *security-policy-name* (SQLSTATE 4274I). The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type.

## Examples:

- User Tina is trying to insert a row in table REGIONS which is protected by the security policy named CONTRIBUTIONS. Tina wants the row to be protected by the security label named EMPLOYEESECLABEL. This statement fails because CONTRIBUTIONS.EMPLOYEESECLABEL is an unknown identifier:

## SECLABEL\_BY\_NAME

```
INSERT INTO REGIONS
VALUES (CONTRIBUTIONS.EMPLOYEESECLABEL, 1, 'Southwest') -- incorrect
```

This statement fails because the first value is a string, it does not have a data type of DB2SECURITYLABEL:

```
INSERT INTO REGIONS
VALUES ('CONTRIBUTIONS.EMPLOYEESECLABEL', 1, 'Southwest') -- incorrect
```

This statement succeeds because the SECLABEL\_BY\_NAME function returns a security label that has a data type of DB2SECURITYLABEL:

```
INSERT INTO REGIONS
VALUES (SECLABEL_BY_NAME('CONTRIBUTIONS', 'EMPLOYEESECLABEL'),
1, 'Southwest') -- correct
```

## SECLABEL\_TO\_CHAR

►►—SECLABEL\_TO\_CHAR—(—*security-policy-name*—,—*security-label*—)—————►►

The schema is SYSIBM.

The SECLABEL\_TO\_CHAR function accepts a security label and returns a string that contains all elements in the security label. The string is in the security label string format.

### *security-policy-name*

A string that specifies a security policy that exists at the current server (SQLSTATE 42704). The string must be a character or graphic string constant or host variable.

### *security-label*

An expression that returns a security label value that is valid for the security policy named by *security-policy-name* (SQLSTATE 4274I). The expression must return a value that is a built-in SYSPROC.DB2SECURITYLABEL distinct type.

## Notes

- If the authorization ID of the statement executes this function on a security label being read from a column with a data type of DB2SECURITYLABEL then that authorization ID's LBAC credentials might affect the output of the function. In such a case an element is not included in the output if the authorization ID does not have read access to that element. An authorization ID has read access to an element if its LBAC credentials would allow it to read data that was protected by a security label containing only that element, and no others.

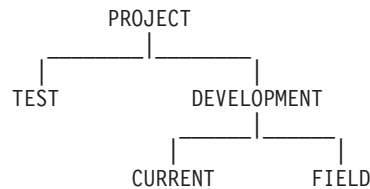
For the rule set DB2LBACRULES only components of type TREE can contain elements that you do not have read access to. For other types of component, if any one of the elements block read access then you will not be able to read the row at all. So only components of type tree will have elements excluded in this way.

Example:

- The EMP table has two columns, RECORDNUM and LABEL; RECORDNUM has data type INTEGER, and LABEL has type DB2SECURITYLABEL. Table EMP is protected by security policy DATA\_ACCESSPOLICY, which uses the DB2LBACRULES rule set and has only one component (GROUPS, of type



TREE). GROUPS has five elements: PROJECT, TEST, DEVELOPMENT, CURRENT, AND FIELD. The following diagram shows the relationship of these elements to one another:



The EMP table contains the following data:

RECORDNUM	LABEL
1	PROJECT
2	(TEST, FIELD)
3	(CURRENT, FIELD)

Djavan holds a security label for reading that contains only the DEVELOPMENT element. This means that Djavan has read access to the DEVELOPMENT, CURRENT, and FIELD elements:

```
SELECT RECORDNUM, SECLABEL_TO_CHAR('DATA_ACCESSPOLICY', LABEL) FROM EMP
```

returns:

RECORDNUM	LABEL
2	FIELD
3	(CURRENT, FIELD)

The row with a RECORDNUM value of 1 is not included in the output, because Djavan's LBAC credentials do not allow him to read that row. In the row with a RECORDNUM value of 2, element TEST is not included in the output, because Djavan does not have read access to that element; Djavan would not have been able to access the row at all if TEST were the only element in the security label. Because Djavan has read access to elements CURRENT and FIELD, both elements appear in the output.

Now Djavan is granted an exemption to the DB2LBACREADTREE rule. This means that no element of a TREE type component will block read access. The same query returns:

RECORDNUM	LABEL
1	PROJECT
2	(TEST, FIELD)
3	(CURRENT, FIELD)

This time the output includes all rows and all elements, because the exemption gives Djavan read access to all of the elements.

## SECOND

►►—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

## SECOND

nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- 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—(—*expression*—)—————►►

The schema is SYSIBM. (The SYSFUN version of the SIGN function continues to be available.)

Returns an indicator of the sign of the argument. If the argument is less than zero, -1 is returned. If the argument is the decimal floating-point value of -0, the decimal floating-point value of -0 is returned. If argument equals zero, 0 is returned. If argument is greater than zero, 1 is returned.

The argument can be of any built-in numeric data type. DECIMAL and REAL values are converted to double-precision floating-point numbers 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
- DECFLOAT(*n*) if the argument is DECFLOAT(*n*)
- DOUBLE otherwise.

The result can be null; if the argument is null, the result is the null value.

Example:

- Assume that host variable PROFIT is a large integer with a value of 50000.

```
VALUES SIGN(:PROFIT)
```

Returns the value 1.

## SIN

```
►►—SIN—(—expression—)—————►
```

The schema is SYSIBM. (The SYSFUN version of the SIN function continues to be available.)

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 type (except for DECFLOAT). 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 can be null or the database is configured with DFT\_SQLMATHWARN set to YES; the result is the null value if the argument is null.

## SINH

```
►►—SINH—(—expression—)—————►
```

The schema is SYSIBM.

Returns the hyperbolic sine of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric data type (except for DECFLOAT). 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 can be null or the database is configured with DFT\_SQLMATHWARN set to YES; the result is the null value if the argument is null.

## SMALLINT

```
►►—SMALLINT—(—numeric-expression—
 └—character-expression—┘)——————►
```

The schema is SYSIBM.

## SMALLINT

The SMALLINT function returns a small integer representation of a number or character string in the form of a small integer constant. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

### *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 4000 bytes. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed. The function interprets data that is passed to it as if it were ASCII characters, even if it is encoded in UTF-8.

The result of the function is CHAR(4). The result can be null; if the argument is null, the result is the null value.

The SOUNDEX function is useful for finding strings for which the sound is known but the precise spelling is not. It makes assumptions about the way that letters and combinations of letters sound that can help to search out words with similar sounds. The comparison can be done directly or by passing the strings as arguments to the DIFFERENCE function.

Example:

Using the EMPLOYEE table, find the EMPNO and LASTNAME of the employee with a surname that sounds like 'Loucesy'.

```
SELECT EMPNO, LASTNAME FROM EMPLOYEE
WHERE SOUNDEX(LASTNAME) = SOUNDEX('Loucesy')
```

This example returns the following:

```
EMPNO LASTNAME

000110 LUCCHESI
```

## SPACE

►►—SPACE—(—*expression*—)—————►►

The schema is SYSFUN.

Returns a character string consisting of blanks with length specified by the 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 SYSIBM. (The SYSFUN version of the SQRT function continues to be available.)

The SQRT function returns the square root of a number.

The argument must be an expression that returns a value of any built-in numeric data type. If the argument is decimal floating-point, the operation is performed in decimal floating-point; otherwise, the argument is converted to double-precision floating-point for processing by the function.

If the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*); otherwise, the result is a double-precision floating-point number.

If the argument is a special decimal floating-point value, the rules for general arithmetic operations for decimal floating-point apply. See “General arithmetic operation rules for decimal floating-point” in “General arithmetic operation rules for decimal floating-point” on page 156.

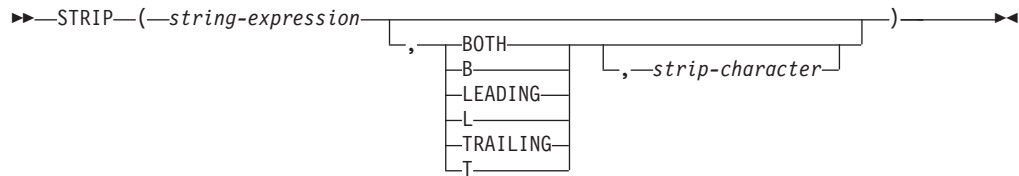
The result can be null; if the argument is null, the result is the null value.

Example:

- Assume that SQUARE is a DECIMAL(2,1) host variable with a value of 9.0.  
VALUES SQRT(:SQUARE)

Returns the approximate value 3.00.

## STRIP



The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature.

The STRIP function removes blanks or occurrences of another specified character from the end or the beginning of a string expression.

The STRIP function is identical to the TRIM scalar function.

*string-expression*

An expression that returns a value that is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type.

**BOTH, LEADING, or TRAILING**

Specifies whether characters are removed from the beginning, the end, or from both ends of the string expression. If this argument is not specified, the characters are removed from both the end and the beginning of the string.

*strip-character*

A single-character constant that specifies the character that is to be removed. The *strip-character* can be any character whose UTF-32 encoding is a single character. The binary representation of the character is matched.

If *strip-character* is not specified and:

- If the *string-expression* is a DBCS graphic string, the default *strip-character* is a DBCS blank, whose code point is dependent on the database code page
- If the *string-expression* is a UCS-2 graphic string, the default *strip-character* is a UCS-2 blank (X'0020')
- Otherwise, the default *strip-character* is an SBCS blank (X'20')

The result is a varying-length string with the same maximum length as the length attribute of the *string-expression*. The actual length of the result is the length of the *string-expression* minus the number of bytes that are removed. If all of the characters are removed, the result is an empty varying-length string. The code page of the result is the same as the code page of the *string-expression*.

Example:

- Assume that the host variable BALANCE of type CHAR(9) has a value of '000345.50'.

```
SELECT STRIP(:BALANCE, LEADING, '0'),
FROM SYSIBM.SYSDUMMY1
```

returns the value '345.50'.

## SUBSTR

►► SUBSTR(*string*, *start*, *length*)

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 the first argument is a host variable, the code page of the result is the database code page. If any argument of the SUBSTR function can be null, the result can be null; if any argument is null, the result is the null value.

#### *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) or hexadecimal zero characters (for BLOB 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 38 shows that the result type and length of the SUBSTR function depend on the type and attributes of its inputs.

Table 38. Data Type and Length of SUBSTR Result

String Argument Data Type	Length Argument	Result Data Type
CHAR(A)	constant ( $l < 255$ )	CHAR( <i>l</i> )
CHAR(A)	not specified but <i>start</i> argument is a constant	CHAR(A- <i>start</i> +1)

Table 38. Data Type and Length of SUBSTR Result (continued)

String Argument Data Type	Length Argument	Result Data Type
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 <i>start</i> 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)
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  $\text{LENGTH}(\text{string}) - \text{start} + 1$ . If *string* is a varying-length string, omission of *length* is an implicit specification of zero or  $\text{LENGTH}(\text{string}) - \text{start} + 1$ , 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)
```

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 '
```

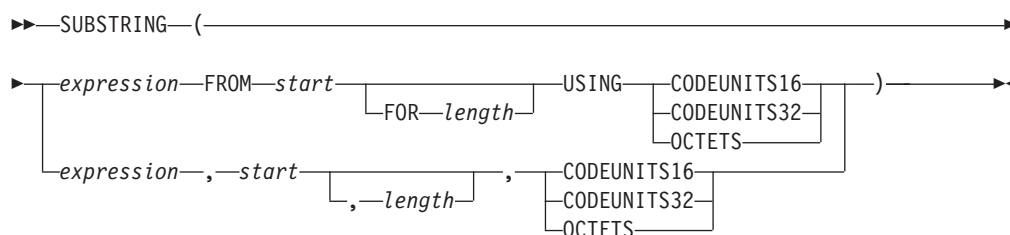


The space at the end of the constant is necessary to preclude initial words such as 'OPERATIONS'.

**Note:**

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.

## SUBSTRING



The schema is SYSIBM.

The SUBSTRING function returns a substring of a string.

*expression*

An expression that returns a value of any built-in string data type. If *expression* is a character string, the result is a character string. If *expression* is a graphic string, the result is a graphic string. If *expression* is a binary string, the result is a binary string.

A substring of *expression* is zero or more contiguous string units of *expression*.

*start*

An expression that specifies the position within *expression* that is to be the first string unit of the result; *start*, which is expressed in the specified string unit, must return an integer. The value of *start* can be positive, negative, or zero; a value of 1 indicates that the first string unit of the result is the first string unit of *expression*. If OCTETS is specified and *expression* is graphic data, *start* must be odd; otherwise, an error is returned (SQLSTATE 428GC).

*length*

An expression that specifies the maximum actual length of the resulting substring.

If *expression* is a fixed-length string, omission of *length* is an implicit specification of  $\text{CHARACTER\_LENGTH}(\text{expression USING string-unit}) - \text{start} + 1$ , which is the number of *string units* (CODEUNITS16, CODEUNITS32, or OCTETS) from *start* to the last position of *expression*. If *expression* is a varying-length string, omission of *length* is an implicit specification of zero or  $\text{CHARACTER\_LENGTH}(\text{expression USING string-unit}) - \text{start} + 1$ , whichever is greater. If the desired length is zero, the result is the empty string.

## SUBSTRING

If specified, *length* must be an expression that returns a value that is a built-in integer data type. The value must be greater than or equal to zero. If a value greater than *n* is specified, where *n* is the (length attribute of *expression*) - *start* + 1, then *n* is used as the length of the resulting substring. The value is expressed in the units that are explicitly specified. If OCTETS is specified, and *expression* is graphic data, *length* must be an even number (SQLSTATE 428GC).

### CODEUNITS16, CODEUNITS32, or OCTETS

Specifies the string unit of *start* and *length*. CODEUNITS16 specifies that *start* and *length* are to be expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *start* and *length* are to be expressed in 32-bit UTF-32 code units. OCTETS specifies that *start* and *length* are to be expressed in bytes.

If a string unit is specified as CODEUNITS16 or CODEUNITS32, and *expression* is a binary string or bit data, an error is returned (SQLSTATE 428GC). If a string unit is specified as OCTETS and *expression* is a binary string, an error is returned (SQLSTATE 42815).

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

When the SUBSTRING function is invoked using OCTETS, and the *source-string* is encoded in a code page that requires more than one byte per code point (mixed or MBCS), the SUBSTRING operation might split a multi-byte code point and the resulting substring might begin or end with a partial code point. If this occurs, the function replaces the bytes of leading or trailing partial code points with blanks in a way that does not change the byte length of the result. (See a related example below.)

The length attribute of the result is equal to the length attribute of *expression*. If any argument of the function can be null, the result can be null; if any argument is null, the result is the null value. The result is not padded with any character. If *expression* has actual length 0, the result also has actual length 0.

### Notes:

- The length attribute of the result is equal to the length attribute of the input string expression. This behavior is different from the behavior of the SUBSTR function, where the length attribute is derived from the *start* and the *length* arguments of the function.

### Examples:

- FIRSTNAME is a VARCHAR(12) column in table T1. One of its values is the 6-character string 'Jürgen'. When FIRSTNAME has this value:

Function ...	Returns ...
-----	-----
<b>SUBSTRING</b> (FIRSTNAME,1,2, <b>CODEUNITS32</b> )	'Jü' -- x'4AC3BC'
<b>SUBSTRING</b> (FIRSTNAME,1,2, <b>CODEUNITS16</b> )	'Jü' -- x'4AC3BC'
<b>SUBSTRING</b> (FIRSTNAME,1,2, <b>OCTETS</b> )	'J ' -- x'4A20' (a truncated string)
<b>SUBSTRING</b> (FIRSTNAME,8, <b>CODEUNITS16</b> )	a zero-length string
<b>SUBSTRING</b> (FIRSTNAME,8,4, <b>OCTETS</b> )	a zero-length string

- C1 is a VARCHAR(12) column in table T1. One of its values is the string 'ABCDEFG'. When C1 has this value:

Function ...	Returns ...
-----	-----
<b>SUBSTRING</b> (C1,-2,2, <b>OCTETS</b> )	a zero-length string
<b>SUBSTRING</b> (C1,-2,4, <b>OCTETS</b> )	'A'
<b>SUBSTRING</b> (C1,-2, <b>OCTETS</b> )	'ABCDEFG'
<b>SUBSTRING</b> (C1,0,1, <b>OCTETS</b> )	a zero-length string

- The following example illustrates how SUBSTRING replaces the bytes of leading or trailing partial multi-byte code points with blanks when the string length unit is OCTETS. Assume that UTF8\_VAR contains the UTF-8 representation of the Unicode string '&N~AB', where '&' is the musical symbol G clef and '~' is the combining tilde character.

```
SUBSTRING(UTF8_VAR, 2, 5, OCTETS)
```

Three blank bytes precede the 'N', and one blank byte follows the 'N'.

## TABLE\_NAME

```
►►—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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

### *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 bytes.

### *objectschema*

A character expression representing the schema used to qualify the supplied *objectname* value before resolution. *objectschema* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.

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.



catalog partition when using the TABLE\_SCHEMA and TABLE\_NAME scalar functions, the BASE\_TABLE table function can be used instead.

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—(—*expression*—)—————►◄

The schema is SYSIBM. (The SYSFUN version of the TAN function continues to be available.)

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 (except for DECFLOAT). It is converted to a double-precision floating-point number for processing by the function.

## TAN

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with DFT\_SQLMATHWARN set to YES; the result is the null value if the argument is null.

## TANH

►►—TANH—(*—expression—*)—————►►

The schema is SYSIBM.

Returns the hyperbolic tangent of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric data type (except for DECFLOAT). 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 can be null or the database is configured with DFT\_SQLMATHWARN set to YES; the result is the null value if the argument is null.

## 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 string representation of a time or timestamp that is not a CLOB, LONG VARCHAR, DBCLOB, or LONG VARGRAPHIC.

Only Unicode databases support an argument that is a graphic string representation of a time or a timestamp. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

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 string:
  - The result is the time represented by the string.

Example:

- Select all notes from the IN\_TRAY sample table that were received at least one hour later in the day (any day) than the current time.

```
SELECT * FROM IN_TRAY
WHERE TIME(RECEIVED) >= CURRENT TIME + 1 HOUR
```

## TIMESTAMP

►—TIMESTAMP—(—*expression*——*expression*—)—►

The schema is SYSIBM.

The TIMESTAMP function returns a timestamp from a value or a pair of values.

Only Unicode databases support an argument that is a graphic string representation of a date, a time, or a timestamp. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

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 string representation of a timestamp, or a string of length 14 that is not a CLOB, LONG VARCHAR, DBCLOB, or LONG VARGRAPHIC.
 

A string of length 14 must be a string of digits that represents a valid date and time in the form *yyyymmddhhmmss*, where *yyyy* is the year, *mm* 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 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 string:
  - The result is the timestamp represented by that string. If the argument is a 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\_FORMAT

►►—TIMESTAMP\_FORMAT—(—*string-expression*—,—*format-string*—)—————►►

The schema is SYSIBM.

The TIMESTAMP\_FORMAT function returns a timestamp that is based on the interpretation of the input string using the specified format.

*string-expression*

An expression that returns a value of the CHAR or VARCHAR data type, with a length attribute that is not greater than 254 (SQLSTATE 42815). The *string-expression* must contain the components of a timestamp that correspond to the format specified by *format-string*.

*format-string*

An expression that returns a value of a built-in character string data type, with a length attribute that is not greater than 254 bytes (SQLSTATE 22007). The *format-string* contains a template of how *string-expression* is interpreted and then converted to a timestamp value. The content of *format-string* can be specified in mixed case.

A valid *format-string* must contain at least one format element, must not contain multiple specifications for any component of a timestamp, and can contain any combination of the format elements, unless otherwise noted in Table 39 (SQLSTATE 22007). For example, *format-string* cannot contain both YY and YYYY, because they are both used to interpret the year component of *string-expression*. Refer to the table to determine which format elements cannot be specified together. Two format elements can optionally be separated by one or more of the following separator characters:

- dash (-)
- period (.)
- slash (/)
- comma (,)
- apostrophe (')
- semi-colon (;)
- colon (:)
- blank ( )

Separator characters can also be specified at the start or end of *format-string*. The separator characters in *format-string* do not have to exactly match the specifications in *expression* (the separators are ignored).

Table 39. Format elements for the TIMESTAMP\_FORMAT function

Format element	Related components of a timestamp	Description
DD	day	Day of month (01-31).
DDD	month, day	Day of year (001-366).



Table 39. Format elements for the TIMESTAMP\_FORMAT function (continued)

Format element	Related components of a timestamp	Description
FF[ <i>n</i> ]	microseconds	Fractional seconds (0-999999). The number <i>n</i> is used to specify the number of digits expected in the <i>string-expression</i> . Valid values for <i>n</i> are 1-6. The default is 6.
HH	hour	HH behaves the same as HH12.
HH12	hour	Hour of the day (01-12) in 12-hour format. AM is the default meridian indicator.
HH24	hour	Hour of the day (00-24) in 24-hour format.
J	year, month, and day	Julian day (number of days since January 1, 4713 BC).
MI	minute	Minute (00-59).
MM	month	Month (01-12).
NNNNNN	microseconds	Microseconds (000000-999999). Same as FF6.
RR	year	Last two digits of the adjusted year (00-99).
RRRR	year	4-digit adjusted year (0000-9999).
SS	seconds	Seconds (00-59).
SSSS	hours, minutes, and seconds	Seconds since previous midnight (00000-86400).
Y	year	Last digit of the year (0-9). First three digits of the current year are used to determine the full 4-digit year.
YY	year	Last two digits of the year (00-99). First two digits of the current year are used to determine the full 4-digit year.
YYY	year	Last three digits of the year (000-999). First digit of the current year is used to determine the full 4-digit year.
YYYY	year	4-digit year (0000-9999).

The RR and RRRR format elements can be used to alter how a specification for a year is to be interpreted by adjusting the value to produce a 2-digit value or a 4-digit value depending on the leftmost two digits of the current year according to the following table.

## TIMESTAMP\_FORMAT

Last two digits of the current year	Two-digit year in <i>string-expression</i>	First two digits of the year component of timestamp
0-50	0-49	First two digits of the current year
51-99	0-49	First two digits of the current year + 1
0-50	50-99	First two digits of the current year - 1
51-99	50-99	First two digits of the current year

For example, if the current year is 2007, '86' with format 'RR' means 1986, but if the current year is 2052, it means 2086.

The following defaults are used when a *format-string* does not include a format element for one of the following components of a timestamp:

Timestamp component	Default
<b>year</b>	current year
<b>month</b>	current month
<b>day</b>	01 (first day of the month)
<b>hour</b>	00
<b>minute</b>	00
<b>second</b>	00
<b>microsecond</b>	000000

Leading zeros can be specified for any component of the timestamp value (that is, month, day, hour, minutes, seconds) that does not have the maximum number of significant digits for the corresponding format element in the *format-string*.

A substring of the *string-expression* representing a component of a timestamp (such as year, month, day, hour, minutes, seconds) can include less than the maximum number of digits for that component of the timestamp indicated by the corresponding format element. Any missing digits default to zero. For example, with a *format-string* of 'YYYY-MM-DD HH24:MI:SS', an input value of '999-3-9 5:7:2' would produce the same result as '0999-03-09 05:07:02'.

The result of the function is a timestamp. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Notes:

- **Determinism:** The TIMESTAMP\_FORMAT function is generally defined as not deterministic. The function is deterministic only if all of the following are true:
  - The *format-string* is a constant
  - The *format-string* includes a format element that fully defines the year without having to use the current year (includes format element YYYY or J)
  - The *format-string* defines the month without having to use the current month (includes format element MM or J)
- **Syntax alternatives:** TO\_DATE and TO\_TIMESTAMP are synonyms for TIMESTAMP\_FORMAT.

Examples:

- Insert a row into the IN\_TRAY table with a receiving timestamp that is equal to one second before the beginning of the year 2000 (December 31, 1999 at 23:59:59).

```
INSERT INTO IN_TRAY (RECEIVED)
VALUES (TIMESTAMP_FORMAT('1999-12-31 23:59:59',
'YYYY-MM-DD HH24:MI:SS'))
```

- An application receives strings of date information into a variable called INDATEVAR. This value is not strictly formatted and might include two or four digits for years, and one or two digits for months and days. Date components might be separated with dash (-) or slash (/) characters and are expected to be in day, month, and year order. Time information consists of hours (in 24-hour format) and minutes, and is usually separated by a colon. Sample values include '15/12/98 13:48' and '9-3-2004 8:02'. Insert such values into the IN\_TRAY table.

```
INSERT INTO IN_TRAY (RECEIVED)
VALUES (TIMESTAMP_FORMAT(:INDATEVAR,
'DD/MM/RRRR HH24:MI'))
```

The use of RRRR in the format allows for 2- and 4-digit year values and assigns missing first two digits based on the current year. If YYYY were used, input values with a 2-digit year would have leading zeros. The slash separator also allows the dash character. Assuming a current year of 2007, resulting timestamps from the sample values are:

```
'15/12/98 13:48' --> 1998-12-15-13.48.00.000000
'9-3-2004 8:02' --> 2004-03-09-08.02.00.000000
```

## TIMESTAMP\_ISO

►►—TIMESTAMP\_ISO—(—*expression*—)—————►►

The schema is 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 the CURRENT DATE special register for the date elements, and zero for the fractional time element.

The argument must be a date, time, or timestamp, or a valid character string representation of a date, time or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is 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.

## TIMESTAMPDIFF

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 and converting the result to CHAR(22). In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

The following assumptions may be used in estimating a difference:

- There are 365 days in a year.
- There are 30 days in a month.
- There are 24 hours in a day.
- There are 60 minutes in an hour.
- There are 60 seconds in a minute.

These assumptions are used when converting the information in the second argument, 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 the difference between '1997-03-01-00.00.00' and '1997-02-01-00.00.00', the result is 30. This is because the difference between the timestamps is 1 month, and the assumption of 30 days in a month applies.

Example:

The following example returns 4277, the number of minutes between two timestamps:

```
TIMESTAMPDIFF(4, CHAR(TIMESTAMP('2001-09-29-11.25.42.483219') -
TIMESTAMP('2001-09-26-12.07.58.065497')))
```

## TO\_CHAR

►► TO\_CHAR—(*timestamp-expression*—, *format-string*—)—————►►

The schema is SYSIBM.

The TO\_CHAR function returns a character representation of a timestamp that has been formatted using a character template.

TO\_CHAR is a synonym for VARCHAR\_FORMAT.

## TO\_DATE

►►—TO\_DATE—(*—string-expression—*,*format-string—*)—►►

The schema is SYSIBM.

The TO\_DATE function returns a timestamp from a character string that has been interpreted using a character template.

TO\_DATE is a synonym for TIMESTAMP\_FORMAT.

## TOTALORDER

►►—TOTALORDER—(*—decfloat-expression1—*,*—decfloat-expression2—*)—►►

The schema is SYSIBM.

The TOTALORDER function returns a SMALLINT value of -1, 0, or 1 that indicates the comparison order of two arguments.

*decfloat-expression1*

An expression that returns a value of any built-in numeric data type. If the argument is not DECFLOAT(34), it is logically converted to DECFLOAT(34) for processing.

*decfloat-expression2*

An expression that returns a value of any built-in numeric data type. If the argument is not a decimal floating-point value, it is converted to DECFLOAT(34) for processing.

Numeric comparison is exact, and the result is determined for finite operands as if range and precision were unlimited. An overflow or underflow condition cannot occur.

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

The semantics of the TOTALORDER function are based on the total order predicate rules of IEEE 754R. TOTALORDER returns the following values:

- -1 if *decfloat-expression1* is lower in order compared to *decfloat-expression2*
- 0 if both *decfloat-expression1* and *decfloat-expression2* have the same order
- 1 if *decfloat-expression1* is higher in order compared to *decfloat-expression2*

The ordering of the special values and finite numbers is as follows:

-NAN<-SNAN<-INFINITY<-0.10<-0.100<-0<0<0.100<0.10<INFINITY<SNAN<NAN

The result of the function is a SMALLINT value. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Examples:

## TOTALORDER

- The following examples show the use of the TOTALORDER function to compare decimal floating point values:

```
TOTALORDER(-INFINITY, -INFINITY) = 0
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-1.0)) = 0
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-1.00)) = -1
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-0.5)) = -1
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(0.5)) = -1
TOTALORDER(DECFLOAT(-1.0), INFINITY) = -1
TOTALORDER(DECFLOAT(-1.0), SNAN) = -1
TOTALORDER(DECFLOAT(-1.0), NAN) = -1
TOTALORDER(NAN, DECFLOAT(-1.0)) = 1
TOTALORDER(-NAN, -NAN) = 0
TOTALORDER(-SNAN, -SNAN) = 0
TOTALORDER(NAN, NAN) = 0
TOTALORDER(SNAN, SNAN) = 0
TOTALORDER(-1.0, -1.0) = 0
TOTALORDER(-1.0, -1.00) = -1
TOTALORDER(-1.0, -0.5) = -1
TOTALORDER(-1.0, 0.5) = -1
TOTALORDER(-1.0, INFINITY) = -1
TOTALORDER(-1.0, SNAN) = -1
TOTALORDER(-1.0, NAN) = -1
```

## TRANSLATE scalar function

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 might have been converted to other characters.

The function converts all the characters in *char-string-exp* or *graphic-string-exp* that also occur in *from-string-exp* to the corresponding characters in *to-string-exp* or, if no corresponding characters exist, to the pad character specified by *pad-char-exp*.

*char-string-exp* or *graphic-string-exp*

Specifies a string that is to be converted.

*to-string-exp*

Specifies a string of characters to which certain characters in *char-string-exp* will be converted.

If a value for *to-string-exp* is not specified, and the data type is not graphic, all characters in *char-string-exp* will be in monospace; that is, the characters a-z will be converted to the characters A-Z, and other characters will be converted to their uppercase equivalents, if they exist. For example, in code page 850, é maps to É, but ÿ is not mapped, because code page 850 does not include Ÿ. If the code point length of the result character is not the same as the code point length of the source character, the source character is not converted.

*from-string-exp*

Specifies a string of characters which, if found in *char-string-exp*, will be converted to the corresponding character in *to-string-exp*. If *from-string-exp* contains duplicate characters, the first one found will be used, and the duplicates will be ignored. If *to-string-exp* is longer than *from-string-exp*, the surplus characters will be ignored. If *to-string-exp* is specified, *from-string-exp* must also be specified.

*pad-char-exp*

Specifies a single character that will be used to pad *to-string-exp* if *to-string-exp* is shorter than *from-string-exp*. The *pad-char-exp* argument must have a length attribute of one. If a value is not specified, a single-byte blank character is assumed.

The arguments can be either character strings of data type CHAR or VARCHAR, or graphic strings of data type GRAPHIC or VARGRAPHIC. They cannot be of data type LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, or DBCLOB.

With *graphic-string-exp*, only *pad-char-exp* is optional (if a value is not specified, the double-byte blank character is assumed), and each argument, including the pad character, must be of a graphic data type.

The data type and code page of the result is the same as the data type and code page of the first argument. If the first argument is a host variable, the code page of the result is the database code page. Each argument other than the first argument is converted to the result code page unless it or the first argument is defined as FOR BIT DATA, in which case no conversion is done.

In a Unicode database where character and graphic are considered to be equivalent data types, there are the following exceptions:

- The code page of the result is 1208 if any argument but the first argument is FOR BIT DATA.
- The code page of the result is the code page that appears most often in the set of arguments, when no argument is FOR BIT DATA.
- The code page of the result is 1200 when two different code pages appear equally often in the set of arguments, when no argument is FOR BIT DATA.

The length attribute of the result is the same as that of the first argument. If any argument can be null, the result can be null. If any argument is null, the result is the null value.

If the arguments are of data type CHAR or VARCHAR, the corresponding characters in *to-string-exp* and *from-string-exp* must have the same number of bytes. For example, it is not valid to convert a single-byte character to a multi-byte character, or to convert a multi-byte character to a single-byte character. The *pad-char-exp* argument cannot be the first byte of a valid multi-byte character (SQLSTATE 42815).

## TRANSLATE scalar function

The characters are matched using a binary comparison. The database collation is not used.

If only *char-string-exp* is specified, single-byte characters will be monocased, and multi-byte characters will remain unchanged.

Examples:

- Assume that the host variable SITE (VARCHAR(30)) has a value of 'Hanauma Bay'.

```
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'.

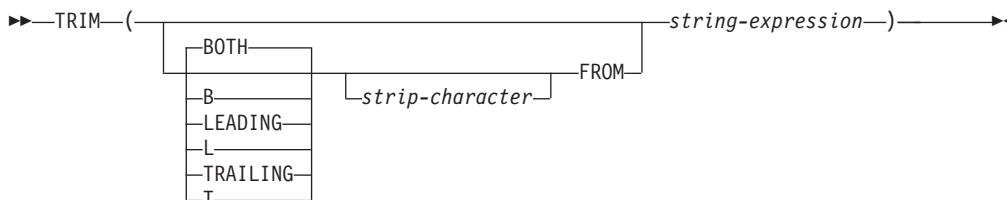
```
TRANSLATE(:SITE, 'bA', 'Bay', '%')
```

Returns the value 'HAnAumA bA%'.

```
TRANSLATE(:SITE, 'r', 'Bu')
```

Returns the value 'Hana ma ray'.

## TRIM



The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature.

The TRIM function removes blanks or occurrences of another specified character from the end or the beginning of a string expression.

### **BOTH, LEADING, or TRAILING**

Specifies whether characters are removed from the beginning, the end, or from both ends of the string expression. If this argument is not specified, the characters are removed from both the end and the beginning of the string.

### *strip-character*

A single-character constant that specifies the character that is to be removed. The *strip-character* can be any character whose UTF-32 encoding is a single character. The binary representation of the character is matched.

If *strip-character* is not specified and:

- If the *string-expression* is a DBCS graphic string, the default *strip-character* is a DBCS blank, whose code point is dependent on the database code page



- If the *string-expression* is a UCS-2 graphic string, the default *strip-character* is a UCS-2 blank (X'0020')
- Otherwise, the default *strip-character* is an SBCS blank (X'20')

**FROM** *string-expression*

An expression that returns a value that is a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type.

The result is a varying-length string with the same maximum length as the length attribute of the *string-expression*. The actual length of the result is the length of the *string-expression* minus the number of bytes that are removed. If all of the characters are removed, the result is an empty varying-length string. The code page of the result is the same as the code page of the *string-expression*.

Examples:

- Assume that the host variable HELLO of type CHAR(9) has a value of ' Hello'.

```
SELECT TRIM(:HELLO),
 TRIM(TRAILING FROM :HELLO)
FROM SYSIBM.SYSDUMMY1
```

returns the values 'Hello' and ' Hello', respectively.

- Assume that the host variable BALANCE of type CHAR(9) has a value of '000345.50'.

```
SELECT TRIM(L '0' FROM :BALANCE),
 FROM SYSIBM.SYSDUMMY1
```

returns the value '345.50'.

**TRUNCATE**

►►—TRUNCATE—(—*expression1*—,—*expression2*—)—————►►

The schema is SYSIBM. (The SYSFUN version of the TRUNCATE function continues to be available.)

Returns *expression1* truncated to *expression2* places to the right of the decimal point if *expression2* is positive, or to the left of the decimal point if *expression2* is zero or negative.

*expression1*

An expression that returns a value of any built-in numeric data type.

*expression2*

An expression that returns a small or a large integer. The absolute value of the integer specifies the number of places to the right of the decimal point for the result if *expression2* is not negative, or to left of the decimal point if *expression2* is negative.

If the absolute value of *expression2* is larger than the number of digits to the left of the decimal point, the result is 0. For example:

```
TRUNCATE(748.58,-4) = 0
```

The data type and length attribute of the result are the same as the data type and length attribute of the first argument.

## TRUNCATE

The result can be null if the argument can be null or if the argument is not a decimal floating-point number and the database is configured with `DFT_SQLMATHWARN` set to `YES`; the result is the null value if the argument is null.

`TRUNC` can be specified in place of `TRUNCATE`.

Examples:

- Using the `EMPLOYEE` table, calculate the average monthly salary for the highest paid employee. Truncate the result two places to the right of the decimal point.

```
SELECT TRUNCATE(MAX(SALARY)/12,2)
FROM EMPLOYEE;
```

Because the highest paid employee earns \$52750.00 per year, the example returns 4395.83.

- Display the number 873.726 truncated 2, 1, 0, -1, and -2 decimal places, respectively.

```
VALUES (
 TRUNCATE(873.726,2),
 TRUNCATE(873.726,1),
 TRUNCATE(873.726,0),
 TRUNCATE(873.726,-1),
 TRUNCATE(873.726,-2),
 TRUNCATE(873.726,-3));
```

This example returns 873.720, 873.700, 873.000, 870.000, 800.000, and 0.000.

- Display the decimal-floating point number 873.726 truncated 0 decimal places.

```
VALUES(TRUNCATE(DECFLOAT(873.726),0))
```

Returns the value 873.

## 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. (This function cannot be used as a source function when creating a user-defined function. Because it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.)

The data type of the result of the function is `INTEGER`. If *expression* can be null, the result can be null; if *expression* is null, the result is the null value.

The value returned by the `TYPE_ID` function is not portable across databases. The value may be different, even though the type schema and type name of the dynamic data type are the same. When coding for portability, use the `TYPE_SCHEMA` and `TYPE_NAME` functions to determine the type schema and type name.

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.

## 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. (This function cannot be used as a source function when creating a user-defined function. Because it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.)

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.

## 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. This function cannot be used as a source function when creating a user-defined function. Because it accepts



The schema is SYSIBM.

The UPPER function returns a string in which all characters have been converted to uppercase characters using the rules associated with the specified locale.

*string-expression*

An expression that returns a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC string. If *string-expression* is CHAR or VARCHAR, the expression must not be FOR BIT DATA (SQLSTATE 42815).

*locale-name*

A character constant that specifies the locale that defines the rules for conversion to uppercase characters. The value of *locale-name* is not case sensitive and must be a valid locale (SQLSTATE 42815). For information on valid locales and their naming, see “Locale names for SQL and XQuery”.

*code-units*

An integer constant that specifies the number of code units in the result. If specified, *code-units* must be an integer between 1 and 32 672 if the result is character data, or between 1 and 16 336 if the result is graphic data (SQLSTATE 42815). If *code-units* is not explicitly specified, it is implicitly the length attribute of *string-expression*. If OCTETS is specified and the result is graphic data, the value of *code-units* must be even (SQLSTATE 428GC).

**CODEUNITS16, CODEUNITS32, or OCTETS**

Specifies the string unit of *code-units*.

CODEUNITS16 specifies that *code-units* is expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *code-units* is expressed in 32-bit UTF-32 code units. OCTETS specifies that *code-units* is expressed in bytes.

If a string unit is not explicitly specified, the data type of the result determines the unit that is used. If the result is graphic data, *code-units* is expressed in two-byte units; otherwise, it is expressed in bytes. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see “String units in built-in functions” in “Character strings”.

The result of the function is VARCHAR if *string-expression* is CHAR or VARCHAR, and VARGRAPHIC if *string-expression* is GRAPHIC or VARGRAPHIC.

The length attribute of the result is determined by the implicit or explicit value of *code-units*, the implicit or explicit string unit, and the result data type, as shown in the following table:

Table 40. Length attribute of the result of UPPER as a function of string unit and result type

String unit	Character result type	Graphic result type
CODEUNITS16	MIN( <i>code-units</i> * 3, 32768)	<i>code-units</i>
CODEUNITS32	MIN( <i>code-units</i> * 4, 32768)	MIN( <i>code-units</i> * 2, 16336)
OCTETS	<i>code-units</i>	MIN( <i>code-units</i> / 2, 16336)

The actual length of the result might be greater than the length of *string-expression*. If the actual length of the result is greater than the length attribute of the result, an error is returned (SQLSTATE 42815). If the number of code units in the result exceeds the value of *code-units*, an error is returned (SQLSTATE 42815).

If *string-expression* is not in UTF-16, this function performs code page conversion of *string-expression* to UTF-16, and of the result from UTF-16 to the code page of

## UPPER (Locale sensitive)

*string-expression*. If either code page conversion results in at least one substitution character, the result includes the substitution character, a warning is returned (SQLSTATE 01517), and the warning flag SQLWARN8 in the SQLCA is set to 'W'.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples:

- Ensure that the characters in the value of column JOB in the EMPLOYEE table are returned in uppercase characters.

```
SELECT UPPER(JOB, 'en_US')
FROM EMPLOYEE
WHERE EMPNO = '000020'
```

The result is the value 'MANAGER'.

- Find the uppercase characters for all the 'İ' characters in a Turkish string.

```
VALUES UPPER('İİİİ', 'tr_TR', CODEUNITS16)
```

The result is the string 'IIII'.

- Find the uppercase form of the German 'ß' (sharp S).

```
VALUES UPPER('ß', 'de', 2, CODEUNITS16)
```

The result is the string 'SS'. Note that *code-units* must be specified in this example, because there are more code units in the result than in *string-expression*.

## VALUE

►►VALUE(—*expression*—, *expression*—)

The schema is SYSIBM.

The VALUE function returns the first argument that is not null.

VALUE is a synonym for COALESCE.

## VARCHAR

**Character to Varchar:**

►►VARCHAR(—*character-expression*—, —*integer*—)

**Graphic to Varchar:**

►►VARCHAR(—*graphic-expression*—, —*integer*—)

**Datetime to Varchar:**

►►—VARCHAR—(—*datetime-expression*—)—————►►

The schema is SYSIBM.

The VARCHAR function returns a varying-length character string representation of:

- A character string, if the first argument is any type of character string
- A graphic string (Unicode databases only), if the first argument is any type of graphic string
- A datetime value, if the argument is a date, time, or timestamp

In a Unicode database, when the output string is truncated part-way through a multiple-byte character:

- If the input was a character string, the partial character is replaced with one or more blanks
- If the input was a graphic string, the partial character is replaced by the empty string

Do not rely on either of these behaviors, because they might change in a future release.

The result of the function is a varying-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.

**Character to Varchar**

*character-expression*

An expression whose value must be of a character string data type with a maximum length of 32 672 bytes.

*integer*

The length attribute of the resulting varying-length character string. The value must be between 0 and 32 672. If this argument is not specified, the length attribute of the result is the same as the length attribute of the argument.

**Graphic to Varchar**

*graphic-expression*

An expression whose value must be of a graphic string data type other than LONG VARGRAPHIC or DBCLOB, and whose maximum length is 16 336 double-byte characters.

*integer*

The length attribute of the resulting varying-length character string. The value must be between 0 and 32 672. If this argument is not specified, the length attribute of the result is the same as the length attribute of the argument.

**Datetime to Varchar**

*datetime-expression*

An expression whose value must be of the DATE, TIME, or TIMESTAMP data type.

Example:

## VARCHAR

- Set the host variable JOB\_DESC, defined as VARCHAR(8), to the VARCHAR equivalent of the job description (which is the value of the JOB column), defined as CHAR(8), for employee Dolores Quintana.

```
SELECT VARCHAR(JOB)
INTO :JOB_DESC
FROM EMPLOYEE
WHERE LASTNAME = 'QUINTANA'
```

## VARCHAR\_BIT\_FORMAT

►—VARCHAR\_BIT\_FORMAT—(—*character-expression*—,—*format-string*—)—————►

The schema is SYSIBM.

The VARCHAR\_BIT\_FORMAT function returns a bit string representation of a character string that has been formatted using a character template. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

### *character-expression*

An expression that returns a value that is a built-in character string that is not a CLOB (SQLSTATE 42815). The required length is determined by the specified format string and how the value is interpreted.

### *format-string*

A character constant that contains a template for how the value for *character-expression* is to be interpreted.

Valid format strings include: 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx' and 'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX' (SQLSTATE 42815) where each 'x' or 'X' corresponds to one hexadecimal digit in the result.

The result of the function is a varying-length character string FOR BIT DATA with the length attribute and actual length based on the format string. For the two valid format strings listed above, the length attribute is 36 and the actual length is 16 bytes. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

### Examples

- Represent a Universal Unique Identifier in its binary form:

```
VARCHAR_BIT_FORMAT('d83d6360-1818-11db-9804-b622a1ef5492',
'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx')
```

Result returned:

```
x'D83D6360181811DB9804B622A1EF5492'
```

- Represent a Universal Unique Identifier in its binary form:

```
VARCHAR_BIT_FORMAT('D83D6360-1818-11DB-9804-B622A1EF5492',
'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX')
```

Result returned:

```
x'D83D6360181811DB9804B622A1EF5492'
```



## VARCHAR\_FORMAT

►►—VARCHAR\_FORMAT—(—*timestamp-expression*—,—*format-string*—)—————►►

The schema is SYSIBM.

The VARCHAR\_FORMAT function returns a string representation of a timestamp that has been formatted according to a specified character template.

### *timestamp-expression*

An expression that returns a value of a built-in timestamp or a built-in character string data type whose length attribute is not greater than 254. The returned value contains a valid string representation of a timestamp (SQLSTATE 42815 or SQLSTATE 22007). For the valid formats of string representations of timestamps, see “String representations of datetime values” on page 79. If *timestamp-expression* is a character string, leading and trailing blanks are removed prior to converting the value to a timestamp.

### *format-string*

An expression that returns a value of a built-in character string data type, with a length attribute that is not greater than 254 bytes (SQLSTATE 22007). The *format-string* contains a template of how *timestamp-expression* is to be formatted.

A valid *format-string* must contain a combination of the format elements listed below (SQLSTATE 22007). Two format elements can optionally be separated by one or more of the following separator characters:

- dash (-)
- period (.)
- slash (/)
- comma (,)
- apostrophe (')
- semi-colon (;)
- colon (:)
- blank ( )

Separator characters can also be specified at the start or end of *format-string*.

Table 41. Format elements for the VARCHAR\_FORMAT function

Format element	Description
CC	Century (01-99). If the last two digits of the four-digit year are zero, the result is the first two digits of the year. Otherwise, the result is the first two digits of the year plus one.
DD	Day of month (01-31).
DDD	Day of year (001-366).
FF[ <i>n</i> ]	Fractional seconds (0-999999). The number <i>n</i> is used to specify the number of digits to include in the returned value. Valid values for <i>n</i> are 1-6. The default is 6.
HH	HH behaves the same as HH12.
HH12	Hour of the day (01-12) in 12-hour format. AM is the default meridian indicator.

Table 41. Format elements for the VARCHAR\_FORMAT function (continued)

Format element	Description
HH24	Hour of the day (00-24) in 24-hour format.
IW	ISO week of the year (01-53). The week starts on Monday and includes seven days. Week 1 is the first week of the year to contain a Thursday, which is equivalent to the first week of the year to contain January 4.
I	ISO year (0-9). The last digit of the year based on the ISO week that is returned.
IY	ISO year (00-99). The last two digits of the year based on the ISO week that is returned.
IYY	ISO year (000-999). The last three digits of the year based on the ISO week that is returned.
YYYY	ISO year (0000-9999). The 4-digit year based on the ISO week that is returned.
J	Julian day (number of days since January 1, 4713 BC).
MI	Minute (00-59).
MM	Month (01-12).
NNNNNN	Microseconds (000000-999999). Same as FF6.
Q	Quarter (1-4), where the months January through March return 1.
RR	RR behaves the same as YY.
RRRR	RRRR behaves the same as YYYY.
SS	Seconds (00-59).
SSSSS	Seconds since previous midnight (00000-86400).
W	Week of the month (1-5), where week 1 starts on the first day of the month and ends on the seventh day.
WW	Week of the year (01-53), where week 1 starts on January 1 and ends on January 7.
Y	Last digit of the year (0-9).
YY	Last two digits of the year (00-99).
YYY	Last three digits of the year (000-999).
YYYY	4-digit year (0000-9999).

The result is a representation of *timestamp-expression* in the format specified by *format-string*. The *format-string* is interpreted as a series of format elements that can optionally be separated by one or more separator characters. A string of characters in *format-string* is interpreted as the longest matching format element in Table 41 on page 393. If two format elements containing the same characters are not delimited by a separator character, the specification is interpreted, starting from the left, as the longest matching format element in the table, and continues until matches are found for the remainder of the format string. For example, 'YYYYYYDD' is interpreted as the format elements 'YYYY', 'YY', and 'DD'.

The result is a varying-length character string. The length attribute is the greater of 100 and the length attribute of the *format-string*. The *format-string* also determines the length attribute and the actual length of the result. The resulting string must not exceed the length attribute of the result (SQLSTATE 22007) . If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Notes:

- *Syntax alternatives:* TO\_CHAR is a synonym for VARCHAR\_FORMAT.

Example:

- Display the table names and creation timestamps for all of the system tables whose name starts with 'SYSU'.

```
SELECT VARCHAR(TABNAME, 20) AS TABLE_NAME,
 VARCHAR_FORMAT(CREATE_TIME, 'YYYY-MM-DD HH24:MI:SS')
 AS CREATION_TIME
FROM SYSCAT.TABLES
WHERE TABNAME LIKE 'SYSU%'
```

This example returns the following:

TABLE_NAME	CREATION_TIME
SYSUSERAUTH	2000-05-19 08:18:56
SYSUSEROPTIONS	2000-05-19 08:18:56

- Assume that the variable TMSTAMP has the following value: 2007-03-09-14.07.38.123456. The following examples show several invocations of the function and the resulting string values. The result type in each case is VARCHAR(100).

Function invocation	Result
<code>VARCHAR_FORMAT(TMSTAMP, 'YYYYMMDDHHMISSFF3')</code>	20070309020738123
<code>VARCHAR_FORMAT(TMSTAMP, 'YYYYMMDDHH24MISS')</code>	20070309140738
<code>VARCHAR_FORMAT(TMSTAMP, 'YYYYMMDDHHMI')</code>	200703090207
<code>VARCHAR_FORMAT(TMSTAMP, 'DD/MM/YY')</code>	09/03/07
<code>VARCHAR_FORMAT(TMSTAMP, 'MM-DD-YYYY')</code>	03-09-2007
<code>VARCHAR_FORMAT(TMSTAMP, 'J')</code>	2454169
<code>VARCHAR_FORMAT(TMSTAMP, 'Q')</code>	1
<code>VARCHAR_FORMAT(TMSTAMP, 'W')</code>	2
<code>VARCHAR_FORMAT(TMSTAMP, 'IW')</code>	10
<code>VARCHAR_FORMAT(TMSTAMP, 'WW')</code>	10

## VARCHAR\_FORMAT\_BIT

►►—VARCHAR\_FORMAT\_BIT—(—*bit-data-expression*—,—*format-string*—)—————►►

The schema is SYSIBM.

The VARCHAR\_FORMAT\_BIT function returns a character representation of a bit string that has been formatted using a character template.

*bit-data-expression*

An expression that returns a value that is a built-in character-string FOR BIT DATA data type (SQLSTATE 42815). The required length is determined by the specified format string and how the value is interpreted.

## VARCHAR\_FORMAT\_BIT

### *format-string*

A character constant that contains a template for how the result is to be formatted.

Valid format strings include: 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx' and 'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX' (SQLSTATE 42815) where each 'x' or 'X' corresponds to one hexadecimal digit from *bit-data-expression*.

The result of the function is a varying-length character string with the length attribute and actual length based on the format string. For the two valid format strings listed above, the length attribute is 36 and the actual length is 36 bytes. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

### Examples

- Represent a Universal Unique Identifier in its formatted form:

```
VARCHAR_FORMAT_BIT(cast(x'd83d6360181811db9804b622a1ef5492'
 as varchar(16) for bit data),
 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx')
```

Result returned:

```
'd83d6360-1818-11db-9804-b622a1ef5492'
```

- Represent a Universal Unique Identifier in its formatted form:

```
VARCHAR_FORMAT_BIT(cast(x'd83d6360181811db9804b622a1ef5492'
 as char(16) for bit data),
 'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX')
```

Result returned:

```
'D83D6360-1818-11DB-9804-B622A1EF5492'
```

## VARGRAPHIC

### Graphic to Vargraphic:

```
►►—VARGRAPHIC—(—graphic-expression—
 |
 |,—integer—|)—————►►
```

### Character to Vargraphic:

```
►►—VARGRAPHIC—(—character-expression—)—————►►
```

### Datetime to Vargraphic:

```
►►—VARGRAPHIC—(—datetime-expression—)—————►►
```

The schema is SYSIBM.

The VARGRAPHIC function returns a varying-length graphic string representation of:

- A graphic string, if the first argument is any type of graphic string
- A character string, converting single-byte characters to double-byte characters, if the first argument is any type of character string

- A datetime value (Unicode databases only), if the argument is a date, time, or timestamp

In a Unicode database, if a supplied argument is a character string, it is first converted to a graphic string before the function is executed. When the output string is truncated, such that the last character is a high surrogate, that surrogate is converted to the blank character (X'0020'). Do not rely on this behavior, because it might change in a future release.

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.

### Graphic to Vargraphic

*graphic-expression*

An expression that returns a value that is a graphic string.

*integer*

An integer value specifying the length attribute of the resulting VARGRAPHIC data type. The value must be between 0 and 16 336. If a value is not specified, the length attribute of the result is the same as the length attribute of the first argument.

If the length of the graphic expression is greater than the length attribute of the result, the result is truncated. A warning is returned (SQLSTATE 01004), unless the truncated characters were all blanks, and the graphic expression was not a long string (LONG VARGRAPHIC or DBCLOB).

### Character to Vargraphic

*character-expression*

An expression whose value must be of a character string data type other than LONG VARCHAR or CLOB, and whose maximum length is 16 336 bytes.

The length attribute of the result is equal to the length attribute of the argument.

Each single-byte character in *character-expression* is converted to its equivalent double-byte representation or to the double-byte substitution character in the result. Each double-byte character in *character-expression* is mapped without additional conversion. If the first byte of a double-byte character appears as the last byte of *character-expression*, it is converted to the double-byte substitution character. The sequential order of the characters in *character-expression* is preserved.

For a Unicode database, this function converts the character string from the code page of the operand to UCS-2. Every character of the operand, including double-byte characters, is converted. If a value for the second argument is provided, it specifies the required length of the resulting string (in UCS-2 characters).

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. All other characters are converted to their corresponding double-byte equivalents. If there is no corresponding double-byte equivalent, the double-byte substitution character for the code page is used.

## VARGRAPHIC

No warning or error code is generated if one or more double-byte substitution characters are returned in the result.

### Datetime to Vargraphic

*datetime-expression*

An expression whose value must be of the DATE, TIME, or TIMESTAMP data type.

## WEEK

►► WEEK(*expression*) ◀◀

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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

## WEEK\_ISO

►► WEEK\_ISO(*expression*) ◀◀

The schema is SYSFUN.

Returns the week of the year of the argument as an integer value in the range 1-53. The week starts with Monday and always includes 7 days. Week 1 is the first week of the year to contain a Thursday, which is equivalent to the first week containing January 4. It is therefore possible to have up to 3 days at the beginning of a year appear in the last week of the previous year. Conversely, up to 3 days at the end of a year may appear in the first week of the next year.

The 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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

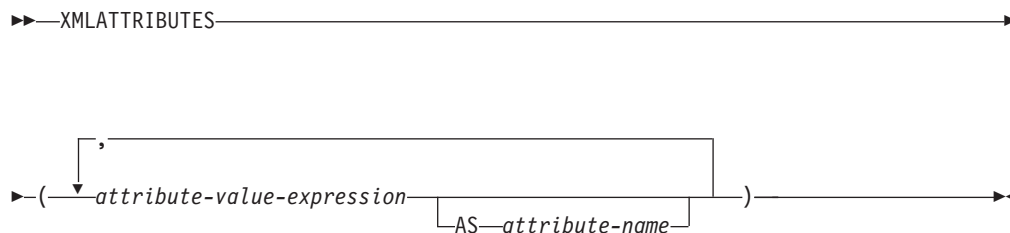
Example:

The following list shows examples of the result of WEEK\_ISO and DAYOFWEEK\_ISO.

DATE	WEEK_ISO	DAYOFWEEK_ISO
1997-12-28	52	7
1997-12-31	1	3
1998-01-01	1	4
1999-01-01	53	5

1999-01-04	1	1
1999-12-31	52	5
2000-01-01	52	6
2000-01-03	1	1

## XMLATTRIBUTES



The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLATTRIBUTES function constructs XML attributes from the arguments. This function can only be used as an argument of the XMLELEMENT function. The result is an XML sequence containing an XQuery attribute node for each non-null input value.

### *attribute-value-expression*

An expression whose result is the attribute value. The data type of *attribute-value-expression* cannot be a structured type (SQLSTATE 42884). The expression can be any SQL expression. If the expression is not a simple column reference, an attribute name must be specified.

### *attribute-name*

Specifies an attribute name. The name is an SQL identifier that must be in the form of an XML qualified name, or QName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. The attribute name cannot be `xmlns` or prefixed with `xmlns:`. A namespace is declared using the function XMLNAMESPACES. Duplicate attribute names, whether implicit or explicit, are not allowed (SQLSTATE 42713).

If *attribute-name* is not specified, *attribute-value-expression* must be a column name (SQLSTATE 42703). The attribute name is created from the column name using the fully escaped mapping from a column name to an XML attribute name.

The data type of the result is XML. If the result of *attribute-value-expression* can be null, the result can be null; if the result of every *attribute-value-expression* is null, the result is the null value.

### Note:

1. **Support in multiple database partition databases:** The BLOB data type and character string data defined as FOR BIT DATA are not supported (SQLSTATE 42884).

Examples:

**Note:** XMLATTRIBUTES does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- Produce an element with attributes.

## XMLATTRIBUTES

```
SELECT E.EMPNO, XMLELEMENT(
 NAME "Emp",
 XMLATTRIBUTES(
 E.EMPNO, E.FIRSTNME || ' ' || E.LASTNAME AS "name"
)
)
AS "Result"
FROM EMPLOYEE E WHERE E.EDLEVEL = 12
```

This query produces the following result:

```
EMPNO Result
000290 <Emp EMPNO="000290" name="JOHN PARKER"></Emp>
000310 <Emp EMPNO="000310" name="MAUDE SETRIGHT"></Emp>
200310 <Emp EMPNO="200310" name="MICHELLE SPRINGER"></Emp>
```

- Produce an element with a namespace declaration that is not used in any QName. The prefix is used in an attribute value.

```
VALUES XMLELEMENT(
 NAME "size",
 XMLNAMESPACES(
 'http://www.w3.org/2001/XMLSchema-instance' AS "xsi",
 'http://www.w3.org/2001/XMLSchema' AS "xsd"
),
 XMLATTRIBUTES(
 'xsd:string' AS "xsi:type"
), '1'
)
```

This query produces the following result:

```
<size xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xsi:type="xsd:string">1</size>
```

## XMLCOMMENT

►► XMLCOMMENT(—*string-expression*—)◄◄

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLCOMMENT function returns an XML value with a single XQuery comment node with the input argument as the content.

*string-expression*

An expression whose value has a character string type: CHAR, VARCHAR or CLOB. The result of the *string-expression* is parsed to check for conformance to the requirements for an XML comment, as specified in the XML 1.0 rule. The result of the *string-expression* must conform to the following regular expression:

```
((Char - '-') | ('-' (Char - '-')))*
```

where Char is defined as any Unicode character excluding surrogate blocks X'FFFE' and X'FFFF'. Basically, the XML comment cannot contain two adjacent hyphens, and cannot end with a hyphen (SQLSTATE 2200S).

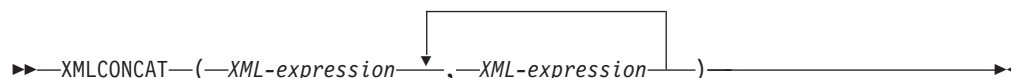
The data type of the result is XML. If the result of *string-expression* can be null, the result can be null; if the input value is null, the result is the null value.

**Note:**



1. **Support in multiple database partition databases:** XMLCOMMENT is not supported (SQLSTATE 42997).

## XMLCONCAT



The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLCONCAT function returns a sequence containing the concatenation of a variable number of XML input arguments.

*XML-expression*

Specifies an expression of data type XML.

The data type of the result is XML. The result is an XML sequence containing the concatenation of the non-null input XML values. Null values in the input are ignored. If the result of any *XML-expression* can be null, the result can be null; if the result of every input value is null, the result is the null value.

### Note:

1. **Support in multiple database partition databases:** The result, at the outer level of XML function nesting, must be an argument of the XMLSERIALIZE function (SQLSTATE 42997).

Example:

**Note:** XMLCONCAT does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- Construct a department element for departments A00 and B01, containing a list of employees sorted by first name. Include an introductory comment immediately preceding the department element.

```

SELECT XMLCONCAT(
 XMLCOMMENT(
 'Confirm these employees are on track for their product schedule'
),
 XMLELEMENT(
 NAME "Department",
 XMLATTRIBUTES(
 E.WORKDEPT AS "name"
),
 XMLAGG(
 XMLELEMENT(
 NAME "emp", E.FIRSTNME
)
 ORDER BY E.FIRSTNME
)
)
)
FROM EMPLOYEE E
WHERE E.WORKDEPT IN ('A00', 'B01')
GROUP BY E.WORKDEPT

```

This query produces the following result:

## XMLCONCAT

```
<!--Confirm these employees are on track for their product schedule-->
<Department name="A00">
<emp>CHRISTINE</emp>
<emp>DIAN</emp>
<emp>GREG</emp>
<emp>SEAN</emp>
<emp>VINCENZO</emp>
</Department>
<!--Confirm these employees are on track for their product schedule-->
<Department name="B01">
<emp>MICHAEL</emp>
</Department>
```

## XMLDOCUMENT

►► XMLDOCUMENT ( *XML-expression* ) ◀◀

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLDOCUMENT function returns an XML value with a single XQuery document node with zero or more children nodes.

### *XML-expression*

An expression that returns an XML value. A sequence item in the XML value must not be an attribute node (SQLSTATE 10507).

The data type of the result is XML. If the result of *XML-expression* can be null, the result can be null; if the input value is null, the result is the null value.

The children of the resulting document node are constructed as described in the following steps. The input expression is a sequence of nodes or atomic values, which is referred to in these steps as the content sequence.

1. If the content sequence contains a document node, the document node is replaced in the content sequence by the children of the document node.
2. Each adjacent sequence of one or more atomic values in the content sequence are replaced with a text node containing the result of casting each atomic value to a string with a single blank character inserted between adjacent values.
3. For each node in the content sequence, a new deep copy of the node is constructed. A deep copy of a node is a copy of the whole subtree rooted at that node, including the node itself and its descendants. Each copied node has a new node identity. Copied element and attribute nodes preserve their type annotation.
4. The nodes in the content sequence become the children of the new document node.

The XMLDOCUMENT function effectively executes the XQuery computed document constructor. The result of

```
XMLQUERY('document {$E}' PASSING BY REF XML-expression AS "E")
```

is equivalent to

```
XMLDOCUMENT(XML-expression)
```

with the exception of the case where *XML-expression* is null and XMLQUERY returns the empty sequence compared to XMLDOCUMENT which returns the null value.

**Note:**

1. **Support in multiple database partition databases:** XMLDOCUMENT is not supported (SQLSTATE 42997).

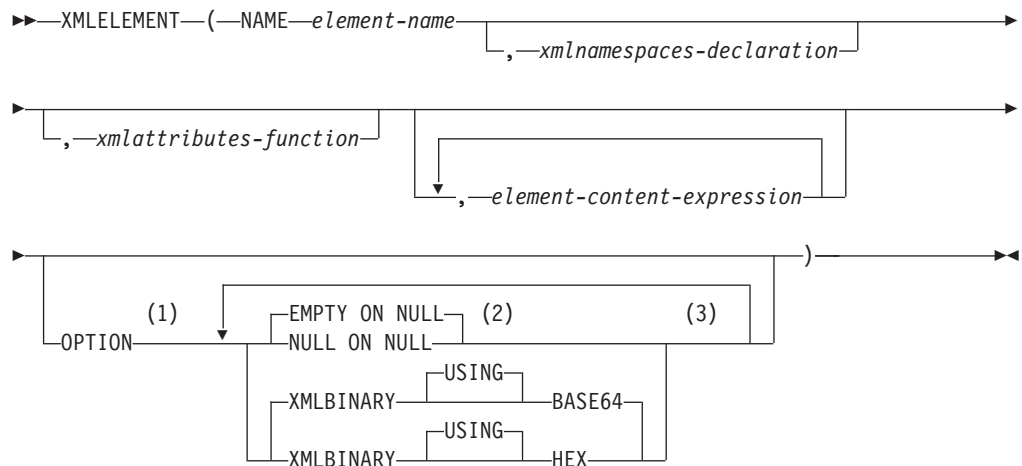
**Example:**

- Insert a constructed document into an XML column.

```

INSERT INTO T1 VALUES(
 123, (
 SELECT XMLDOCUMENT(
 XMLELEMENT(
 NAME "Emp", E.FIRSTNAME || ' ' || E.LASTNAME, XMLCOMMENT(
 'This is just a simple example'
)
)
)
)
 FROM EMPLOYEE E
 WHERE E.EMPNO = '000120'
)

```

**XMLLEMENT****Notes:**

- 1 The OPTION clause can only be specified if at least one *xmlattributes-function* or *element-content-expression* is specified.
- 2 NULL ON NULL or EMPTY ON NULL can only be specified if at least one *element-content-expression* is specified.
- 3 The same clause must not be specified more than once.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLLEMENT function returns an XML value that is an XQuery element node.

**NAME element-name**

Specifies the name of an XML element. The name is an SQL identifier that must be in the form of an XML qualified name, or QName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope (SQLSTATE 42635).

## XMLELEMENT

### *xmlnamespaces-declaration*

Specifies the XML namespace declarations that are the result of the XMLNAMESPACES declaration. The namespaces that are declared are in the scope of the XMLELEMENT function. The namespaces apply to any nested XML functions within the XMLELEMENT function, regardless of whether or not they appear inside another subselect.

If *xmlnamespaces-declaration* is not specified, namespace declarations are not associated with the constructed element.

### *xmlattributes-function*

Specifies the XML attributes for the element. The attributes are the result of the XMLATTRIBUTES function.

### *element-content-expression*

The content of the generated XML element node is specified by an expression or a list of expressions. The data type of *element-content-expression* cannot be a structured type (SQLSTATE 42884). The expression can be any SQL expression.

If *element-content-expression* is not specified, an empty string is used as the content for the element and OPTION NULL ON NULL or EMPTY ON NULL must not be specified.

## OPTION

Specifies additional options for constructing the XML element. If no OPTION clause is specified, the default is EMPTY ON NULL XMLBINARY USING BASE64. This clause has no impact on nested XMLELEMENT invocations specified in *element-content-expression*.

### EMPTY ON NULL or NULL ON NULL

Specifies whether a null value or an empty element is to be returned if the values of each *element-content-expression* is a null value. This option only affects null handling of element contents, not attribute values. The default is EMPTY ON NULL.

#### EMPTY ON NULL

If the value of each *element-content-expression* is null, an empty element is returned.

#### NULL ON NULL

If the value of each *element-content-expression* is null, a null value is returned.

### XMLBINARY USING BASE64 or XMLBINARY USING HEX

Specifies the assumed encoding of binary input data, character string data with the FOR BIT DATA attribute, or a distinct type that is based on one of these types. The encoding applies to element content or attribute values. The default is XMLBINARY USING BASE64.

#### XMLBINARY USING BASE64

Specifies that the assumed encoding is base64 characters, as defined for XML schema type xs:base64Binary encoding. The base64 encoding uses a 65-character subset of US-ASCII (10 digits, 26 lowercase characters, 26 uppercase characters, '+', and '/') to represent every six bits of the binary or bit data with one printable character in the subset. These characters are selected so that they are universally representable. Using this method, the size of the encoded data is 33 percent larger than the original binary or bit data.

#### XMLBINARY USING HEX

Specifies that the assumed encoding is hexadecimal characters, as

defined for XML schema type `xs:hexBinary` encoding. The hexadecimal encoding represents each byte (8 bits) with two hexadecimal characters. Using this method, the encoded data is twice the size of the original binary or bit data.

This function takes an element name, an optional collection of namespace declarations, an optional collection of attributes, and zero or more arguments that make up the content of the XML element. The result is an XML sequence containing an XML element node or the null value.

The data type of the result is XML. If any of the *element-content-expression* arguments can be null, the result can be null; if all the *element-content-expression* argument values are null and the NULL ON NULL option is in effect, the result is the null value.

**Note:**

1. **Support in multiple database partition databases:** The function is only supported as it was in Version 8. The result, at the outer level of XML value function nesting, must be an argument of the XMLSERIALIZE function. The null handling options and binary encoding options cannot be specified (SQLSTATE 42997). BLOB and character string data defined as FOR BIT DATA cannot be specified (SQLSTATE 42884).

When constructing elements that will be copied as content of another element that defines default namespaces, default namespaces should be explicitly undeclared in the copied element to avoid possible errors that could result from inheriting the default namespace from the new parent element. Predefined namespace prefixes ('`xs`', '`xsi`', '`xml`', and '`sqlxml`') must also be declared explicitly when they are used.

2. **Constructing an element node:** The resulting element node is constructed as follows:
  - a. The *xmlnamespaces-declaration* adds a set of in-scope namespaces for the constructed element. Each in-scope namespace associates a namespace prefix (or the default namespace) with a namespace URI. The in-scope namespaces define the set of namespace prefixes that are available for interpreting QNames within the scope of the element.
  - b. If the *xmlattributes-function* is specified, it is evaluated and the result is a sequence of attribute nodes.
  - c. Each *element-content-expression* is evaluated and the result is converted into a sequence of nodes as follows:
    - If the result type is not XML, it is converted to an XML text node whose content is the result of *element-content-expression* mapped to XML according to the rules of mapping SQL data values to XML data values (see the table that describes supported casts from non-XML values to XML values in "Casting between data types").
    - If the result type is XML, then in general the result is a sequence of items. Some of the items in that sequence might be document nodes. Each document node in the sequence is replaced by the sequence of its top-level children. Then for each node in the resulting sequence, a new deep copy of the node is constructed, including its children and attributes. Each copied node has a new node identity. Copied element and attribute nodes preserve their type annotation. For each adjacent sequence of one or more atomic values returned in the sequence, a new text node is constructed, containing the result of casting each atomic value to a string, with a single blank character inserted between adjacent

values. Adjacent text nodes in the content sequence are merged into a single text node by concatenating their contents, with no intervening blanks. After concatenation, any text node whose content is a zero-length string is deleted from the content sequence.

- d. The result sequence of XML attributes and the resulting sequences of all *element-content-expression* specifications are concatenated into one sequence which is called the content sequence. Any sequence of adjacent text nodes in the content sequence is merged into a single text node. If all the *element-content-expression* arguments are empty strings, or an *element-content-expression* argument is not specified, an empty element is returned.
  - e. The content sequence must not contain an attribute node following a node that is not an attribute node (SQLSTATE 10507). Attribute nodes occurring in the content sequence become attributes of the new element node. Two or more of these attribute nodes must not have the same name (SQLSTATE 10503). A namespace declaration is created corresponding to any namespace used in the names of the attribute nodes if the namespace URI is not in the in-scope namespaces of the constructed element.
  - f. Element, text, comment, and processing instruction nodes in the content sequence become the children of the constructed element node.
  - g. The constructed element node is given a type annotation of `xs:anyType`, and each of its attributes is given a type annotation of `xdt:untypedAtomic`. The node name of the constructed element node is `element-name` specified after the `NAME` keyword.
3. **Rules for using namespaces within XMLELEMENT:** Consider the following rules about scoping of namespaces:
- The namespaces declared in the `XMLNAMESPACES` declaration are the in-scope namespaces of the element node constructed by the `XMLELEMENT` function. If the element node is serialized, then each of its in-scope namespaces will be serialized as a namespace attribute unless it is an in-scope namespace of the parent of the element node and the parent element is serialized too.
  - If an `XMLQUERY` or `XMLEXISTS` is in an *element-content-expression*, then the namespaces becomes the statically known namespaces of the XQuery expression of the `XMLQUERY` or `XMLEXISTS`. Statically known namespaces are used to resolve the QNames in the XQuery expression. If the XQuery prolog declares a namespace with the same prefix, within the scope of the XQuery expression, the namespace declared in the prolog will override the namespaces declared in the `XMLNAMESPACES` declaration.
  - If an attribute of the constructed element comes from an *element-content-expression*, its namespace might not already be declared as an in-scope namespace of the constructed element, in this case, a new namespace is created for it. If this would result in a conflict, which means that the prefix of the attribute name is already bound to a different URI by a in-scope namespace, DB2 generates a prefix that does not cause such a conflict and the prefix used in the attribute name is changed to the new prefix, and a namespace is created for this new prefix. The generated new prefix follows the following pattern: `"db2ns-xx"`, where `"x"` is a character chosen from the set `[A-Z,a-z,0-9]`. For example:

```
VALUES XMLELEMENT(
 NAME "c", XMLQUERY(
 'declare namespace ipo="www.ipo.com"; $m/ipo:a/@ipo:b'
)
 PASSING XMLPARSE(
```

```

 DOCUMENT '<tst:a xmlns:tst="www.ipo.com" tst:b="2"/>'
) AS "m"
)
)

```

returns:

```
<c xmlns:tst="www.ipo.com" tst:b="2"/>
```

A second example:

```

VALUES XMLELEMENT(
 NAME "tst:c", XMLNAMESPACES(
 'www.tst.com' AS "tst"
),
 XMLQUERY(
 'declare namespace ipo="www.ipo.com"; $m/ipo:a/@ipo:b'
 PASSING XMLPARSE(
 DOCUMENT '<tst:a xmlns:tst="www.ipo.com" tst:b="2"/>'
) AS "m"
)
)
)

```

returns:

```
<tst:c xmlns:tst="www.tst.com" xmlns:db2ns-a1="www.ipo.com"
db2ns-a1:b="2"/>
```

Examples:

**Note:** XMLELEMENT does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- Construct an element with the NULL ON NULL option.

```

SELECT E.FIRSTNME, E.LASTNAME, XMLELEMENT(
 NAME "Emp", XMLELEMENT(
 NAME "firstname", E.FIRSTNME
),
 XMLELEMENT(
 NAME "lastname", E.LASTNAME
)
 OPTION NULL ON NULL
)
AS "Result"
FROM EMPLOYEE E
WHERE E.EDLEVEL = 12

```

This query produces the following result:

FIRSTNME	LASTNAME	Emp
JOHN	PARKER	<Emp><firstname>JOHN</firstname> <lastname>PARKER</lastname></Emp>
MAUDE	SETRIGHT	<Emp><firstname>MAUDE</firstname> <lastname>SETRIGHT</lastname></Emp>
MICHELLE	SPRINGER	<Emp><firstname>MICHELLE</firstname> <lastname>SPRINGER</lastname></Emp>

- Produce an element with a list of elements nested as child elements.

```

SELECT XMLELEMENT(
 NAME "Department", XMLATTRIBUTES(
 E.WORKDEPT AS "name"
),
 XMLAGG(
 XMLELEMENT(
 NAME "emp", E.FIRSTNME
)
)
 ORDER BY E.FIRSTNME

```

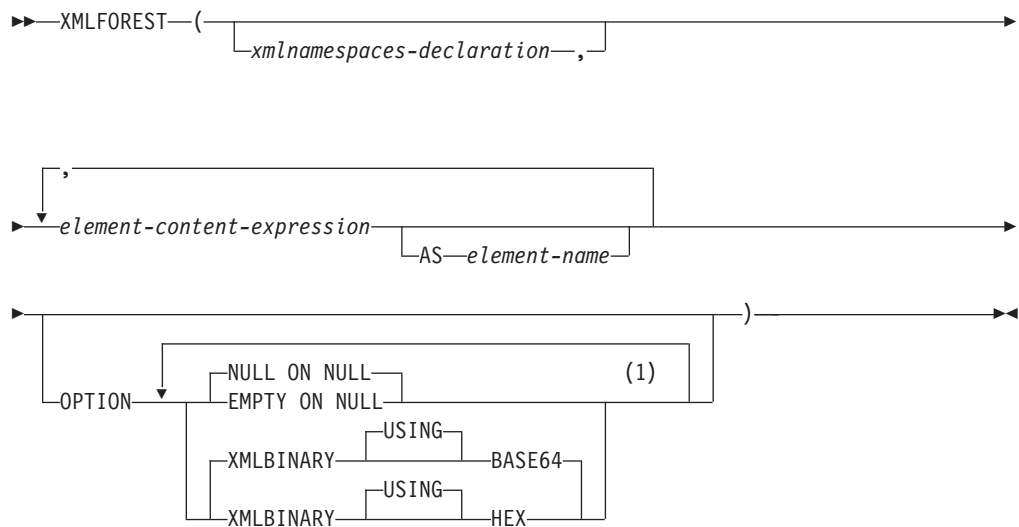
## XMLFOREST

```
)
)
AS "dept_list"
FROM EMPLOYEE E
WHERE E.WORKDEPT IN ('A00', 'B01')
GROUP BY WORKDEPT
```

This query produces the following result:

```
dept_list
<Department name="A00">
<emp>CHRISTINE</emp>
<emp>SEAN</emp>
<emp>VINCENZO</emp>
</Department>
<Department name="B01">
<emp>MICHAEL</emp>
</Department>
```

## XMLFOREST



### Notes:

- 1 The same clause must not be specified more than once.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLFOREST function returns an XML value that is a sequence of XQuery element nodes.

### *xmlnamespaces-declaration*

Specifies the XML namespace declarations that are the result of the XMLNAMESPACES declaration. The namespaces that are declared are in the scope of the XMLFOREST function. The namespaces apply to any nested XML functions within the XMLFOREST function, regardless of whether or not they appear inside another subselect.

If *xmlnamespaces-declaration* is not specified, namespace declarations are not associated with the constructed elements.



*element-content-expression*

The content of the generated XML element node is specified by an expression. The data type of *element-content-expression* cannot be a structured type (SQLSTATE 42884). The expression can be any SQL expression. If the expression is not a simple column reference, an element name must be specified.

**AS** *element-name*

Specifies the XML element name as an SQL identifier. The element name must be of the form of an XML qualified name, or QName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope (SQLSTATE 42635). If *element-name* is not specified, *element-content-expression* must be a column name (SQLSTATE 42703, SQLCODE -206). The element name is created from the column name using the fully escaped mapping from a column name to an QName.

**OPTION**

Specifies additional options for constructing the XML element. If no OPTION clause is specified, the default is NULL ON NULL XMLBINARY USING BASE64. This clause has no impact on nested XMLELEMENT invocations specified in *element-content-expression*.

**EMPTY ON NULL or NULL ON NULL**

Specifies whether a null value or an empty element is to be returned if the values of each *element-content-expression* is a null value. This option only affects null handling of element contents, not attribute values. The default is NULL ON NULL.

**EMPTY ON NULL**

If the value of each *element-content-expression* is null, an empty element is returned.

**NULL ON NULL**

If the value of each *element-content-expression* is null, a null value is returned.

**XMLBINARY USING BASE64 or XMLBINARY USING HEX**

Specifies the assumed encoding of binary input data, character string data with the FOR BIT DATA attribute, or a distinct type that is based on one of these types. The encoding applies to element content or attribute values. The default is XMLBINARY USING BASE64.

**XMLBINARY USING BASE64**

Specifies that the assumed encoding is base64 characters, as defined for XML schema type xs:base64Binary encoding. The base64 encoding uses a 65-character subset of US-ASCII (10 digits, 26 lowercase characters, 26 uppercase characters, '+', and '/') to represent every six bits of the binary or bit data with one printable character in the subset. These characters are selected so that they are universally representable. Using this method, the size of the encoded data is 33 percent larger than the original binary or bit data.

**XMLBINARY USING HEX**

Specifies that the assumed encoding is hexadecimal characters, as defined for XML schema type xs:hexBinary encoding. The hexadecimal encoding represents each byte (8 bits) with two hexadecimal characters. Using this method, the encoded data is twice the size of the original binary or bit data.

## XMLFOREST

This function takes an optional set of namespace declarations and one or more arguments that make up the name and element content for one or more element nodes. The result is an XML sequence containing a sequence of XQuery element nodes or the null value.

The data type of the result is XML. If any of the *element-content-expression* arguments can be null, the result can be null; if all the *element-content-expression* argument values are null and the NULL ON NULL option is in effect, the result is the null value.

The XMLFOREST function can be expressed by using XMLCONCAT and XMLELEMENT. For example, the following two expressions are semantically equivalent.

```
XMLFOREST(xmlnamespaces-declaration, arg1 AS name1, arg2 AS name2 ...)
XMLCONCAT(
 XMLELEMENT(
 NAME name1, xmlnamespaces-declaration, arg1
),
 XMLELEMENT(
 NAME name2, xmlnamespaces-declaration, arg2
)
 ...
)
```

### Note:

1. **Support in multiple database partition databases:** The function is only supported as it was in Version 8. The result, at the outer level of XML value function nesting, must be an argument of the XMLSERIALIZE function. The null handling options and binary encoding options cannot be specified (SQLSTATE 42997). BLOB and character string data defined as FOR BIT DATA cannot be specified (SQLSTATE 42884).

When constructing elements that will be copied as content of another element that defines default namespaces, default namespaces should be explicitly undeclared in the copied element to avoid possible errors that could result from inheriting the default namespace from the new parent element. Predefined namespace prefixes ('xs', 'xsi', 'xml', and 'sqlxml') must also be declared explicitly when they are used.

Example:

**Note:** XMLFOREST does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- Construct a forest of elements with a default namespace.

```
SELECT EMPNO,
 XMLFOREST(
 XMLNAMESPACES(
 DEFAULT 'http://hr.org', 'http://fed.gov' AS "d"
),
 LASTNAME, JOB AS "d:job"
)
AS "Result"
FROM EMPLOYEE
WHERE EDLEVEL = 12
```

This query produces the following result:

```
EMPNO Result
000290 <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">PARKER
</LASTNAME>
```

```

<d:job xmlns="http://hr.org" xmlns:d="http://fed.gov">OPERATOR</d:job>

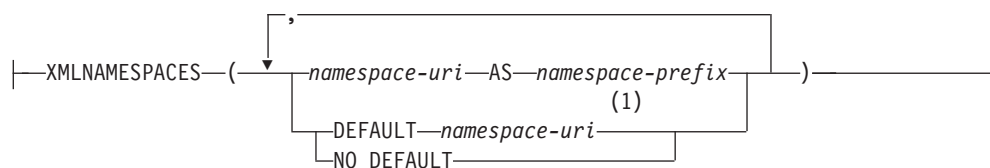
000310 <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">SETRIGHT
</LASTNAME>
<d:job xmlns="http://hr.org" xmlns:d="http://fed.gov">OPERATOR</d:job>

200310 <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">SPRINGER
</LASTNAME>
<d:job xmlns="http://hr.org" xmlns:d="http://fed.gov">OPERATOR</d:job>

```

## XMLNAMESPACES

### xmlns:declaration:



### Notes:

- 1 DEFAULT or NO DEFAULT can only be specified once in arguments of XMLNAMESPACES.

The schema is SYSIBM. The declaration name cannot be specified as a qualified name.

The XMLNAMESPACES declaration constructs namespace declarations from the arguments. This declaration can only be used as an argument for specific functions such as XMLELEMENT, XMLFOREST and XMLTABLE. The result is one or more XML namespace declarations containing in-scope namespaces for each non-null input value.

#### *namespace-uri*

Specifies the namespace universal resource identifier (URI) as an SQL character string constant. This character string constant must not be empty if it is used with a *namespace-prefix* (SQLSTATE 42815).

#### *namespace-prefix*

Specifies a namespace prefix. The prefix is an SQL identifier that must be in the form of an XML NCName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. The prefix cannot be `xml` or `xmlns` and the prefix must be unique within the list of namespace declarations (SQLSTATE 42635).

#### **DEFAULT** *namespace-uri*

Specifies the default namespace to use within the scope of this namespace declaration. The *namespace-uri* applies for unqualified names in the scope unless overridden in a nested scope by another DEFAULT declaration or a NO DEFAULT declaration.

#### **NO DEFAULT**

Specifies that no default namespace is to be used within the scope of this namespace declaration. There is no default namespace in the scope unless overridden in a nested scope by a DEFAULT declaration.

The data type of the result is XML. The result is an XML namespace declaration for each specified namespace. The result cannot be null.

## XMLNAMESPACES

Examples:

**Note:** XMLNAMESPACES does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- Produce an XML element named `adm:employee` and an XML attribute `adm:department`, both associated with a namespace whose prefix is `adm`.

```
SELECT EMPNO, XMLELEMENT(
 NAME "adm:employee", XMLNAMESPACES(
 'http://www.adm.com' AS "adm"
),
 XMLATTRIBUTES(
 WORKDEPT AS "adm:department"
),
 LASTNAME
)
FROM EMPLOYEE
WHERE JOB = 'ANALYST'
```

This query produces the following result:

```
000130 <adm:employee xmlns:adm="http://www.adm.com" adm:department="C01">
 QUINTANA</adm:employee>
000140 <adm:employee xmlns:adm="http://www.adm.com" adm:department="C01">
 NICHOLLS</adm:employee>
200140 <adm:employee xmlns:adm="http://www.adm.com" adm:department="C01">
 NATZ</adm:employee>
```

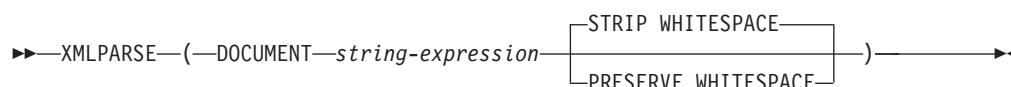
- Produce an XML element named `'employee'`, which is associated with a default namespace, and a sub-element named `'job'`, which does not use a default namespace, but whose sub-element named `'department'` does use a default namespace.

```
SELECT EMP.EMPNO, XMLELEMENT(
 NAME "employee", XMLNAMESPACES(
 DEFAULT 'http://hr.org'
),
 EMP.LASTNAME, XMLELEMENT(
 NAME "job", XMLNAMESPACES(
 NO DEFAULT
),
 EMP.JOB, XMLELEMENT(
 NAME "department", XMLNAMESPACES(
 DEFAULT 'http://adm.org'
),
 EMP.WORKDEPT
)
)
)
FROM EMPLOYEE EMP
WHERE EMP.EDLEVEL = 12
```

This query produces the following result:

```
000290 <employee xmlns="http://hr.org">PARKER<job xmlns="">OPERATOR
 <department xmlns="http://adm.org">E11</department></job></employee>
000310 <employee xmlns="http://hr.org">SETRIGHT<job xmlns="">OPERATOR
 <department xmlns="http://adm.org">E11</department></job></employee>
200310 <employee xmlns="http://hr.org">SPRINGER<job xmlns="">OPERATOR
 <department xmlns="http://adm.org">E11</department></job></employee>
```

## XMLPARSE



The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLPARSE function parses the argument as an XML document and returns an XML value.

**DOCUMENT**

Specifies that the character string expression to be parsed must evaluate to a well-formed XML document that conforms to XML 1.0, as modified by the XML Namespaces recommendation (SQLSTATE 2200M).

*string-expression*

Specifies an expression that returns a character string or BLOB value. If a parameter marker is used, it must explicitly be cast to one of the supported data types.

**STRIP WHITESPACE or PRESERVE WHITESPACE**

Specifies whether or not whitespace in the input argument is to be preserved. If neither is specified, STRIP WHITESPACE is the default.

**STRIP WHITESPACE**

Specifies that text nodes containing only whitespace characters up to 1000 bytes in length will be stripped, unless the nearest containing element has the attribute `xml:space='preserve'`. If any text node begins with more than 1000 bytes of whitespace, an error is returned (SQLSTATE 54059).

The whitespace characters in the CDATA section are also affected by this option. DTDs may have DOCTYPE declarations for elements, but the content models of elements are not used to determine if whitespace is stripped or not.

**PRESERVE WHITESPACE**

Specifies that all whitespace is to be preserved, even when the nearest containing element has the attribute `xml:space='default'`.

The data type of the result is XML. If the result of *string-expression* can be null, the result can be null; if the result of *string-expression* is null, the result is the null value.

**Note:**

1. **Support in multiple database partition databases:** XMLPARSE is not supported (SQLSTATE 42997).
2. **Encoding of the input string:** The input string may contain an XML declaration that identifies the encoding of the characters in the XML document. If the string is passed to the XMLPARSE function as a character string, it will be converted to the code page at the database server. This code page may be different from the originating code page and the encoding identified in the XML declaration.

Therefore, applications should avoid direct use of XMLPARSE with character string input and should send strings containing XML documents directly using host variables to maintain the match between the external code page and the

encoding in the XML declaration. If XMLPARSE must be used in this situation, a BLOB type should be specified as the argument to avoid code page conversion.

3. **Handling of DTDs:** External document type definitions (DTDs) and entities must be registered in a database. Both internal and external DTDs are checked for valid syntax. During the parsing process, the following actions are also performed:
  - Default values that are defined by the internal and external DTDs are applied.
  - Entity references and parameter entities are replaced by their expanded forms.
  - If an internal DTD and an external DTD define the same element, an error is returned (SQLSTATE 2200M).
  - If an internal DTD and an external DTD define the same entity or attribute, the internal definition is chosen.

After parsing, internal DTDs and entities, as well as references to external DTDs and entities, are not preserved in the stored representation of the value.

4. **Character conversion in non-UTF-8 databases:** Code page conversion occurs when an XML document is parsed into a non-Unicode database server, if the document is passed in from a host variable or parameter marker of a character data type, or from a character string literal. Parsing an XML document using a host variable or parameter marker of type XML, BLOB or FOR BIT DATA (CHAR FOR BIT DATA or VARCHAR FOR BIT DATA) prevents code page conversion. When a character data type is used, care must be taken to ensure that all characters in the XML document have a matching code point in the target database code page, otherwise substitution characters may be introduced. The configuration parameter `enable_xmlchar` can be used to help ensure the integrity of XML data stored in a non-Unicode database. Setting this parameter to "NO" blocks the insertion of XML documents from character data types. The BLOB and FOR BIT DATA data types are still allowed, as documents passed into a database using these data types avoid code page conversion.

## Example

Using the PRESERVE WHITESPACE option preserves the white space characters in the XML document inserted into the table, including the white space characters in the description element.

```
INSERT INTO PRODUCT VALUES ('100-103-99','Tool bag',14.95,NULL,NULL,NULL,
XMLPARSE(DOCUMENT
 '<produce xmlns="http://posample.org" pid="100-103-99">
 <description>
 <name>Tool bag</name>
 <details>
 Super Deluxe tool bag:
 - 26 inches long, 12 inches wide
 - Curved padded handle
 - Locking latch
 - Reinforced exterior pockets
 </details>
 <price>14.95</price>
 <weight>3 kg</weight>
 </description>
 </product>' PRESERVE WHITESPACE));
```

Running the following select statement

```
SELECT XMLQUERY ('$d/*:product/*:description/*:details' passing DESCRIPTION as "d")
FROM PRODUCT WHERE PID = '100-103-99' ;
```

returns the details element with the white space characters:

```
<details xmlns="http://posample.org">
 Super Deluxe tool bag:
 - 26 inches long, 12 inches wide
 - Curved padded handle
 - Locking latch
 - Reinforced exterior pockets
</details>
```

## XMLPI

```
►► XMLPI ((—NAME— pi-name) , —string-expression)
```

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLPI function returns an XML value with a single XQuery processing instruction node.

### NAME *pi-name*

Specifies the name of a processing instruction. The name is an SQL identifier that must be in the form of an XML NCName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. The name cannot be the word 'xml' in any case combination (SQLSTATE 42634).

### *string-expression*

An expression that returns a value that is a character string. The resulting string is converted to UTF-8 and must conform to the content of an XML processing instruction as specified in XML 1.0 rules (SQLSTATE 2200T):

- The string must not contain the substring '?>' since this substring terminates a processing instruction
- Each character of the string can be any Unicode character excluding the surrogate blocks, X'FFFE' and X'FFFF'.

The resulting string becomes the content of the constructed processing instruction node.

The data type of the result is XML. If the result of *string-expression* can be null, the result can be null; if the result of *string-expression* is null, the result is the null value. If *string-expression* is an empty string or is not specified, an empty processing instruction node is returned.

### Note:

1. **Support in multiple database partition databases:** XMLPI is not supported (SQLSTATE 42997).

Examples:

- Generate an XML processing instruction node.

```
SELECT XMLPI (
 NAME "Instruction", 'Push the red button'
)
FROM SYSIBM.SYSDUMMY1
```

This query produces the following result:

## XMLPI

<?Instruction Push the red button?>

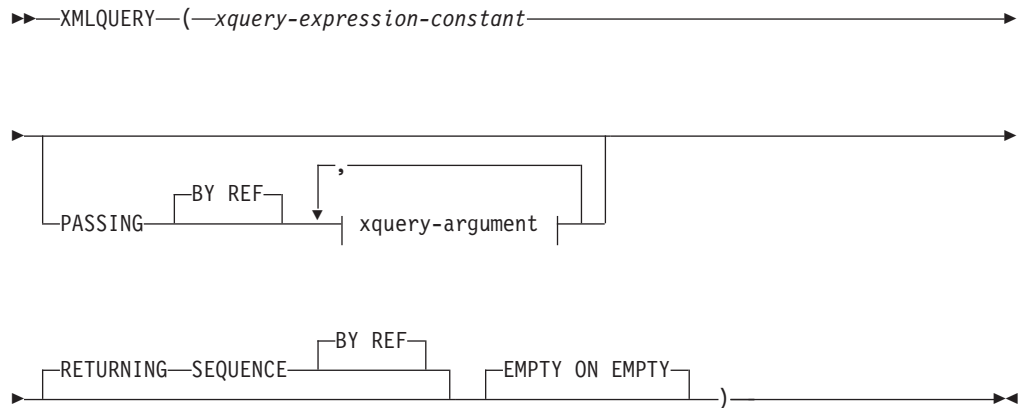
- Generate an empty XML processing instruction node.

```
SELECT XMLPI(
 NAME "Warning"
)
FROM SYSIBM.SYSDUMMY1
```

This query produces the following result:

<?Warning ?>

## XMLQUERY



### xquery-argument:

(1)  
| -xquery-variable-expression- AS -identifier- |  
BY REF

### Notes:

- 1 The data type of the expression cannot be DECFLOAT.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLQUERY function returns an XML value from the evaluation of an XQuery expression possibly using specified input arguments as XQuery variables.

### xquery-expression-constant

Specifies an SQL character string constant that is interpreted as an XQuery expression using supported XQuery language syntax. The constant string is converted to UTF-8 before being parsed as an XQuery statement. The XQuery expression executes using an optional set of input XML values, and returns an output sequence that is also returned as the value of the XMLQUERY expression. The value for *xquery-expression-constant* must not be an empty string or a string of blank characters (SQLSTATE 10505).

### PASSING

Specifies input values and the manner in which these values are passed to the XQuery expression specified by *xquery-expression-constant*. By default, every unique column name that is in the scope where the function is invoked is implicitly passed to the XQuery expression using the name of the column as the variable name. If an *identifier* in a specified *xquery-argument* matches an



in-scope column name, then the explicit *xquery*-argument is passed to the XQuery expression overriding that implicit column.

#### BY REF

Specifies that the default passing mechanism is by reference for any *xquery-variable-expression* of data type XML and for the returned value. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any, directly from the specified input expressions, preserving all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons involving some nodes contained between the two input arguments might refer to nodes within the same XML node tree.

This clause has no impact on how non-XML values are passed. The non-XML values create a new copy of the value during the cast to XML.

#### *xquery*-argument

Specifies an argument that is to be passed to the XQuery expression specified by *xquery-expression-constant*. An argument specifies a value and the manner in which that value is to be passed. The argument includes an SQL expression that is evaluated.

- If the resulting value is of type XML, it becomes an *input-xml-value*. A null XML value is converted to an XML empty sequence.
- If the resulting value is not of type XML, it must be castable to the XML data type. A null value is converted to an XML empty sequence. The converted value becomes an *input-xml-value*.

When the *xquery-expression-constant* is evaluated, an XQuery variable is presented with a value equal to *input-xml-value* and a name specified by the AS clause.

#### *xquery-variable-expression*

Specifies an SQL expression whose value is available to the XQuery expression specified by *xquery-expression-constant* during execution. The expression cannot contain a sequence reference (SQLSTATE 428F9) or an OLAP function (SQLSTATE 42903). The data type of the expression cannot be DECFLOAT.

#### AS *identifier*

Specifies that the value generated by *xquery-variable-expression* will be passed to *xquery-expression-constant* as an XQuery variable. The variable name will be *identifier*. The leading dollar sign (\$) that precedes variable names in the XQuery language is not included in *identifier*. The identifier must be a valid XQuery variable name and is restricted to an XML NCName (SQLSTATE 42634). The identifier must not be greater than 128 bytes in length. Two arguments within the same PASSING clause cannot use the same identifier (SQLSTATE 42711).

#### BY REF

Indicates that an XML input value is to be passed by reference. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any, directly from the specified input expressions, preserving all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons involving some nodes contained between the two input arguments might refer to nodes within the same XML node tree. If BY REF is not specified following an *xquery-variable-expression*, XML arguments are passed by

## XMLQUERY

way of the default passing mechanism that is provided through the syntax that follows the `PASSING` keyword. This option cannot be specified for non-XML values. When a non-XML value is passed, the value is converted to XML; this process creates a copy.

### RETURNING SEQUENCE

Indicates that the `XMLQUERY` expression returns a sequence.

### BY REF

Indicates that the result of the XQuery expression is returned by reference. If this value contains nodes, any expression using the return value of the XQuery expression will receive node references directly, preserving all node properties, including the original node identities and document order. Referenced nodes will remain connected within their node trees. If the `BY REF` clause is not specified and the `PASSING` is specified, the default passing mechanism is used. If `BY REF` is not specified and `PASSING` is not specified, the default returning mechanism is `BY REF`.

### EMPTY ON EMPTY

Specifies that an empty sequence result from processing the XQuery expression is returned as an empty sequence.

The data type of the result is XML; it cannot be null.

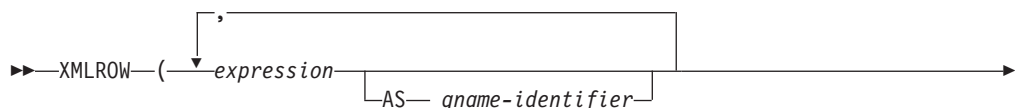
If the evaluation of the XQuery expression results in an error, then the `XMLQUERY` function returns the XQuery error (SQLSTATE class '10').

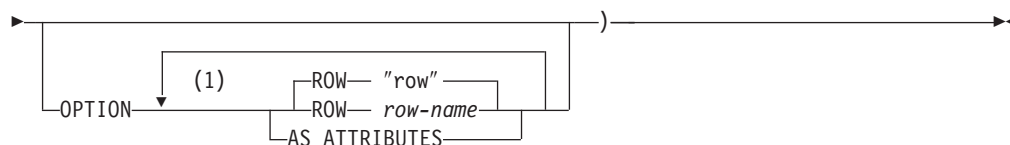
### Note:

1. **XMLQUERY usage restrictions:** The `XMLQUERY` function cannot be:
  - Part of the `ON` clause that is associated with a `JOIN` operator or a `MERGE` statement (SQLSTATE 42972)
  - Part of the `GENERATE KEY USING` or `RANGE THROUGH` clause in the `CREATE INDEX EXTENSION` statement (SQLSTATE 428E3)
  - Part of the `FILTER USING` clause in the `CREATE FUNCTION (External Scalar)` statement, or the `FILTER USING` clause in the `CREATE INDEX EXTENSION` statement (SQLSTATE 428E4)
  - Part of a check constraint or a column generation expression (SQLSTATE 42621)
  - Part of a group-by-clause (SQLSTATE 42822)
  - Part of an argument for a column-function (SQLSTATE 42607)
2. **XMLQUERY as a subquery:** An `XMLQUERY` expression that acts as a subquery can be restricted by statements that restrict subqueries.
3. **Support in multiple database partition databases:** `XMLQUERY` is not supported (SQLSTATE 42997).

## XMLROW

The `XMLROW` function returns an XML value with a single XQuery document node containing one top-level element node.



**Notes:**

- 1 The same clause must not be specified more than once.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

*expression*

The content of each generated XML element node is specified by an expression. The data type of the expression cannot be a structured type (SQLSTATE 42884). The expression can be any SQL expression. If the expression is not a simple column reference, an element name must be specified.

**AS *qname-identifier***

Specifies the XML element name or attribute name as an SQL identifier. The *qname-identifier* must be of the form of an XML qualified name, or QName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope (SQLSTATE 42635). If *qname-identifier* is not specified, *expression* must be a column name (SQLSTATE 42703, SQLCODE -206). The element name or attribute name is created from the column name using the fully escaped mapping from a column name to an QName.

**OPTION**

Specifies additional options for constructing the XML value. If no OPTION clause is specified, the default behavior applies.

**AS ATTRIBUTES**

Specifies that each expression is mapped to an attribute value with column name or *qname-identifier* serving as the attribute name.

**ROW *row-name***

Specifies the name of the element to which each row is mapped. If this option is not specified, the default element name is "row".

**Notes**

By default, each row in the result set is mapped to an XML value as follows:

- Each row is transformed into an XML element named "row" and each column is transformed into a nested element with the column name as the element name.
- The null handling behavior is NULL ON NULL. A NULL value in a column maps to the absence of the subelement. If all column values are NULL, a NULL value is returned by the function.
- The binary encoding scheme for BLOB and FOR BIT DATA data types is base64Binary encoding.
- A document node will be added implicitly to the row element to make the XML result a well-formed single-rooted XML document.

## XMLROW

### Examples

Assume the following table T1 with columns C1 and C2 that contain numeric data stored in a relational format:

C1	C2
1	2
-	2
1	-
-	-

4 record(s) selected.

- The following example shows an XMLRow query and output fragment with default behavior, using a sequence of row elements to represent the table:

```
SELECT XMLROW(C1, C2) FROM T1
<row><C1>1</C1><C2>2</C2></row>
<row><C2>2</C2></row>
<row><C1>1</C1></row>
```

4 record(s) selected.

- The following example shows an XMLRow query and output fragment with attribute centric mapping. Instead of appearing as nested elements as in the previous example, relational data is mapped to element attributes:

```
SELECT XMLROW(C1, C2 OPTION AS ATTRIBUTES) FROM T1
<row C1="1" C2="2"/>
<row C2="2"/>
<row C1="1"/>
```

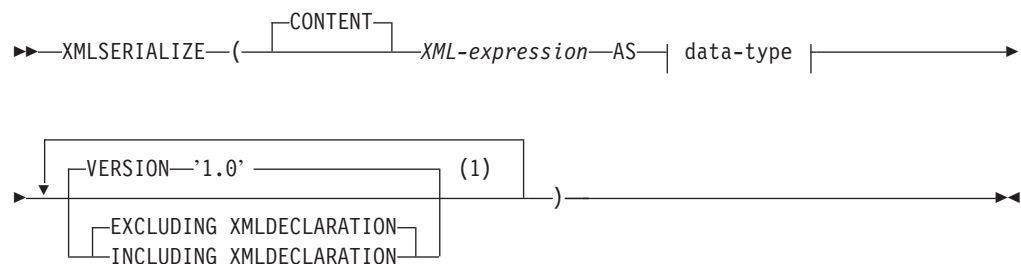
4 record(s) selected.

- The following example shows an XMLRow query and output fragment with the default <row> element replaced by <entry>. Columns C1 and C2 are returned as <column1> and <column2> elements, and the total of C1 and C2 is returned inside a <total> element:

```
SELECT XMLROW(
 C1 AS "column1", C2 AS "column2",
 C1+C2 AS "total" OPTION ROW "entry")
FROM T1
<entry><column1>1</column1><column2>2</column2><total>3</total></entry>
<entry><column2>2</column2></entry>
<entry><column1>1</column1></entry>
```

4 record(s) selected.

## XMLSERIALIZE





## XMLSERIALIZE

*XML-expression* prior to serializing the resulting XML nodes. If the result of *XML-expression* can be null, the result can be null; if the result of *XML-expression* is null, the result is the null value.

### Note:

1. **Support in multiple database partition databases:** The function is only supported as it was in Version 8. The CONTENT keyword must be specified, a BLOB data type cannot be specified, and an XMLDECLARATION option cannot be specified (SQLSTATE 42997).
2. **Encoding in the serialized result:** The serialized result is encoded with UTF-8. If XMLSERIALIZE is used with a character data type, and the INCLUDING XMLDECLARATION clause is specified, the resulting character string containing serialized XML might have an XML encoding declaration that does not match the code page of the character string. Following serialization, which uses UTF-8 encoding, the character string that is returned from the server to the client is converted to the code page of the client, and that code page might be different from UTF-8.

Therefore, applications should avoid direct use of XMLSERIALIZE INCLUDING XMLDECLARATION that return character string types and should retrieve XML values directly into host variables to maintain the match between the external code page and the encoding in the XML declaration. If XMLSERIALIZE must be used in this situation, a BLOB type should be specified to avoid code page conversion.

3. **Syntax alternative:** XMLCLOB(*XML-expression*) can be specified in place of XMLSERIALIZE(*XML-expression* AS CLOB(2G)). It is supported only for compatibility with previous DB2 releases.

## XMLTEXT

►► XMLTEXT (—*string-expression*—) ◀◀

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLTEXT function returns an XML value with a single XQuery text node having the input argument as the content.

*string-expression*

An expression whose value has a character string type: CHAR, VARCHAR or CLOB.

The data type of the result is XML. If the result of *string-expression* can be null, the result can be null; if the input value is null, the result is the null value. If the result of *string-expression* is an empty string, the result value is an empty text node.

### Note:

1. **Support in non-Unicode databases and multiple database partition databases:** XMLTEXT is not supported (SQLSTATE 42997).

Examples:

- Create a simple XMLTEXT query.

```
VALUES(
 XMLTEXT(
 'The stock symbol for Johnson&Johnson is JNJ.'
)
)
```

This query produces the following serialized result:

```
1

The stock symbol for Johnson&Johnson is JNJ.
```

Note that the '&' sign is mapped to '&amp;,' when a text node is serialized.

- Use XMLTEXT with XMLAGG to construct mixed content. Suppose that the content of table T is as follows:

seqno	plaintext	emphertext
1	This query shows how to construct	mixed content
2	using XMLAGG and XMLTEXT. Without	XMLTEXT
3	XMLAGG will not have text nodes to group with other nodes, therefore, cannot generate	mixed content

```
SELECT XMLELEMENT(
 NAME "para", XMLAGG(
 XMLCONCAT(
 XMLTEXT(
 PLAINTEXT
),
 XMLELEMENT(
 NAME "emphasis", EMPHTEXT
)
)
)
 ORDER BY SEQNO
), '.'
) AS "result"
FROM T
```

This query produces the following result:

```
result

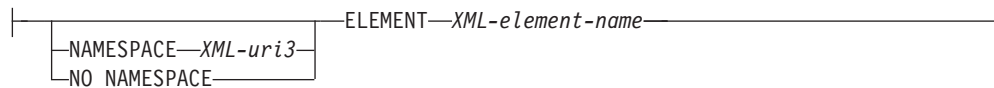
<para>This query shows how to construct <emphasis>mixed content</emphasis>
using XMLAGG and XMLTEXT. Without <emphasis>XMLTEXT</emphasis> , XMLAGG
will not have text nodes to group with other nodes, therefore, cannot generate
<emphasis>mixed content</emphasis>.</para>
```

## XMLVALIDATE

```
XMLVALIDATE((DOCUMENT XML-expression) XML-validate-according-to-clause)
```

### XML-validate-according-to-clause:

```
ACCORDING TO XMLSCHEMA ID XML-schema-name
 URI XML-uri1
 NO NAMESPACE LOCATION XML-uri2
XML-valid-element-clause
```

**XML-valid-element-clause:**

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLVALIDATE function returns a copy of the input XML value augmented with information obtained from XML schema validation, including default values and type annotations.

**DOCUMENT**

Specifies that the XML value resulting from *XML-expression* must be a well-formed XML document that conforms to XML Version 1.0 (SQLSTATE 2200M).

*XML-expression*

An expression that returns a value of data type XML. If *XML-expression* is an XML host variable or an implicitly or explicitly typed parameter marker, the function performs a validating parse that strips ignorable whitespace and the CURRENT IMPLICIT XMLPARSE OPTION setting is not considered.

**XML-validate-according-to-clause**

Specifies the information that is to be used when validating the input XML value.

**ACCORDING TO XMLSCHEMA**

Indicates that the XML schema information for validation is explicitly specified. If this clause is not included, the XML schema information must be provided in the content of the *XML-expression* value.

**ID** *XML-schema-name*

Specifies an SQL identifier for the XML schema that is to be used for validation. The name, including the implicit or explicit SQL schema qualifier, must uniquely identify an existing XML schema in the XML schema repository at the current server. If no XML schema by this name exists in the implicitly or explicitly specified SQL schema, an error is returned (SQLSTATE 42704).

**URI** *XML-uri1*

Specifies the target namespace URI of the XML schema that is to be used for validation. The value of *XML-uri1* specifies a URI as a character string constant that is not empty. The URI must be the target namespace of a registered XML schema (SQLSTATE 4274A) and, if no LOCATION clause is specified, it must uniquely identify the registered XML schema (SQLSTATE 4274B).

**NO NAMESPACE**

Specifies that the XML schema for validation has no target namespace. The target namespace URI is equivalent to an empty character string that cannot be specified as an explicit target namespace URI.

**LOCATION** *XML-uri2*

Specifies the XML schema location URI of the XML schema that is to be used for validation. The value of *XML-uri2* specifies a URI as a character string constant that is not empty. The XML schema location URI, combined with the target namespace URI, must identify a



registered XML schema (SQLSTATE 4274A), and there must be only one such XML schema registered (SQLSTATE 4274B).

#### **XML-valid-element-clause**

Specifies that the XML value in *XML-expression* must have the specified element name as the root element of the XML document.

#### **NAMESPACE *XML-uri3* or NO NAMESPACE**

Specifies the target namespace for the element that is to be validated. If neither clause is specified, the specified element is assumed to be in the same namespace as the target namespace of the registered XML schema that is to be used for validation.

#### **NAMESPACE *XML-uri3***

Specifies the namespace URI for the element that is to be validated. The value of *XML-uri3* specifies a URI as a character string constant that is not empty. This can be used when the registered XML schema that is to be used for validation has more than one namespace.

#### **NO NAMESPACE**

Specifies that the element for validation has no target namespace. The target namespace URI is equivalent to an empty character string which cannot be specified as an explicit target namespace URI.

#### **ELEMENT *xml-element-name***

Specifies the name of a global element in the XML schema that is to be used for validation. The specified element, with implicit or explicit namespace, must match the root element of the value of *XML-expression* (SQLSTATE 22535 or 22536).

The data type of the result is XML. If the value of *XML-expression* can be null, the result can be null; if the value of *XML-expression* is null, the result is the null value.

The XML validation process is performed on a serialized XML value. Because XMLVALIDATE is invoked with an argument of type XML, this value is automatically serialized prior to validation processing with the follow two exceptions.

- If the argument to XMLVALIDATE is an XML host variable or an implicitly or explicitly typed parameter marker, then a validating parse operation is performed on the input value (no implicit non-validating parse is performed and CURRENT IMPLICIT XMLPARSE OPTION setting is not considered).
- If the argument to XMLVALIDATE is an XMLPARSE invocation using the option PRESERVE WHITESPACE, then the XML parsing and XML validation of the document may be combined into a single validating parse operation.

If an XML value has previously been validated, the annotated type information from the previous validation is removed by the serialization process. However, any default values and entity expansions from the previous validation remain unchanged. If validation is successful, all ignorable whitespace characters are stripped from the result.

To validate a document whose root element does not have a namespace, an xsi:noNamespaceSchemaLocation attribute must be present on the root element.

#### **Note:**

1. **Support in multiple database partition databases:** XMLVALIDATE is not supported (SQLSTATE 42997).

2. **Determining the XML schema:** The XML schema can be specified explicitly as part of XMLVALIDATE invocation, or determined from the XML schema information in the input XML value. If the XML schema information is not specified during invocation, the target namespace and the schema location in the input XML value are used to identify the registered schema for validation. If an explicit XML schema is not specified, the input XML value must contain an XML schema information hint (SQLSTATE 2200M). Explicit or implicit XML schema information must identify a registered XML schema (SQLSTATE 42704, 4274A, or 22532), and there must be only one such registered XML schema (SQLSTATE 4274B or 22533).
3. **XML schema authorization:** The XML schema used for validation must be registered in the XML schema repository prior to use. The privileges held by the authorization ID of the statement must include at least one of the following:
  - USAGE privilege on the XML schema that is to be used during validation
  - SYSADM or DBADM authority

Examples:

- Validate using the XML schema identified by the XML schema hint in the XML instance document.

```
INSERT INTO T1(XMLCOL)
VALUES (XMLVALIDATE(?))
```

Assume that the input parameter marker is bound to an XML value that contains the XML schema information.

```
<po:order
 xmlns:po='http://my.world.com'
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://my.world.com/world.xsd" >
...
</po:order>
```

Further, assume that the XML schema that is associated with the target namespace "http://my.world.com" and by schemaLocation hint "http://my.world.com/world.xsd" is found in the XML schema repository.

Based on these assumptions, the input XML value will be validated and the type annotated according to that XML schema.

- Validate using the XML schema identified by the SQL name PODOCS.WORLDPO.

```
INSERT INTO T1(XMLCOL)
VALUES (
 XMLVALIDATE(
 ? ACCORDING TO XMLSCHEMA ID PODOCS.WORLDPO
)
)
```

Assuming that the XML schema that is associated with SQL name FOO.WORLDPO is found in the XML repository, the input XML value will be validated and the type annotated according to that XML schema.

- Validate a specified element of the XML value.

```
INSERT INTO T1(XMLCOL)
VALUES (
 XMLVALIDATE(
 ? ACCORDING TO XMLSCHEMA ID FOO.WORLDPO
 NAMESPACE 'http://my.world.com/Mary'
 ELEMENT "po"
)
)
```

Assuming that the XML schema that is associated with SQL name FOO.WORLDP0 is found in the XML repository, the XML schema will be validated against the element "po", whose namespace is 'http://my.world.com/Mary'.

- XML schema is identified by target namespace and schema location.

```
INSERT INTO T1(XMLCOL)
VALUES (
 XMLVALIDATE(
 ? ACCORDING TO XMLSCHEMA URI 'http://my.world.com'
 LOCATION 'http://my.world.com/world.xsd'
)
)
```

Assuming that an XML schema associated with the target namespace "http://my.world.com" and by schemaLocation hint "http://my.world.com/world.xsd" is found in the XML schema repository, the input XML value will be validated and the type annotated according to that schema.

## XMLXSROBJECTID

►►—XMLXSROBJECTID—(—*xml-value-expression*—)—————►►

The schema is SYSIBM.

The XMLXSROBJECTID function returns the XSR object identifier of the XML schema used to validate the XML document specified in the argument. The XSR object identifier is returned as a BIGINT value and provides the key to a single row in SYSCAT.XSROBJECTS.

*xml-value-expression*

Specifies an expression that results in a value with a data type of XML. The resulting XML value must be an XML sequence with a single item that is an XML document or the null value (SQLSTATE 42815). If the argument is null, the function returns null. If *xml-value-expression* does not specify a validated XML document, the function returns 0.

### Note:

1. **Support in multiple database partition databases:** XMLXSROBJECTID is not supported (SQLSTATE 42997).
2. The XML schema corresponding to an XSR object ID returned by the function might no longer exist, because an XML schema can be dropped without affecting XML values that were validated using the XML schema. Therefore, queries that use the XSR object ID to fetch further XML schema information from the catalog views might return an empty result set.
3. Applications can use the XSR object identifier to retrieve additional information about the XML schema. For example, the XSR object identifier can be used to return the individual XML schema documents that make up a registered XML schema from SYSCAT.SYSXSROBJECTCOMPONENTS, and the hierarchy of XML schema documents in the XML schema from SYSCAT.XSROBJECTHIERARCHIES.

Examples:

- Retrieve the XML schema identifier for the XML document XMLDOC stored in the table MYTABLE.

```
SELECT XMLXSROBJECTID(XMLDOC) FROM MYTABLE
```

## XMLXSROBJECTID

- Retrieve the XML schema documents associated with the XML document that has a specific ID (in this case where DOCKEY = 1) in the table MYTABLE, including the hierarchy of the XML schema documents that make up the XML schema.

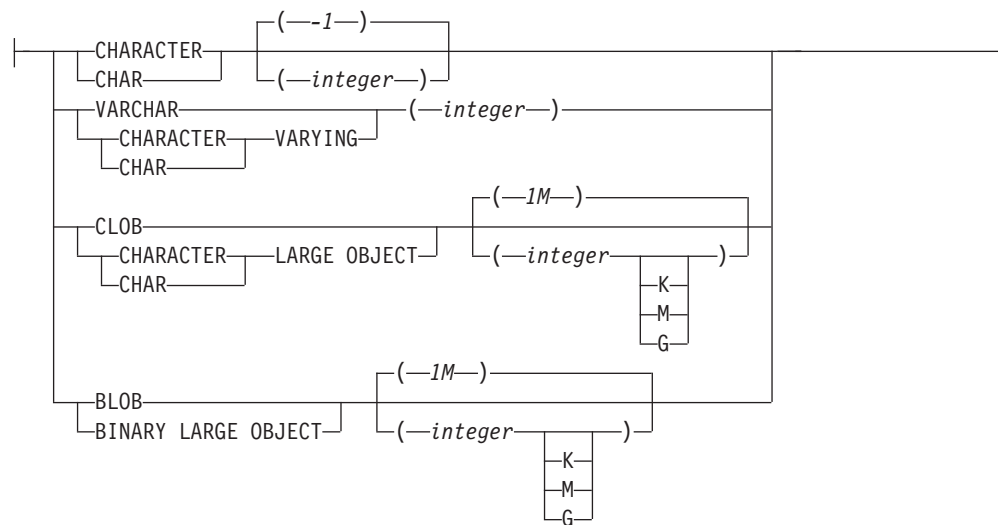
```
SELECT H.HTYPE, C.TARGETNAMESPACE, C.COMPONENT
FROM SYSCAT.XSROBJECTCOMPONENTS C, SYSCAT.XSROBJECTHIERARCHIES H
WHERE C.OBJECTID =
 (SELECT XMLXSROBJECTID(XMLDOC) FROM MYTABLE
 WHERE DOCKEY = 1)
AND C.OBJECTID = H.XSROBJECTID
```

## XSLTRANSFORM

Use XSLTRANSFORM to convert XML data into other formats, including the conversion of XML documents that conform to one XML schema into documents that conform to another schema.

```
►►—XSLTRANSFORM—►►
►—(—xml-document—USING—xsl-styleSheet—)
 |
 |—WITH—xsl-parameters—|
 |
 |—AS CLOB(2G)—|
 |—AS—| data-type —|
```

### data-type:



The schema is SYSIBM. This function cannot be specified as a qualified name.

The XSLTRANSFORM function transforms an XML document into a different data format. The data can be transformed into any form possible for the XSLT processor, including but not limited to XML, HTML, or plain text.

All paths used by XSLTRANSFORM are internal to the database system. This command cannot currently be used directly with files or stylesheets residing in an external file system.

### xml-document

An expression that returns a well-formed XML document with a data type of XML, CHAR, VARCHAR, CLOB, or BLOB. This is the document that is transformed using the XSL style sheet specified in *xsl-styleSheet*.

**Note:**

The XML document must at minimum be single-rooted and well-formed.

*xsl-stylesheet*

An expression that returns a well-formed XML document with a data type of XML, CHAR, VARCHAR, CLOB, or BLOB. The document is an XSL style sheet that conforms to the W3C XSLT Version 1.0 Recommendation. Style sheets incorporating XQUERY statements or the `xsl:include` declaration are not supported. This stylesheet is applied to transform the value specified in *xml-document*.

*xsl-parameters*

An expression that returns a well-formed XML document or null with a data type of XML, CHAR, VARCHAR, CLOB, or BLOB. This is a document that provides parameter values to the XSL stylesheet specified in *xsl-stylesheet*. The value of the parameter can be specified as an attribute, or as a text node.

The syntax of the parameter document is as follows:

```
<params xmlns="http://www.ibm.com/XSLTransformParameters">
<param name="..." value="..." />
<param name="...">enter value here</param>
...
</params>
```

**Note:**

The stylesheet document must have `xsl:param` element(s) in it with name attribute values that match the ones specified in the parameter document.

**AS** *data-type*

Specifies the result data type. The implicit or explicit length attribute of the specified result data type must be sufficient to contain the transformed output (SQLSTATE 22001). The default result data type is CLOB(2G).

**Note:**

If either the *xml-document* argument or the *xsl-stylesheet* argument is null, the result will be null.

Code page conversion might occur when storing any of the above documents in a CHAR, VARCHAR, or CLOB column, which might result in a character loss.

**Example**

This example illustrates how to use XSLT as a formatting engine. To get set up, first insert the two example documents below into the database.

```
INSERT INTO XML_TAB VALUES
(1,
 '<?xml version="1.0"?>
<students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation = "/home/steffen/xsd/xslt.xsd">
<student studentID="1" firstName="Steffen" lastName="Siegmund"
 age="23" university="Rostock"/>
</students>',
 '<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:param name="headline"/>
```

## XSLTRANSFORM

```
<xsl:param name="showUniversity"/>
<xsl:template match="students">
 <html>
 <head/>
 <body>
 <h1><xsl:value-of select="$headline"/></h1>
 <table border="1">
 <thead>
 <tr>
 <th>
 <tr>
 <td width="80">StudentID</td>
 <td width="200">First Name</td>
 <td width="200">Last Name</td>
 <td width="50">Age</td>
 <xsl:choose>
 <xsl:when test="$showUniversity = 'true'">
 <td width="200">University</td>
 </xsl:when>
 </xsl:choose>
 </tr>
 </th>
 </thead>
 <xsl:apply-templates/>
 </table>
 </body>
 </html>
 </xsl:template>
 <xsl:template match="student">
 <tr>
 <td><xsl:value-of select="@studentID"/></td>
 <td><xsl:value-of select="@firstName"/></td>
 <td><xsl:value-of select="@lastName"/></td>
 <td><xsl:value-of select="@age"/></td>
 <xsl:choose>
 <xsl:when test="$showUniversity = 'true' ">
 <td><xsl:value-of select="@university"/></td>
 </xsl:when>
 </xsl:choose>
 </tr>
 </xsl:template>
</xsl:stylesheet>';
```

Next, call the XSLTRANSFORM function to convert the XML data into HTML and display it.

```
SELECT XSLTRANSFORM (XML_DOC USING XSL_DOC AS CLOB(1M)) FROM XML_TAB;
```

The result is this document:

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
<h1></h1>
<table border="1">
<thead>
<tr>
<td width="80">StudentID</td>
<td width="200">First Name</td>
<td width="200">Last Name</td>
<td width="50">Age</td>
</tr>
</thead>
<tr>
<td>1</td>
<td>Steffen</td><td>Siegmond</td>
<td>23</td>
```

```

</tr>
 </table>
</body>
</html>

```

In this example, the output is HTML and the parameters influence only what HTML is produced and what data is brought over to it. As such it illustrates the use of XSLT as a formatting engine for end-user output.

### Usage note:

This function is **NOT** intended for high-performance applications and **CANNOT** replace similar functionality in the application server.

## 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. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument specified:

- If the argument is a date, timestamp, or valid string representation of a date or timestamp:
  - The result is the year part of the value, which is an integer between 1 and 9999.
- If the argument is a date duration or timestamp duration:
  - The result is the year part of the value, which is an integer between -9999 and 9999. A nonzero result has the same sign as the argument.

Examples:

- 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.

```

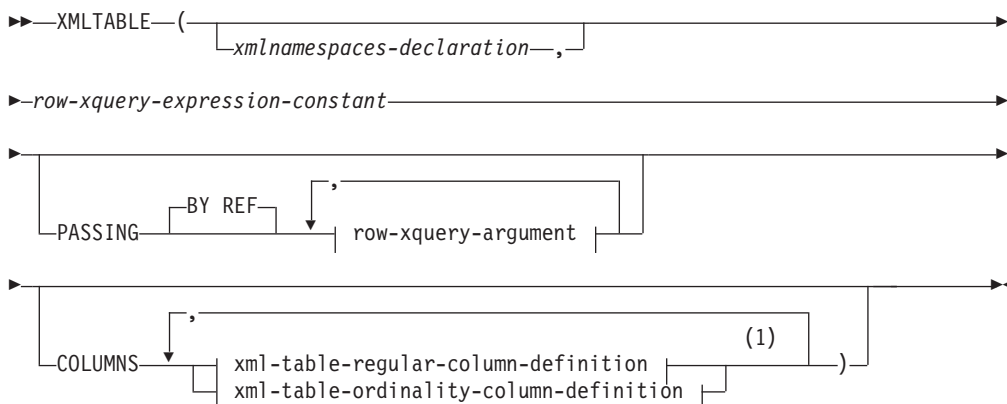
SELECT * FROM PROJECT
WHERE YEAR(PRENDATE - PRSTDATE) < 1

```

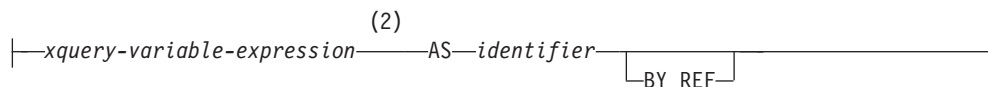
## Table functions

A table function can be used only in the FROM clause of a statement. Table functions return columns of a table, resembling a table created through a simple CREATE TABLE statement. Table functions can be qualified with a schema name.

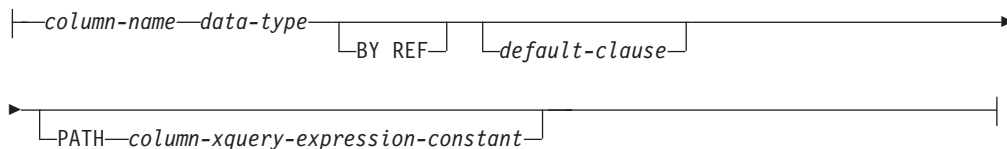
### XMLTABLE



#### row-xquery-argument:



#### xml-table-regular-column-definition:



#### xml-table-ordinality-column-definition:



#### Notes:

- 1 The xml-table-ordinality-column-definition clause must not be specified more than once (SQLSTATE 42614, SQLCODE -637).
- 2 The data type of the expression cannot be DECFLOAT.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The XMLTABLE function returns a result table from the evaluation of XQuery expressions, possibly using specified input arguments as XQuery variables. Each sequence item in the result sequence of the row XQuery expression represents a row of the result table.



*xmlnamespaces-declaration*

Specifies one or more XML namespace declarations that become part of the static context of the *row-xquery-expression-constant* and the *column-xquery-expression-constant*. The set of statically known namespaces for XQuery expressions which are arguments of XMLTABLE is the combination of the pre-established set of statically known namespaces and the namespace declarations specified in this clause. The XQuery prolog within an XQuery expression may override these namespaces.

If *xmlnamespaces-declaration* is not specified, only the pre-established set of statically known namespaces apply to the the XQuery expressions.

*row-xquery-expression-constant*

Specifies an SQL character string constant that is interpreted as an XQuery expression using supported XQuery language syntax. The constant string is converted directly to UTF-8 without conversion to the database or section code page. The XQuery expression executes using an optional set of input XML values, and returns an output XQuery sequence where a row is generated for each item in the sequence. The value for *row-xquery-expression-constant* must not be an empty string or a string of all blanks (SQLSTATE 10505).

**PASSING**

Specifies input values and the manner in which these values are passed to the XQuery expression specified by *row-xquery-expression-constant*. By default, every unique column name that is in the scope where the function is invoked is implicitly passed to the XQuery expression using the name of the column as the variable name. If an *identifier* in a specified row-xquery-argument matches an in-scope column name, then the explicit row-xquery-argument is passed to the XQuery expression overriding that implicit column.

**BY REF**

Specifies that any XML input arguments are, by default, passed by reference. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any, directly from the specified input expressions, preserving all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons involving some nodes contained between the two input arguments might refer to nodes within the same XML node tree.

This clause has no impact on how non-XML values are passed. The non-XML values create a new copy of the value during the cast to XML.

**row-xquery-argument**

Specifies an argument that is to be passed to the XQuery expression specified by *row-xquery-expression-constant*. An argument specifies a value and the manner in which that value is to be passed. The argument includes an SQL expression that is evaluated before passing the result to the XQuery expression.

- If the resulting value is of type XML, it becomes an *input-xml-value*. A null XML value is converted to an XML empty sequence.
- If the resulting value is not of type XML, it must be castable to the XML data type. A null value is converted to an XML empty sequence. The converted value becomes an *input-xml-value*.

When the *row-xquery-expression-constant* is evaluated, an XQuery variable is presented with a value equal to *input-xml-value* and a name specified by the AS clause.

*xquery-variable-expression*

Specifies an SQL expression whose value is available to the XQuery expression specified by *row-xquery-expression-constant* during execution. The expression cannot contain a NEXT VALUE expression, PREVIOUS VALUE expression (SQLSTATE 428F9), or an OLAP function (SQLSTATE 42903). The data type of the expression cannot be DECFLOAT.

**AS** *identifier*

Specifies that the value generated by *xquery-variable-expression* will be passed to *row-xquery-expression-constant* as an XQuery variable. The variable name will be *identifier*. The leading dollar sign (\$) that precedes variable names in the XQuery language is not included in *identifier*. The identifier must be a valid XQuery variable name and is restricted to an XML NCName. The identifier must not be greater than 128 bytes in length. Two arguments within the same PASSING clause cannot use the same identifier (SQLSTATE 42711).

**BY REF**

Indicates that an XML input value is to be passed by reference. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any, directly from the specified input expressions, preserving all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons involving some nodes contained between the two input arguments might refer to nodes within the same XML node tree. If BY REF is not specified following an *xquery-expression-variable*, XML arguments are passed by way of the default passing mechanism that is provided through the syntax that follows the PASSING keyword. This option cannot be specified for non-XML values (SQLSTATE 42636). When a non-XML value is passed, the value is converted to XML; this process creates a copy.

**COLUMNS**

Specifies the output columns of the result table. If this clause is not specified, a single unnamed column of data type XML is returned by reference, with the value based on the sequence item from evaluating the XQuery expression in the *row-xquery-expression-constant* (equivalent to specifying PATH '.'). To reference the result column, a *column-name* must be specified in the *correlation-clause* following the function.

**xml-table-regular-column-definition**

Specifies the output columns of the result table including the column name, data type, XML passing mechanism and an XQuery expression to extract the value from the sequence item for the row

*column-name*

Specifies the name of the column in the result table. The name cannot be qualified and the same name cannot be used for more than one column of the table (SQLSTATE 42711).

*data-type*

Specifies the data type of the column. See CREATE TABLE for the syntax and a description of types available. A *data-type* may be used in XMLTable if there is a supported XMLCAST from the XML data type to the specified *data-type*.

**BY REF**

Specifies that XML values are returned by reference for columns of data type XML. By default, XML values are returned BY REF. When XML values are returned by reference, the XML value includes the input node trees, if any, directly from the result values, and preserves all properties, including the original node identities and document order. This option cannot be specified for non-XML columns (SQLSTATE 42636). When a non-XML column is processed, the value is converted from XML; this process creates a copy.

*default-clause*

Specifies a default value for the column. See CREATE TABLE for the syntax and a description of the *default-clause*. For XMLTABLE result columns, the default is applied when the processing the XQuery expression contained in *column-xquery-expression-constant* returns an empty sequence.

**PATH** *column-xquery-expression-constant*

Specifies an SQL character string constant that is interpreted as an XQuery expression using supported XQuery language syntax. The constant string is converted directly to UTF-8 without conversion to the database or section code page. The *column-xquery-expression-constant* specifies an XQuery expression that determines the column value with respect to an item that is the result of evaluating the XQuery expression in *row-xquery-expression-constant*. Given an item from the result of processing the *row-xquery-expression-constant* as the externally provided context item, the *column-xquery-expression-constant* is evaluated, returning an output sequence. The column value is determined based on this output sequence as follows.

- If the output sequence contains zero items, the *default-clause* provides the value of the column.
- If an empty sequence is returned and no *default-clause* was specified, a null value is assigned to the column.
- If a non-empty sequence is returned, the value is XMLCAST to the *data-type* specified for the column. An error could be returned from processing this XMLCAST.

The value for *column-xquery-expression-constant* must not be an empty string or a string of all blanks (SQLSTATE 10505). If this clause is not specified, the default XQuery expression is simply the *column-name*.

**xml-table-ordinality-column-definition**

Specifies the ordinality column of the result table.

*column-name*

Specifies the name of the column in the result table. The name cannot be qualified and the same name cannot be used for more than one column of the table (SQLSTATE 42711).

**FOR ORDINALITY**

Specifies that *column-name* is the ordinality column of the result table. The data type of this column is BIGINT. The value of this column in the result table is the sequential number of the item for the row in the resulting sequence from evaluating the XQuery expression in *row-xquery-expression-constant*.

If the evaluation of any of the XQuery expressions results in an error, then the XMLTABLE function returns the XQuery error (SQLSTATE class '10').

## XMLTABLE

### Note:

1. **Support in multiple database partition databases:** XMLTABLE is not supported (SQLSTATE 42997).

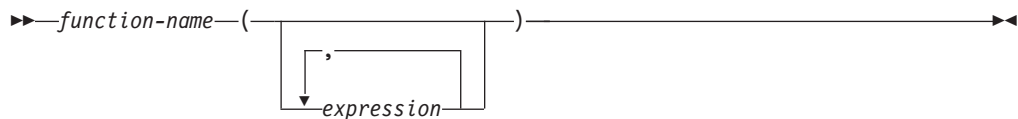
### Examples:

- List as a table result the purchase order items for orders with a status of 'NEW'.

```
SELECT U."PO ID", U."Part #", U."Product Name",
 U."Quantity", U."Price", U."Order Date"
FROM PURCHASEORDER P,
 XMLTABLE(XMLNAMESPACES('http://podemo.org' AS "pod"),
 '$po/PurchaseOrder/itemlist/item' PASSING P.PORDER AS "po"
 COLUMNS "PO ID" INTEGER PATH '../@POid',
 "Part #" CHAR(6) PATH 'product/@pid',
 "Product Name" CHAR(50) PATH 'product/pod:name',
 "Quantity" INTEGER PATH 'quantity',
 "Price" DECIMAL(9,2) PATH 'product/pod:price',
 "Order Date" TIMESTAMP PATH '../dateTime'
) AS U
WHERE P.STATUS = 'NEW'
```

---

## User-defined functions



*User-defined functions (UDFs)* are extensions or additions to the existing built-in functions of the SQL language. A user-defined function can be a scalar function, which returns a single value each time it is called; a column function, which is passed a set of like values and returns a single value for the set; a row function, which returns one row; or a table function, which returns a table.

A number of user-defined functions are provided in the SYSFUN and SYSPROC schemas.

A UDF can be a column function only if it is sourced on an existing column function. A UDF is referenced by means of a qualified or unqualified function name, followed by parentheses enclosing the function arguments (if any). A user-defined column or scalar function registered with the database can be referenced in the same contexts in which any built-in function can appear. A user-defined row function can be referenced only implicitly when registered as a transform function for a user-defined type. A user-defined table function registered with the database can be referenced only in the FROM clause of a SELECT statement.

Function arguments must correspond in number and position to the parameters specified for the user-defined function when it was registered with the database. In addition, the arguments must be of data types that are promotable to the data types of the corresponding defined parameters.

The result of the function is specified in the RETURNS clause. The RETURNS clause, defined when the UDF was registered, determines whether or not a function is a table function. If the RETURNS NULL ON NULL INPUT clause is specified (or defaulted to) when the function is registered, the result is null if any

argument is null. In the case of table functions, this is interpreted to mean a return table with no rows (that is, an empty table).

Following are some examples of user-defined functions:

- A scalar UDF called ADDRESS extracts the home address from resumes stored in script format. The ADDRESS function expects a CLOB argument and returns a VARCHAR(4000) value:

```
SELECT EMPNO, ADDRESS(RESUME) FROM EMP_RESUME
WHERE RESUME_FORMAT = 'SCRIPT'
```

- Table T2 has a numeric column A. Invoking the scalar UDF called ADDRESS from the previous example:

```
SELECT ADDRESS(A) FROM T2
```

raises an error (SQLSTATE 42884), because no function with a matching name and with a parameter that is promotable from the argument exists.

- A table UDF called WHO returns information about the sessions on the server machine that were active at the time that the statement is executed. The WHO function is invoked from within a FROM clause that includes the keyword TABLE and a mandatory correlation variable. The column names of the WHO() table were defined in the CREATE FUNCTION statement.

```
SELECT ID, START_DATE, ORIG_MACHINE
FROM TABLE(WHO()) AS QQ
WHERE START_DATE LIKE 'MAY%'
```

## User-defined functions

---

## Chapter 4. Procedures

---

### Procedures overview

A procedure is an application program that can be started through the SQL CALL statement. The procedure is specified by a procedure name, which may be followed by arguments that are enclosed within parentheses.

The argument or arguments of a procedure are individual scalar values, which can be of different types and can have different meanings. The arguments can be used to pass values into the procedure, receive return values from the procedure, or both.

User-defined procedures are procedures that are registered to a database in SYSCAT.ROUTINES, using the CREATE PROCEDURE statement. One such set of functions is provided with the database manager, in a schema called SYSFUN, and another in a schema called SYSPROC.

Procedures can be qualified with the schema name.

---

### XSR\_ADDSCHEMADOC procedure

```
► XSR_ADDSCHEMADOC (—rschema—, —name—, —schemalocation—, —content—, —
► docproperty—)
```

The schema is SYSPROC.

Each XML schema in the XML schema repository (XSR) can consist of one or more XML schema documents. Where an XML schema consists of multiple documents, the XSR\_ADDSCHEMADOC stored procedure is used to add every XML schema other than the primary XML schema document.

#### Authorization

The authorization ID of the caller of the procedure must be the owner of the XSR object as recorded in the catalog view SYSCAT.XSROBJECTS.

##### *rschema*

An input argument of type VARCHAR (128) that specifies the SQL schema for the XML schema. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR, which is to be moved to the complete state. (The other part of the SQL identifier is supplied by the name argument.) This argument can have a NULL value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

##### *name*

An input argument of type VARCHAR (128) that specifies the name of the

## XSR\_ADDSCHEMADOC procedure

XML schema. The complete SQL identifier for the XML schema is *rschema.name*. The XML schema name must already exist as a result of calling the XSR\_REGISTER stored procedure, and XML schema registration cannot yet be completed. This argument cannot have a NULL value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

### *schemalocation*

An input argument of type VARCHAR (1000), which can have a NULL value, that indicates the schema location of the primary XML schema document to which the XML schema document is being added. This argument is the external name of the XML schema, that is, the primary document can be identified in the XML instance documents with the xsi:schemaLocation attribute.

### *content*

An input parameter of type BLOB (30M) that contains the content of the XML schema document being added. This argument cannot have a NULL value; an XML schema document must be supplied.

### *docproperty*

An input parameter of type BLOB (5M) that indicates the properties for the XML schema document being added. This parameter can have a NULL value; otherwise, the value is an XML document.

### Example:

```
CALL SYSPROC.XSR_ADDSCHEMADOC (
 'user1',
 'POschema',
 'http://myPOschema/address.xsd',
 :content_host_var,
 0)
```

---

## XSR\_COMPLETE procedure

```
►► XSR_COMPLETE (—rschema—, —name—, —schemaproperties—, —————►
► —isusedfordecomposition—) —————►►
```

The schema is SYSPROC.

The XSR\_COMPLETE procedure is the final stored procedure to be called as part of the XML schema registration process, which registers XML schemas with the XML schema repository (XSR). An XML schema is not available for validation until the schema registration completes through a call to this stored procedure.

### **Authorization:**

The authorization ID of the caller of the procedure must be the owner of the XSR object as recorded in the catalog view SYSCAT.XSROBJECTS.

### *rschema*

An input argument of type VARCHAR (128) that specifies the SQL schema for the XML schema. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR, which is to be moved to the complete state. (The other part of the SQL identifier is supplied by the name argument.) This argument can have a NULL value, which indicates that the default SQL



schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

*name*

An input argument of type VARCHAR (128) that specifies the name of the XML schema. The complete SQL identifier for the XML schema, for which a completion check is to be performed, is *rschema.name*. The XML schema name must already exist as a result of calling the XSR\_REGISTER stored procedure, and XML schema registration cannot yet be completed. This argument cannot have a NULL value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

*schemaproperties*

An input argument of type BLOB (5M) that specifies properties, if any, associated with the XML schema. The value for this argument is either NULL, if there are no associated properties, or an XML document representing the properties for the XML schema.

*isusedfordecomposition*

An input parameter of type integer that indicates if an XML schema is to be used for decomposition. If an XML schema is to be used for decomposition, this value should be set to 1; otherwise, it should be set to zero.

## Example:

```
CALL SYSPROC.XSR_COMPLETE(
 'user1',
 'POschema',
 :schemaproperty_host_var,
 0)
```

**XSR\_DTD procedure**

►► XSR\_DTD (—rschema—, —name—, —systemid—, —publicid—, —content—) ◀◀

The schema is SYSPROC.

The XSR\_DTD procedure registers a document type declaration (DTD) with the XML schema repository (XSR).

**Authorization**

The authorization ID of the caller of the procedure must have at least one of the following:

- SYSADM or DBADM authority.
- IMPLICIT\_SCHEMA database authority if the SQL schema does not exist.
- CREATEIN privilege if the SQL schema exists.

*rschema*

An input and output argument of type VARCHAR (128) that specifies the SQL schema for the DTD. The SQL schema is one part of the SQL identifier used to identify this DTD in the XSR. (The other part of the SQL identifier is supplied by the *name* argument.) This argument can have a NULL value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA

## XSR\_DTD procedure

special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. Relational schemas that begin with the string 'SYS' must not be used for this value. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

### *name*

An input and output argument of type VARCHAR (128) that specifies the name of the DTD. The complete SQL identifier for the DTD is *rschema.name* and should be unique among all objects in the XSR. This argument accepts a NULL value. When a NULL value is provided for this argument, a unique value is generated and stored within the XSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

### *systemid*

An input parameter of type VARCHAR (1000) that specifies the system identifier of the DTD. The system ID of the DTD should match the uniform resource identifier of the DTD in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration (as prefixed by the SYSTEM keyword, if used). This argument cannot have a NULL value. The system ID can be specified together with a public ID.

### *publicid*

An input parameter of type VARCHAR (1000) that specifies the public identifier of the DTD. The public ID of a DTD should match the uniform resource identifier of the DTD in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration (as prefixed by the PUBLIC keyword, if used). This argument accepts a NULL value and should be used only if also specified in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration.

### *content*

An input parameter of type BLOB (30M) that contains the content of the DTD document. This argument cannot have a NULL value.

Example: Register the DTD identified by the system ID *http://www.test.com/person.dtd* and public ID *http://www.test.com/person*:

```
CALL SYSPROC.XSR_DTD ('MYDEPT' ,
 'PERSONDTD' ,
 'http://www.test.com/person.dtd' ,
 'http://www.test.com/person' ,
 :content_host_variable
)
```

---

## XSR\_EXTENTITY procedure

►►—XSR\_EXTENTITY—(—*rschema*—,—*name*—,—*systemid*—,—*publicid*—,——————►  
►—*content*—)—————►◄

The schema is SYSPROC.

The XSR\_EXTENTITY procedure registers an external entity with the XML schema repository (XSR).

## Authorization

The authorization ID of the caller of the procedure must have at least one of the following:

- SYSADM or DBADM authority.
- IMPLICIT\_SCHEMA database authority if the SQL schema does not exist.
- CREATEIN privilege if the SQL schema exists.

### *rschema*

An input and output argument of type VARCHAR (128) that specifies the SQL schema for the external entity. The SQL schema is one part of the SQL identifier used to identify this external entity in the XSR. (The other part of the SQL identifier is supplied by the *name* argument.) This argument can have a NULL value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. Relational schemas that begin with the string 'SYS' must not be used for this value. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

### *name*

An input and output argument of type VARCHAR (128) that specifies the name of the external entity. The complete SQL identifier for the external entity is *rschema.name* and should be unique among all objects in the XSR. This argument accepts a NULL value. When a NULL value is provided for this argument, a unique value is generated and stored within the XSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

### *systemid*

An input parameter of type VARCHAR (1000) that specifies the system identifier of the external entity. The system ID of the external entity should match the uniform resource identifier of the external entity in the ENTITY declaration (as prefixed by the SYSTEM keyword, if used). This argument cannot have a NULL value. The system ID can be specified together with a public ID.

### *publicid*

An input parameter of type VARCHAR (1000) that specifies the public identifier of the external entity. The public ID of a external entity should match the uniform resource identifier of the external entity in the ENTITY declaration (as prefixed by the PUBLIC keyword, if used). This argument accepts a NULL value and should be used only if also specified in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration.

### *content*

An input parameter of type BLOB (30M) that contains the content of the external entity document. This argument cannot have a NULL value.

Example: Register the external entities identified by the system identifiers <http://www.test.com/food/chocolate.txt> and <http://www.test.com/food/cookie.txt>:

```
CALL SYSPROC.XSR_EXTENTITY ('FOOD' ,
 'CHOCOLATE' ,
 'http://www.test.com/food/chocolate.txt' ,
 NULL ,
 :content_of_chocolate.txt_as_a_host_variable
)
```

## XSR\_EXTENTITY procedure

```
CALL SYSPROC.XSR_EXTENTITY ('FOOD' ,
 'COOKIE' ,
 'http://www.test.com/food/cookie.txt' ,
 NULL ,
 :content_of_cookie.txt_as_a_host_variable
)
```

---

## XSR\_REGISTER procedure

```
► XSR_REGISTER (rschema , name , schemalocation , content , docproperty)
```

The schema is SYSPROC.

The XSR\_REGISTER procedure is the first stored procedure to be called as part of the XML schema registration process, which registers XML schemas with the XML schema repository (XSR).

### Authorization

The authorization ID of the caller of the procedure must have at least one of the following:

- SYSADM or DBADM authority.
- IMPLICIT\_SCHEMA database authority if the SQL schema does not exist.
- CREATEIN privilege if the SQL schema exists.

#### *rschema*

An input and output argument of type VARCHAR (128) that specifies the SQL schema for the XML schema. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR. (The other part of the SQL identifier is supplied by the name argument.) This argument can have a NULL value, which indicates that the default SQL schema, as defined in the CURRENT\_SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. Relational schemas that begin with the string 'SYS' must not be used for this value. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

#### *name*

An input and output argument of type VARCHAR (128) that specifies the name of the XML schema. The complete SQL identifier for the XML schema is *rschema.name* and should be unique among all objects in the XSR. This argument accepts a NULL value. When a NULL value is provided for this argument, a unique value is generated and stored within the XSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

#### *schemalocation*

An input argument of type VARCHAR (1000), which can have a NULL value, that indicates the schema location of the primary XML schema document. This argument is the external name of the XML schema, that is, the primary document can be identified in the XML instance documents with the xsi:schemaLocation attribute.

*content*

An input parameter of type BLOB (30M) that contains the content of the primary XML schema document. This argument cannot have a NULL value; an XML schema document must be supplied.

*docproperty*

An input parameter of type BLOB (5M) that indicates the properties for the primary XML schema document. This parameter can have a NULL value; otherwise, the value is an XML document.

Example: The following example shows how to call the XSR\_REGISTER procedure from the command line:

```
CALL SYSPROC.XSR_REGISTER(
 'user1',
 'POschema',
 'http://myPOschema/PO.xsd',
 :content_host_var,
 :docproperty_host_var)
```

Example: The following example shows how to call the XSR\_REGISTER procedure from a Java application program:

```
stmt = con.prepareStatement("CALL SYSPROC.XSR_REGISTER (?, ?, ?, ?, ?)");
String xsrObjectName = "myschema1";
String xmlSchemaLocation = "po.xsd";
stmt.setNull(1, java.sql.Types.VARCHAR);
stmt.setString(2, xsrObjectName);
stmt.setString(3, xmlSchemaLocation);
stmt.setBinaryStream(4, buffer, (int)length);
stmt.setNull(5, java.sql.Types.BLOB);
stmt.registerOutParameter(1, java.sql.Types.VARCHAR);
stmt.registerOutParameter(2, java.sql.Types.VARCHAR);
stmt.execute();
```

---

## XSR\_UPDATE procedure

```
►► XSR_UPDATE (—rschema1—, —name1—, —rschema2—, —name2—, —————►
►—dropnewschema—) —————►◄
```

The schema is SYSPROC.

The XSR\_UPDATE stored procedure is used to evolve an existing XML schema in the XML schema repository (XSR). This enables you to modify or extend an existing XML schema so that it can be used to validate both already existing and newly inserted XML documents.

The original XML schema and the new XML schema specified as arguments to XSR\_UPDATE must both be registered and completed in the XSR before the procedure is called. These XML schemas must also be compatible. For details about the compatibility requirements see *Compatibility requirements for evolving an XML schema*.

### Authorization

The privileges held by the authorization ID of the caller of the procedure must include at least one of the following:

## XSR\_UPDATE procedure

- OWNER of the XML schema specified by the SQL schema *rschema1* and the object name *name1*
- SYSADM or DBADM authority.
- ALTERIN privilege on the SQL schema specified by the *rschema1* argument and, if the *dropnewschema* argument is not equal to zero, DROPIN privilege on the SQL schema specified by the *rschema2* argument.

### *rschema1*

An input argument of type VARCHAR (128) that specifies the SQL schema for the original XML schema to be updated. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR. (The other part of the SQL identifier is supplied by the *name1* argument.) This argument cannot have a NULL value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

### *name1*

An input argument of type VARCHAR (128) that specifies the name of the original XML schema to be updated. The complete SQL identifier for the XML schema is *rschema1.name1* . This XML schema must already be registered and completed in the XSR. This argument cannot have a NULL value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

### *rschema2*

An input argument of type VARCHAR (128) that specifies the SQL schema for the new XML schema that will be used to update the original XML schema. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR. (The other part of the SQL identifier is supplied by the *name2* argument.) This argument cannot have a NULL value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

### *name2*

An input argument of type VARCHAR (128) that specifies the name of the new XML schema that will be used to update the original XML schema. The complete SQL identifier for the XML schema is *rschema2.name2*. This XML schema must already be registered and completed in the XSR. This argument cannot have a NULL value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

### *dropnewschema*

An input parameter of type integer that indicates whether the new XML schema should be dropped after it is used to update the original XML schema. Setting this parameter to any non-zero value will cause the new XML schema to be dropped. This argument cannot have a null value.

### Example:

```
CALL SYSPROC.XSR_UPDATE(
 'STORE',
 'PROD',
 'STORE',
 'NEWPROD',
 1)
```

The contents of the XML schema STORE.PROD is updated with the contents of STORE.NEWPROD, and the XML schema STORE.NEWPROD is dropped.

---

## Chapter 5. SQL queries

A *query* specifies a result table. A query is a component of certain SQL statements. The three forms of a query are:

- subselect
- fullselect
- select-statement.

### Authorization

For each table, view, or nickname referenced in the query, the authorization ID of the statement must have at least one of the following:

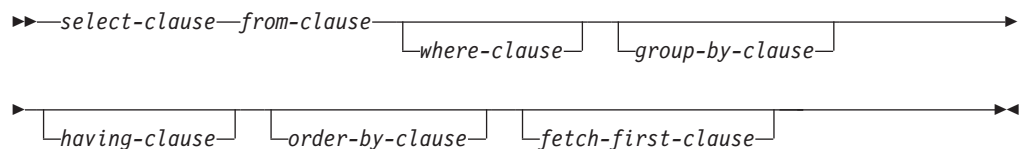
- SYSADM or DBADM authority
- CONTROL privilege
- SELECT privilege.

Group privileges, with the exception of PUBLIC, are not checked for queries that are contained in static SQL statements.

For nicknames, authorization requirements of the data source for the object referenced by the nickname are applied when the query is processed. The authorization ID of the statement may be mapped to a different authorization ID at the data source.

---

## Subselect



The *subselect* is a component of the fullselect.

A subselect specifies a result table derived from the tables, views or nicknames identified in the FROM clause. The derivation can be described as a sequence of operations in which the result of each operation is input for the next. (This is only a way of describing the subselect. The method used to perform the derivation can be quite different from this description. If portions of the subselect do not actually need to be executed for the correct result to be obtained, they might or might not be executed.)

The clauses of the subselect are processed in the following sequence:

1. FROM clause
2. WHERE clause
3. GROUP BY clause
4. HAVING clause
5. SELECT clause
6. ORDER BY clause





representations of the same number are considered equal. For example, -0 is equal to +0 and 2.0 is equal to 2.00. Each of the decimal floating-point special values are also considered equal: -NAN equals -NAN, -SNAN equals -SNAN, -INFINITY equals -INFINITY, INFINITY equals INFINITY, SNAN equals SNAN, and NAN equals NAN.

When the data type of a column is decimal floating-point, and multiple representations of the same number exist in the column, the particular value that is returned for a SELECT DISTINCT can be any one of the representations in the column. For more information, see “Numeric comparisons” on page 100.

For compatibility with other SQL implementations, UNIQUE can be specified as a synonym for DISTINCT.

## Select list notation

- \* Represents a list of names that identify the columns of table R, excluding any columns defined as IMPLICITLY HIDDEN. The first name in the list identifies the first column of R, the second name identifies the second column of R, and so on.

The list of names is established when the program containing the SELECT clause is bound. Hence \* (the asterisk) does not identify any columns that have been added to a table after the statement containing the table reference has been bound.

### *expression*

Specifies the values of a result column. Can be any expression that is a valid SQL language element, but commonly includes column names. Each column name used in the select list must unambiguously identify a column of R.

### *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*, excluding any columns defined as IMPLICITLY HIDDEN. 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.

## Subselect

The number of columns in the result of SELECT is the same as the number of expressions in the operational form of the select list (that is, the list established when the statement is prepared), and cannot exceed 500 for a 4K page size or 1012 for an 8K, 16K, or 32K page size.

### Limitations on string columns

For limitations on the select list, see “Restrictions Using Varying-Length Character Strings”.

### 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 expression or column-name unambiguously identifies a column of R (see “group-by-clause” on page 464) 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 aggregate 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, an SQL parameter, an SQL variable, or a global variable
- A result of a set operation if at least one of the corresponding items in the select list is nullable
- An arithmetic expression or view column that is derived from an arithmetic expression and the database is configured with DFT\_SQLMATHWARN set to Yes
- A scalar subselect
- A dereference operation
- A GROUPING SETS *grouping-expression*

### Names of result columns

- If the AS clause is specified, the name of the result column is the name specified on the AS clause.
- If the AS clause is not specified and a column list is specified in the correlation clause, the name of the result column is the corresponding name in the correlation column list.
- If neither an AS clause nor a column list in the correlation clause is specified and the result column is derived only from a single column (without any functions or operators), then the result column name is the unqualified name of that column.
- If neither an AS clause nor a column list in the correlation clause is specified and the result column is derived only from a single SQL variable or SQL parameter (without any functions or operators), then the result column name is the unqualified name of that SQL variable or SQL parameter.
- If neither an AS clause nor a column list in the correlation clause is specified and 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. The system assigns temporary numbers (as character strings) to these columns.

### Data types of result columns

Each column of the result of SELECT acquires a data type from the expression from which it is derived.

When the expression is ...	The data type of the result column is ...
the name of any numeric column	the same as the data type of the column, with the same precision and scale for DECIMAL columns, or the same precision for DECFLOAT columns.
an integer constant	INTEGER.
a decimal constant	DECIMAL, with the precision and scale of the constant.
a floating-point constant	DOUBLE.
a decimal floating-point constant	DECFLOAT(34)

## Subselect

When the expression is ...	The data type of the result column is ...
the name of any numeric variable	the same as the data type of the variable, with the same precision and scale for DECIMAL variables, or the same precision for DECFLOAT variables.
a hexadecimal constant representing <i>n</i> bytes	VARCHAR( <i>n</i> ); the code page is the database code page.
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 <i>n</i>	VARCHAR( <i>n</i> ).
a graphic string constant of length <i>n</i>	VARGRAPHIC( <i>n</i> ).
the name of a datetime column	the same as the data type of the column.
the name of a user-defined type column	the same as the data type of the column.
the name of a reference type column	the same as the data type of the column.

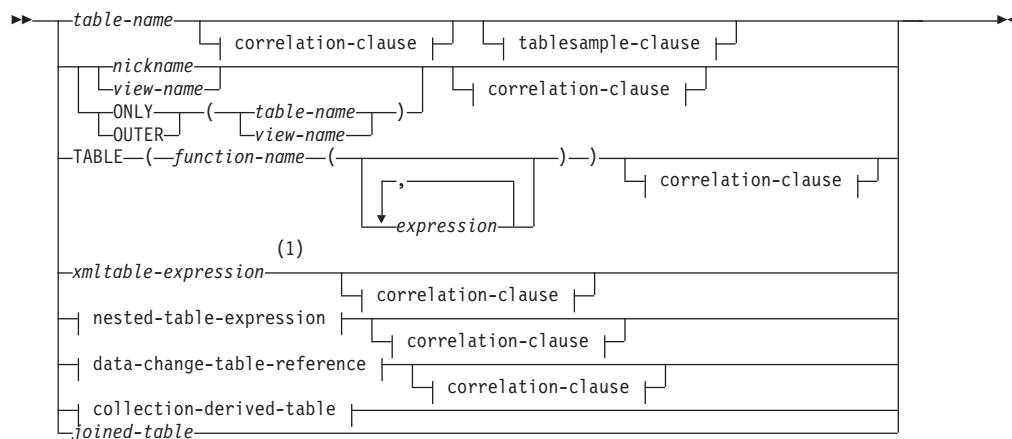
## from-clause



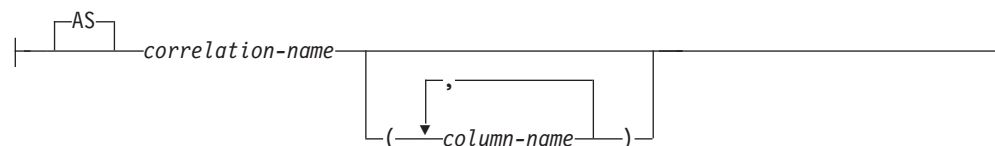
The FROM clause specifies an intermediate result table.

If only one *table-reference* is specified, the intermediate result table is 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-reference* (the Cartesian product). Each row of the result is a row from the first *table-reference* concatenated with a row from the second *table-reference*, concatenated in turn with a row from the third, and so on. The number of rows in the result is the product of the number of rows in all the individual table references. For a description of *table-reference*, see “table-reference.”

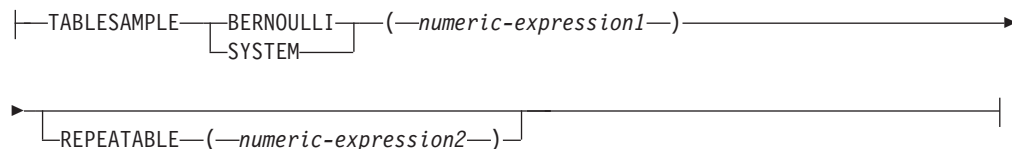
## table-reference



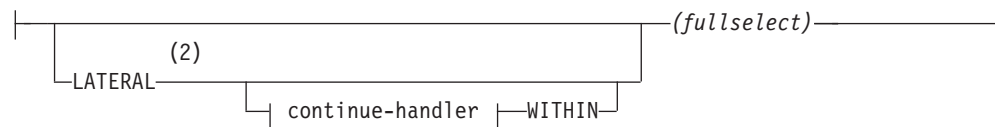
**correlation-clause:**



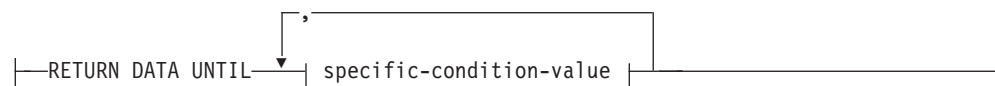
**tablesample-clause:**



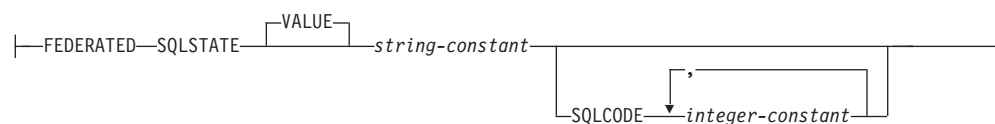
**nested-table-expression:**



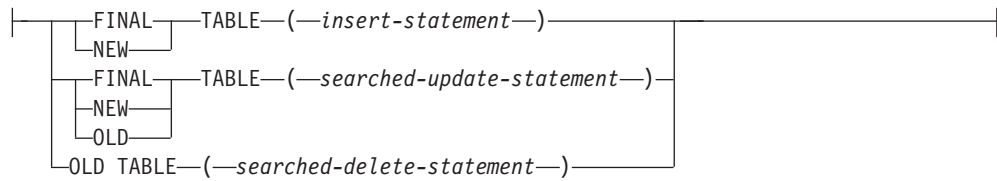
**continue-handler:**



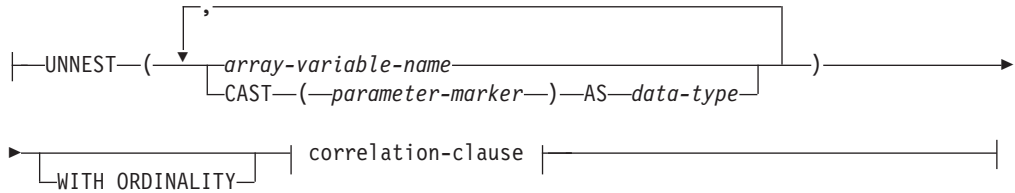
**specific-condition-value:**



**data-change-table-reference:**



**collection-derived-table:**



**Notes:**

- 1 An XMLTABLE expression can be part of a table-reference. In this case, subexpressions within the XMLTABLE expression are in-scope of prior range variables in the FROM clause. For more information, see the description of “XMLTABLE”.
- 2 TABLE can be specified in place of LATERAL.

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 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 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 461.

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*

If a *correlation-clause* does not follow a *function-name* reference, *xmltable-expression*, nested table expression, or *data-change-table-reference*, there is no exposed name for that table reference.

Each *correlation-name* is defined as a designator of the immediately preceding *table-name*, *view-name*, *nickname*, *function-name* reference, *xmltable-expression*, nested table expression, or *data-change-table-reference*. Any qualified reference to a column must use the exposed name. If the same table name, view or nickname 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-names* can also be specified to give names to the columns of the table reference. If the *correlation-clause* does not include *column-names*, the exposed column names are determined as follows:

- Column names of the referenced table, view, or nickname when the *table-reference* is a *table-name*, *view-name*, *nickname*, or *alias-name*
- Column names specified in the RETURNS clause of the CREATE FUNCTION statement when the *table-reference* is a *function-name* reference
- Column names specified in the COLUMNS clause of the *xmltable-expression* when the *table-reference* is an *xmltable-expression*
- Column names exposed by the fullselect when the *table-reference* is a *nested-table-expression*
- Column names from the target table of the data change statement, along with any defined INCLUDE columns when the *table-reference* is a *data-change-table-reference*

In general, collection-derived tables, table functions, and nested table expressions can be specified on any from-clause. Columns from the table functions, nested table expressions, or collection-derived tables can be referenced in the select list and in the rest of the subselect using the correlation name. The scope of this correlation name is the same as correlation names for other tables, views, or nicknames in the FROM clause. A nested table expression can be used:

## Subselect

- 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

A collection-derived table can be used to unnest the elements of arrays into rows. If more than one array is specified, the first array provides the first column in the result table, the second array provides the second column, and so on. If `WITH ORDINALITY` is specified, an extra column of type `INTEGER`, which contains the position of the elements in the arrays, is appended. If the cardinalities of the arrays are not identical, the cardinality of the resulting table is the same as the array with the largest cardinality. The column values in the table are set to the null value for all rows whose subindex value is greater than the cardinality of the corresponding array. In other words, if each array is viewed as a table with two columns (one for the subindices and one for the data), then `UNNEST` performs an `OUTER JOIN` among the arrays, using equality on the subindices as a join predicate.

`UNNEST` can be specified only within an SQL procedure (SQLSTATE 42887).

An expression in the select list of a nested table expression that is referenced within, or is the target of, a data change statement within a fullselect is only valid when it does not include:

- A function that reads or modifies SQL data
- A function that is non-deterministic
- A function that has external action
- An OLAP function

If a view is referenced directly in, or as the target of a nested table expression in a data change statement within a `FROM` clause, the view must either be symmetric (have `WITH CHECK OPTION` specified) or satisfy the restriction for a `WITH CHECK OPTION` view.

If the target of a data change statement within a `FROM` clause is a nested table expression, the modified rows are not requalified, `WHERE` clause predicates are not re-evaluated, and `ORDER BY` or `FETCH FIRST` operations are not redone.

The optional `tablesample-clause` can be used to obtain a random subset (a sample) of the rows from the specified *table-name*, rather than the entire contents of that *table-name*, for this query. This sampling is in addition to any predicates that are specified in the *where-clause*. Unless the optional `REPEATABLE` clause is specified, each execution of the query will usually yield a different sample, except in degenerate cases where the table is so small relative to the sample size that any sample must return the same rows. The size of the sample is controlled by the *numeric-expression1* in parentheses, representing an approximate percentage (P) of the table to be returned. The method by which the sample is obtained is specified after the `TABLESAMPLE` keyword, and can be either `BERNOULLI` or `SYSTEM`. For both methods, the exact number of rows in the sample may be different for each execution of the query, but on average should be approximately P percent of the table, before any predicates further reduce the number of rows.

The *table-name* must be a stored table. It can be a materialized query table (MQT) name, but not a subselect or table expression for which an MQT has been defined, because there is no guarantee that the database manager will route to the MQT for that subselect.



Semantically, sampling of a table occurs before any other query processing, such as applying predicates or performing joins. Repeated accesses of a sampled table within a single execution of a query (such as in a nested-loop join or a correlated subquery) will return the same sample. More than one table may be sampled in a query.

BERNOULLI sampling considers each row individually. It includes each row in the sample with probability  $P/100$  (where  $P$  is the value of *numeric-expression1*), and excludes each row with probability  $1 - P/100$ , independently of the other rows. So if the *numeric-expression1* evaluated to the value 10, representing a ten percent sample, each row would be included with probability 0.1, and excluded with probability 0.9.

SYSTEM sampling permits the database manager to determine the most efficient manner in which to perform the sampling. In most cases, SYSTEM sampling applied to a *table-name* means that each page of *table-name* is included in the sample with probability  $P/100$ , and excluded with probability  $1 - P/100$ . All rows on each page that is included qualify for the sample. SYSTEM sampling of a *table-name* generally executes much faster than BERNOULLI sampling, because fewer data pages need to be retrieved; however, it can often yield less accurate estimates for aggregate functions (SUM(SALES), for example), especially if the rows of *table-name* are clustered on any columns referenced in that query. The optimizer may in certain circumstances decide that it is more efficient to perform SYSTEM sampling as if it were BERNOULLI sampling, for example when a predicate on *table-name* can be applied by an index and is much more selective than the sampling rate  $P$ .

The *numeric-expression1* specifies the size of the sample to be obtained from *table-name*, expressed as a percentage. It must be a constant numeric expression that cannot contain columns, parameter markers, or host variables. The expression must evaluate to a positive number that is less than or equal to 100, but can be between 1 and 0. For example, a value of 0.01 represents one one-hundredth of a percent, meaning that 1 row in 10 000 would be sampled, on average. A *numeric-expression1* that evaluates to 100 is handled as if the *tablesample-clause* were not specified. If *numeric-expression1* evaluates to the null value, or to a value that is greater than 100 or less than 0, an error is returned (SQLSTATE 2202H).

It is sometimes desirable for sampling to be repeatable from one execution of the query to the next; for example, during regression testing or query "debugging". This can be accomplished by specifying the REPEATABLE clause. The REPEATABLE clause requires the specification of a *numeric-expression2* in parentheses, which serves the same role as the seed in a random number generator. Adding the REPEATABLE clause to the *tablesample-clause* of any *table-name* ensures that repeated executions of that query (using the same value for *numeric-expression2*) return the same sample, assuming, of course, that the data itself has not been updated, reorganized, or repartitioned. To guarantee that the same sample of *table-name* is used across multiple queries, use of a global temporary table is recommended. Alternatively, the multiple queries could be combined into one query, with multiple references to a sample that is defined using the WITH clause.

Following are some examples:

*Example 1:* Request a 10% Bernoulli sample of the Sales table for auditing purposes.

```
SELECT * FROM Sales
TABLESAMPLE BERNOULLI(10)
```

## Subselect

*Example 2:* Compute the total sales revenue in the Northeast region for each product category, using a random 1% SYSTEM sample of the Sales table. The semantics of SUM are for the sample itself, so to extrapolate the sales to the entire Sales table, the query must divide that SUM by the sampling rate (0.01).

```
SELECT SUM(Sales.Revenue) / (0.01)
FROM Sales TABLESAMPLE SYSTEM(1)
WHERE Sales.RegionName = 'Northeast'
GROUP BY Sales.ProductCategory
```

*Example 3:* Using the REPEATABLE clause, modify the previous query to ensure that the same (yet random) result is obtained each time the query is executed. (The value of the constant enclosed by parentheses is arbitrary.)

```
SELECT SUM(Sales.Revenue) / (0.01)
FROM Sales TABLESAMPLE SYSTEM(1) REPEATABLE(3578231)
WHERE Sales.RegionName = 'Northeast'
GROUP BY Sales.ProductCategory
```

## Table 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 defined in the CREATE TABLE statement.

- Table Function Resolution

The arguments specified in a table function reference, together with the function name, are used by an algorithm called *function resolution* to determine the exact function to be used. This is no different from what happens with other functions (such as scalar functions) that are used in a statement.

- Table Function Arguments

As with scalar function arguments, table function arguments can in general be any valid SQL expression. The following examples are valid syntax:

```
Example 1: SELECT c1
FROM TABLE(tf1('Zachary')) AS z
WHERE c2 = 'FLORIDA';
```

```
Example 2: SELECT c1
FROM TABLE(tf2 (:hostvar1, CURRENT DATE)) AS z;
```

```
Example 3: SELECT c1
FROM t
WHERE c2 IN
 (SELECT c3 FROM
 TABLE(tf5(t.c4)) AS z -- correlated reference
) -- to previous FROM clause
```

- Table Functions That Modify SQL Data

Table functions that are specified with the MODIFIES SQL DATA option can only be used as the last table reference in a *select-statement*, *common-table-expression*, or RETURN statement that is a subselect, a SELECT INTO, or a *row-fullselect* in a SET statement. Only one table function is allowed in one FROM clause, and the table function arguments must be correlated to all other table references in the subselect (SQLSTATE 429BL). The following examples have valid syntax for a table function with the MODIFIES SQL DATA property:

```

Example 1: SELECT c1
 FROM TABLE(tfmod('Jones')) AS z

Example 2: SELECT c1
 FROM t1, t2, TABLE(tfmod(t1.c1, t2.c1)) AS z

Example 3: SET var =
 (SELECT c1
 FROM TABLE(tfmod('Jones')) AS z)

Example 4: RETURN SELECT c1
 FROM TABLE(tfmod('Jones')) AS z

Example 5: WITH v1(c1) AS
 (SELECT c1
 FROM TABLE(tfmod(:hostvar1)) AS z)
 SELECT c1
 FROM v1, t1 WHERE v1.c1 = t1.c1

```

## Error tolerant nested-table-expression

Certain errors that occur within a *nested-table-expression* can be tolerated, and instead of returning an error, the query can continue and return a result.

Specifying the RETURN DATA UNTIL clause will cause any rows that are returned from the fullselect before the indicated condition is encountered to make up the result set from the fullselect. This means that a partial result set (which could also be an empty result set) from the fullselect is acceptable as the result for the *nested-table-expression*.

The FEDERATED keyword restricts the condition to handle only errors that occur at a remote data source.

The condition can be specified as an SQLSTATE value, with a *string-constant* length of 5. You can optionally specify an SQLCODE value for each specified SQLSTATE value. For portable applications, specify SQLSTATE values as much as possible, because SQLCODE values are generally not portable across platforms and are not part of the SQL standard.

Only certain conditions can be tolerated. Errors that do not allow the rest of the query to be executed cannot be tolerated, and an error is returned for the whole query. The *specific-condition-value* might specify conditions that cannot actually be tolerated by the database manager, even if a specific SQLSTATE or SQLCODE value is specified, and for these cases, an error is returned.

The following SQLSTATE values and SQLCODE values have the potential, when specified, to be tolerated by the database manager:

- SQLSTATE 08001; SQLCODEs -1336, -30080, -30081, -30082
- SQLSTATE 08004
- SQLSTATE 42501
- SQLSTATE 42704; SQLCODE -204
- SQLSTATE 42720
- SQLSTATE 28000

A query or view containing an error tolerant *nested-table-expression* is read-only.

## Subselect

The fullselect of an error tolerant *nested-table-expression* is not optimized using materialized query tables.

### 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 correlated reference must be from a *table-reference* at a higher level in the hierarchy of subqueries. This hierarchy includes the table-references that have already been resolved in the left-to-right processing of the FROM clause. For nested table expressions, the LATERAL keyword must 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
 WHERE t.c3 = z.c4; -- in FROM, so t.c2
 -- is known

Example 2: SELECT t.c1, z.c5
 FROM t, TABLE(tf4(2 * t.c2)) AS z -- t precedes tf4
 WHERE t.c3 = z.c4; -- in FROM, so t.c2
 -- is known

Example 3: SELECT d.deptno, d.deptname,
 empinfo.avgsal, empinfo.empcount
 FROM department d,
 LATERAL (SELECT AVG(e.salary) AS avgsal,
 COUNT(*) AS empcount
 FROM employee e -- department precedes
 WHERE e.workdept=d.deptno -- and TABLE is
) AS empinfo; -- specified, so
 -- d.deptno is known
```

But the following examples are not valid:

```
Example 4: SELECT t.c1, z.c5
 FROM TABLE(tf6(t.c2)) AS z, t -- cannot resolve t in t.c2!
 WHERE t.c3 = z.c4; -- compare to Example 1 above.

Example 5: SELECT a.c1, b.c5
 FROM TABLE(tf7a(b.c2)) AS a, TABLE(tf7b(a.c6)) AS b
 WHERE a.c3 = b.c4; -- cannot resolve b in b.c2!

Example 6: SELECT d.deptno, d.deptname,
 empinfo.avgsal, empinfo.empcount
 FROM department d,
 (SELECT AVG(e.salary) AS avgsal,
 COUNT(*) AS empcount
 FROM employee e -- department precedes
 WHERE e.workdept=d.deptno -- but TABLE is not
) AS empinfo; -- specified, so
 -- d.deptno is unknown
```

### Data change table references

A *data-change-table-reference* clause specifies an intermediate result table. This table is based on the rows that are directly changed by the searched UPDATE, searched DELETE, or INSERT statement that is included in the clause. A *data-change-table-reference* can be specified as the only *table-reference* in the FROM clause of the outer fullselect that is used in a *select-statement*, a SELECT INTO statement, or a common table expression. A *data-change-table-reference* can be specified as the only table reference in the only fullselect in a SET Variable statement (SQLSTATE 428FL). The target table or view of the data change

statement is considered to be a table or view that is referenced in the query; therefore, the authorization ID of the query must have SELECT privilege on that target table or view. A *data-change-table-reference* clause cannot be specified in a view definition, materialized query table definition, or FOR statement (SQLSTATE 428FL).

The target of the UPDATE, DELETE, or INSERT statement cannot be a temporary view defined in a common table expression (SQLSTATE 42807).

#### **FINAL TABLE**

Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement as they appear at the completion of the data change statement. If there are AFTER triggers or referential constraints that result in further operations on the table that is the target of the SQL data change statement, an error is returned (SQLSTATE 57058, SQLSTATE 560C6). If the target of the SQL data change statement is a view that is defined with an INSTEAD OF trigger for the type of data change, an error is returned (SQLSTATE 428G3).

#### **NEW TABLE**

Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement prior to the application of referential constraints and AFTER triggers. Data in the target table at the completion of the statement might not match the data in the intermediate result table because of additional processing for referential constraints and AFTER triggers.

#### **OLD TABLE**

Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement as they existed prior to the application of the data change statement.

*(searched-update-statement)*

Specifies a searched UPDATE statement. A WHERE clause or a SET clause in the UPDATE statement cannot contain correlated references to columns outside of the UPDATE statement.

*(searched-delete-statement)*

Specifies a searched DELETE statement. A WHERE clause in the DELETE statement cannot contain correlated references to columns outside of the DELETE statement.

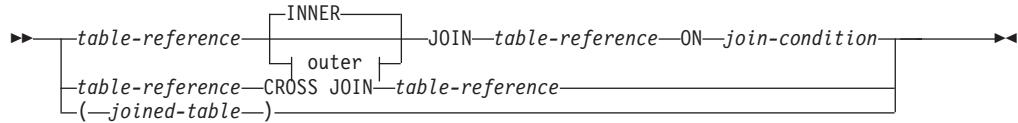
*(insert-statement)*

Specifies an INSERT statement. A fullselect in the INSERT statement cannot contain correlated references to columns outside of the fullselect of the INSERT statement.

The content of the intermediate result table for a *data-change-table-reference* is determined when the cursor opens. The intermediate result table contains all manipulated rows, including all the columns in the specified target table or view. All the columns of the target table or view for an SQL data change statement are accessible using the column names from the target table or view. If an INCLUDE clause was specified within a data change statement, the intermediate result table will contain these additional columns.

#### **joined-table**

## Subselect



### outer:



A *joined table* specifies an intermediate result table that is the result of either an inner join or an outer join. The table is derived by applying one of the join operators: CROSS, INNER, LEFT OUTER, RIGHT OUTER, or FULL OUTER to its operands.

Cross joins represent the cross product of the tables, where each row of the left table is combined with every row of the right table. Inner joins can be thought of as the cross product of the tables, keeping only the rows where the join condition is true. The result table might 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:

- **Left outer join** includes rows from the left table that were missing from the inner join.
- **Right outer join** includes rows from the right table that were missing from the inner join.
- **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
- It cannot include an SQL function

- Any column referenced in an expression of the *join-condition* must be a column of one of the operand tables of the associated join (in the scope of the same joined-table clause)
- Any function referenced in an expression of the *join-condition* of a full outer join must be deterministic and have no external action
- It cannot include an XMLQUERY or XMLEXISTS expression

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 conditions.

## 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 CROSS JOIN T2 consists of all possible pairings of their rows.
- The result of T1 INNER JOIN T2 consists of their paired rows where the join-condition is true.
- The result of T1 LEFT OUTER JOIN T2 consists of their paired rows where the join-condition is true and, for each unpaired row of T1, the concatenation of that row with the null row of T2. 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:

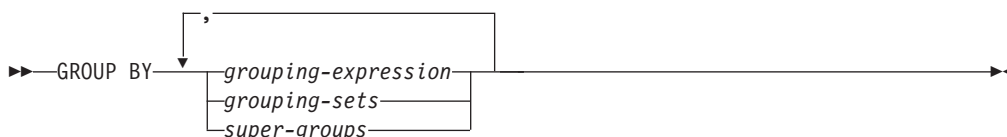
- Each *column-name* must unambiguously identify a column of R or be a correlated reference. A *column-name* is a correlated reference if it identifies a column of a *table-reference* in an outer subselect.

## 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 may be executed just once, whereas a subquery with a correlated reference may have to be executed once for each row.

### group-by-clause



The GROUP BY clause specifies an intermediate result table that consists of a grouping of the rows of R. R is the result of the previous clause of the subselect.

In its simplest form, a GROUP BY clause contains a *grouping expression*. A grouping expression is an *expression* used in defining the grouping of R. Each expression or *column name* included in grouping-expression must unambiguously identify a column of R (SQLSTATE 42702 or 42703). A grouping expression cannot include a scalar fullselect or an XMLQUERY or XMLEXISTS expression (SQLSTATE 42822), or any expression or function that is variant or has an external action (SQLSTATE 42845).

**Note:** The following expressions, which do not contain an explicit column reference, can be used in a *grouping-expression* to identify a column of R:

- ROW CHANGE TIMESTAMP FOR *table-designator*
- ROW CHANGE TOKEN FOR *table-designator*
- RID\_BIT or RID scalar function

More complex forms of the GROUP BY clause include *grouping-sets* and *super-groups*. For a description of these forms, see “grouping-sets” on page 465 and “super-groups” on page 466, 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.

If a *grouping-expression* contains decimal floating-point columns, and multiple representations of the same number exist in these columns, the number that is returned can be any of the representations of the number.

A *grouping-expression* can be used in a search condition in a HAVING clause, in an expression in a SELECT clause or in a *sort-key-expression* of an ORDER BY clause (see “order-by-clause” on page 471 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

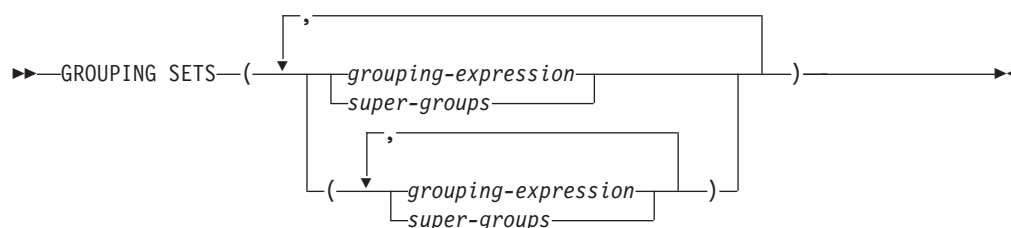


parentheses are used to ensure that the corresponding expression is evaluated in the same order. Thus,  $3+(col1+col2)$  would also be allowed in the select list. If the concatenation operator is used, the *grouping-expression* must be used exactly as the expression was specified in the select list.

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.

### 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 466.

Note that grouping sets are the fundamental building blocks for GROUP BY operations. A simple GROUP BY with a single column can be considered a grouping set with one element. For example:

**GROUP BY a**

is the same as

**GROUP BY GROUPING SETS((a))**

and

**GROUP BY a,b,c**

is the same as

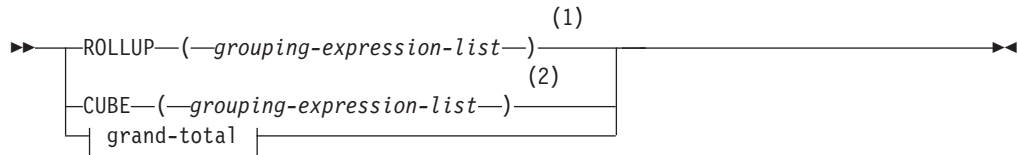
## Subselect

**GROUP BY GROUPING SETS((a,b,c))**

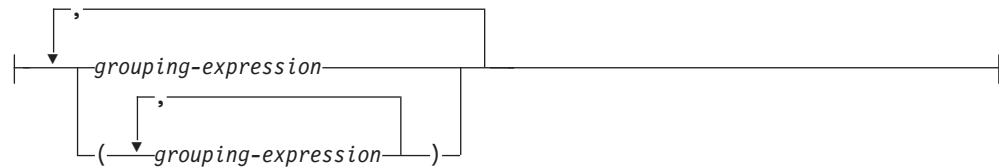
Non-aggregation columns from the select list of the subselect that are excluded from a grouping set will return a null for such columns for each row generated for that grouping set. This reflects the fact that aggregation was done without considering the values for those columns.

Example C2 through Example C7 illustrate the use of grouping sets.

### super-groups



### grouping-expression-list:



### grand-total:



### Notes:

- 1 Alternate specification when used alone in group-by-clause is: grouping-expression-list WITH ROLLUP.
- 2 Alternate specification when used alone in group-by-clause is: grouping-expression-list WITH CUBE.

### ROLLUP ( grouping-expression-list )

A *ROLLUP grouping* is an extension to the GROUP BY clause that produces a result set containing *sub-total* rows in addition to the “regular” grouped rows. *Sub-total* rows are “super-aggregate” rows that contain further aggregates whose values are derived by applying the same column functions that were used to obtain the grouped rows. These rows are called sub-total rows, because that is their most common use; however, any column function can be used for the aggregation. For instance, MAX and AVG are used in Example C8.

A ROLLUP grouping is a series of *grouping-sets*. The general specification of a ROLLUP with *n* elements

**GROUP BY ROLLUP(C<sub>1</sub>, C<sub>2</sub>, . . . , C<sub>n-1</sub>, C<sub>n</sub>)**

is equivalent to

```

GROUP BY GROUPING SETS((C1,C2,...,Cn-1,Cn)
 (C1,C2,...,Cn-1)
 ...
 (C1,C2)
 (C1)
 ())

```

Note that the  $n$  elements of the ROLLUP translate to  $n+1$  grouping sets. Note also that the order in which the *grouping-expressions* is specified is significant for ROLLUP. For example:

```
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 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.

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 example, a specification of:

```
GROUP BY CUBE(a,b,c)
```

is equivalent to:

```

GROUP BY GROUPING SETS((a,b,c)
 (a,b)
 (a,c)
 (b,c)
 (a)
 (b)
 (c)
 ())

```

Note that the three elements of the CUBE translate into eight grouping sets.

The order of specification of elements does not matter for CUBE. 'CUBE (DayOfYear, Sales\_Person)' and 'CUBE (Sales\_Person, DayOfYear)' yield the same result sets. The use of the word 'same' applies to content of the result set, not to its order. The ORDER BY clause is the only way to guarantee the order of the rows in the result set. Example C4 illustrates the use of CUBE.

#### *grouping-expression-list*

A *grouping-expression-list* is used within a CUBE or ROLLUP clause to define

## Subselect

the number of elements in the CUBE or ROLLUP operation. This is controlled by using parentheses to delimit elements with multiple *grouping-expressions*.

The rules for a *grouping-expression* are described in “group-by-clause” on page 464. For example, suppose that a query is to return the total expenses for the ROLLUP of City within a Province but not within a County. However, the clause:

```
GROUP BY ROLLUP(Province, County, City)
```

results in unwanted subtotal rows for the County. In the clause:

```
GROUP BY ROLLUP(Province, (County, City))
```

the composite (County, City) forms one element in the ROLLUP and, therefore, a query that uses this clause will yield the desired result. In other words, the two-element ROLLUP:

```
GROUP BY ROLLUP(Province, (County, City))
```

generates:

```
GROUP BY GROUPING SETS((Province, County, City)
 (Province)
 ())
```

and the three-element ROLLUP generates:

```
GROUP BY GROUPING SETS((Province, County, City)
 (Province, County)
 (Province)
 ())
```

Example C2 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 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)
```

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:

## Subselect

```
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 to be 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.

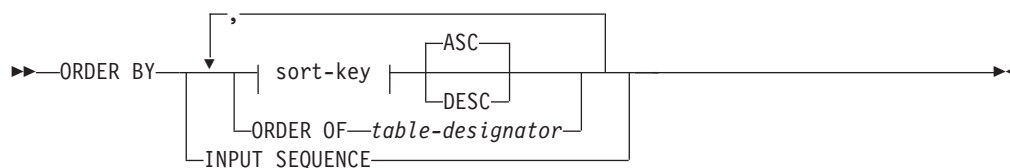
If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a group of R, and the results used in applying the search condition. In actuality, the subquery is executed for each group only if it contains a correlated reference. For an illustration of the difference, see Example A6 and Example A7.

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 include column names when they are arguments to an aggregate function, correlated column references, global variables, host variables, literals, special registers, SQL variables, or SQL parameters.

**Note:** The following expressions can only be specified in a HAVING clause if they are contained within an aggregate function (SQLSTATE 42803):

- ROW CHANGE TIMESTAMP FOR *table-designator*
- ROW CHANGE TOKEN FOR *table-designator*
- RID\_BIT or RID scalar function

**order-by-clause****sort-key:**

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. Each *sort-key* cannot have a data type of LONG VARCHAR, CLOB, LONG VARGRAPHIC, DBCLOB, BLOB, distinct type on any of these types, or structured type (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 a *simple-integer* or, in some cases, by a *sort-key-expression* that matches the expression in the select list (see details of *sort-key-expression*). A column is unnamed if the AS clause is not specified and it is derived from a constant, an expression with operators, or a function.

Ordering is performed in accordance with comparison rules. If an ORDER BY clause contains decimal floating-point columns, and multiple representations of the same number exist in these columns, the ordering of the multiple representations of the same number is unspecified. The null value is higher than all other values. If the ORDER BY clause does not completely order the rows, rows with duplicate values of all identified columns are displayed in an arbitrary order.

*simple-column-name*

Usually identifies a column of the result table. In this case, *simple-column-name* must be the column name of a named column in the select list.

The *simple-column-name* can also identify a column name of a table, view, or nested table identified in the FROM clause if the query is a subselect. This includes columns defined as implicitly hidden. An error occurs if the subselect:

- 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 names in sort keys” below.

*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.

## Subselect

### *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), an XMLQUERY or XMLEXISTS expression (SQLSTATE 42822), 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” below.

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.

### ORDER OF *table-designator*

Specifies that the same ordering used in *table-designator* should be applied to the result table of the subselect. There must be a table reference matching *table-designator* in the FROM clause of the subselect that specifies this clause (SQLSTATE 42703). The subselect (or fullselect) corresponding to the specified *table-designator* must include an ORDER BY clause that is dependant on the data (SQLSTATE 428FI). The ordering that is applied is the same as if the columns of the ORDER BY clause in the nested subselect (or fullselect) were included in the outer subselect (or fullselect), and these columns were specified in place of the ORDER OF clause.

Note that this form is not allowed in a fullselect (other than the degenerative form of a fullselect). For example, the following is not valid:

```
(SELECT C1 FROM T1
 ORDER BY C1)
UNION
SELECT C1 FROM T2
 ORDER BY ORDER OF T1
```

The following example *is* valid:

```
SELECT C1 FROM
 (SELECT C1 FROM T1
 UNION
 SELECT C1 FROM T2
 ORDER BY C1) AS UTABLE
ORDER BY ORDER OF UTABLE
```

### INPUT SEQUENCE

Specifies that, for an INSERT statement, the result table will reflect the input order of ordered data rows. INPUT SEQUENCE ordering can only be specified if an INSERT statement is used in a FROM clause (SQLSTATE 428G4). See



“table-reference” on page 452. If INPUT SEQUENCE is specified and the input data is not ordered, the INPUT SEQUENCE clause is ignored.

## Notes

- **Column names in sort keys:**

- The column name is qualified.

The query must be a *subselect* (SQLSTATE 42877). The column name must unambiguously identify a column of some table, view or nested table in the FROM clause of the subselect (SQLSTATE 42702). The value of the column is used to compute the value of the sort specification.

- The column name is unqualified.

- The query is a subselect.

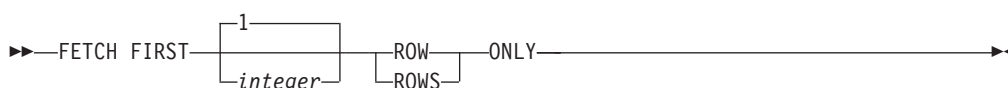
If the column name is identical to the name of more than one column of the result table, the column name must unambiguously identify a column of some table, view or nested table in the FROM clause of the ordering subselect (SQLSTATE 42702). If the column name is identical to one column, that column is used to compute the value of the sort specification. If the column name is not identical to a column of the result table, then it must unambiguously identify a column of some table, view or nested table in the FROM clause of the fullselect in the select-statement (SQLSTATE 42702).

- The query is not a subselect (it includes set operations such as union, except or intersect).

The column name must not be identical to the name of more than one column of the result table (SQLSTATE 42702). The column name must be identical to exactly one column of the result table (SQLSTATE 42707), and this column is used to compute the value of the sort specification.

- **Limits:** The use of a *sort-key-expression* or a *simple-column-name* where the column is not in the select list 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.

## fetch-first-clause



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). The value of *integer* must be a positive integer (not zero).

Use of the *fetch-first-clause* influences query optimization of the subselect or fullselect, based on the fact that at most *integer* rows will be retrieved. If both the *fetch-first-clause* is specified in the outermost fullselect and the *optimize-for-clause* is specified for the select statement, the database manager will use the *integer* from the *optimize-for-clause* to influence query optimization of the outermost fullselect.

## Subselect

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 is used to influence the communications buffer size.

If the fullselect contains an SQL data change statement in the FROM clause, all the rows are modified regardless of the limit on the number of rows to fetch.

### 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')
```

*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

```

*Example A10:* Query the EMP\_ACT table and return those project numbers that have an employee whose salary is in the top 10 of all employees.

## Subselect

```
SELECT EMP_ACT.EMPNO,PROJNO
FROM EMP_ACT
WHERE EMP_ACT.EMPNO IN
 (SELECT EMPLOYEE.EMPNO
 FROM EMPLOYEE
 ORDER BY SALARY DESC
 FETCH FIRST 10 ROWS ONLY)
```

*Example A11:* Assuming that PHONES and IDS are two SQL variables with array values of the same cardinality, turn these arrays into a table with three columns (one for each array and one for the position), and one row per array element.

```
SELECT T.PHONE, T.ID, T.INDEX FROM UNNEST(PHONES, IDS)
WITH ORDINALITY AS T(PHONE, ID, INDEX)
ORDER BY T.INDEX
```

## 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.

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.

```
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. 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

## Subselect

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.

```
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 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 LUCCHESI	3
	13	6 LUCCHESI	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.

## Subselect

*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	LUCCHESI	4
13	6	-	27
13	7	GOUNOT	21
13	7	LEE	21
13	7	LUCCHESI	4
13	7	-	46
13	-	-	73
-	-	-	73

*Example C4:* If you run the same query as Example C3 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	LUCCHESI	4
13	6	-	27
13	7	GOUNOT	21
13	7	LEE	21
13	7	LUCCHESI	4
13	7	-	46
13	-	GOUNOT	32
13	-	LEE	33
13	-	LUCCHESI	8
13	-	-	73
-	6	GOUNOT	11
-	6	LEE	12
-	6	LUCCHESI	4
-	6	-	27
-	7	GOUNOT	21
-	7	LEE	21
-	7	LUCCHESI	4
-	7	-	46
-	-	GOUNOT	32



```

- - LEE 33
- - LUCCHESSI 8
- - - 73

```

*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
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

```

## Subselect

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) 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
```

*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

## Subselect

-	GROUP 2	-	-	3 Quebec	40
-	GROUP 2	-	-	3 -	104
-	GROUP 2	-	-	4 Manitoba	17
-	GROUP 2	-	-	4 Ontario-North	1
-	GROUP 2	-	-	4 Ontario-South	14
-	GROUP 2	-	-	4 Quebec	11
-	GROUP 2	-	-	4 -	43
-	GROUP 2	-	-	12 Manitoba	2
-	GROUP 2	-	-	12 Ontario-South	4
-	GROUP 2	-	-	12 Quebec	2
-	GROUP 2	-	-	12 -	8
-	GROUP 2	-	-	- -	155
GROUP 1	-	-	-	- -	155

*Step 4:* Notice that because R1 and R2 are used in different grouping sets, whenever R1 is non-null in the result, R2 is null and whenever R2 is non-null in the result, R1 is null. That means you can consolidate these columns into a single column using the COALESCE function. You can also use this column in the ORDER BY clause to keep the results of the two grouping sets together.

```

SELECT COALESCE(R1,R2) AS GROUP,
 WEEK(SALES_DATE) AS WEEK,
 DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
 MONTH(SALES_DATE) AS MONTH,
 REGION, SUM(SALES) AS UNITS_SOLD
FROM SALES, (VALUES('GROUP 1','GROUP 2')) AS X(R1,R2)
GROUP BY GROUPING SETS ((R1, ROLLUP(WEEK(SALES_DATE),
 DAYOFWEEK(SALES_DATE))),
 (R2, ROLLUP(MONTH(SALES_DATE), REGION)))
ORDER BY GROUP, WEEK, DAY_WEEK, MONTH, REGION;

```

results in:

GROUP	WEEK	DAY_WEEK	MONTH	REGION	UNITS_SOLD
GROUP 1		13	6	- -	27
GROUP 1		13	7	- -	46
GROUP 1		13	-	- -	73
GROUP 1		14	1	- -	31
GROUP 1		14	2	- -	43
GROUP 1		14	-	- -	74
GROUP 1		53	1	- -	8
GROUP 1		53	-	- -	8
GROUP 1		-	-	- -	155
GROUP 2		-	-	3 Manitoba	22
GROUP 2		-	-	3 Ontario-North	8
GROUP 2		-	-	3 Ontario-South	34
GROUP 2		-	-	3 Quebec	40
GROUP 2		-	-	3 -	104
GROUP 2		-	-	4 Manitoba	17
GROUP 2		-	-	4 Ontario-North	1
GROUP 2		-	-	4 Ontario-South	14
GROUP 2		-	-	4 Quebec	11
GROUP 2		-	-	4 -	43
GROUP 2		-	-	12 Manitoba	2
GROUP 2		-	-	12 Ontario-South	4
GROUP 2		-	-	12 Quebec	2
GROUP 2		-	-	12 -	8
GROUP 2		-	-	- -	155

*Example 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,

```

```

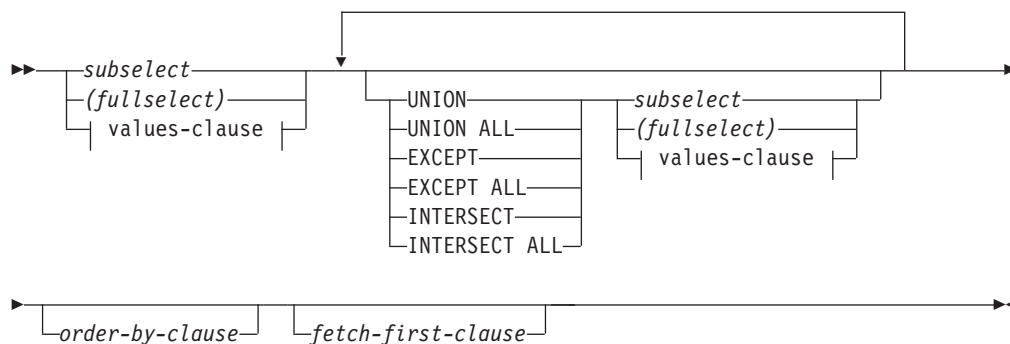
MAX(SALES) AS BEST_SALE,
CAST(ROUND(AVG(DECIMAL(SALES)),2) AS DECIMAL(5,2)) AS AVG_UNITS_SOLD
FROM SALES
GROUP BY CUBE(MONTH(SALES_DATE),REGION)
ORDER BY MONTH, REGION

```

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



### values-clause:



**values-row:**

The *fullselect* is a component of the select-statement, the INSERT statement, and the CREATE VIEW statement. It is also a component of certain predicates which, in turn, are components of a statement. A fullselect that is a component of a predicate is called a *subquery*, and a fullselect that is enclosed in parentheses is sometimes called a subquery.

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 specifications of *values-row*, 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  $n$  specifications of *values-row*,  $RE_1$  to  $RE_n$ , where  $n$  is greater than 1, is equivalent to:

```
RE1 UNION ALL RE2 ... UNION ALL REn
```

This means that the corresponding expressions of each *values-row* must be comparable (SQLSTATE 42825).

**UNION or UNION ALL**

Derives a result table by combining two other result tables (R1 and R2). If UNION ALL is specified, the result consists of all rows in R1 and R2. If UNION is specified without the ALL option, the result is the set of all rows in either R1 or R2, with the duplicate rows eliminated. In either case, however, each row of the UNION table is either a row from R1 or a row from R2.

**EXCEPT or EXCEPT ALL**

Derives a result table by combining two other result tables (R1 and R2). If EXCEPT ALL is specified, the result consists of all rows that do not have a

corresponding row in R2, where duplicate rows are significant. If EXCEPT is specified without the ALL option, the result consists of all rows that are only in R1, with duplicate rows in the result of this operation eliminated.

For compatibility with other SQL implementations, MINUS can be specified as a synonym for EXCEPT.

### INTERSECT or INTERSECT ALL

Derives a result table by combining two other result tables (R1 and R2). If INTERSECT ALL is specified, the result consists of all rows that are in both R1 and R2. If INTERSECT is specified without the ALL option, the result consists of all rows that are in both R1 and R2, with the duplicate rows eliminated.

#### *order-by-clause*

A fullselect that contains an ORDER BY or FETCH FIRST clause cannot be specified in:

- A materialized query table
- The outermost fullselect of a view (SQLSTATE 428FJ).

**Note:** An ORDER BY clause in a fullselect does not affect the order of the rows returned by a query. An ORDER BY clause only affects the order of the rows returned if it is specified in the outermost fullselect.

The number of columns in the result tables R1 and R2 must be the same (SQLSTATE 42826). If the ALL keyword is not specified, R1 and R2 must not include any columns having a data type of LONG VARCHAR, CLOB, LONG VARGRAPHIC, DBCLOB, BLOB, distinct type on any of these types, or structured type (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.

**Duplicate rows:** Two rows are duplicates if each value in the first is equal to the corresponding value of the second. For determining duplicates, two null values are considered equal, and two decimal floating-point representations of the same number are considered equal. For example, 2.00 and 2.0 have the same value (2.00 and 2.0 compare as equal) but have different exponents, which allows you to represent both 2.00 and 2.0. So, for example, if the result table of a UNION operation contains a decimal floating-point column and multiple representations of the same number exist, the one that is returned (for example, 2.00 or 2.0) is unpredictable. For more information, see “Numeric comparisons” on page 100.

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

## Fullselect

R1 and R2.

R1	R2	UNION ALL	UNION	EXCEPT ALL	EXCEPT	INTER- SECT ALL	INTER- SECT
1	1	1	1	1	2	1	1
1	1	1	2	2	5	1	3
1	3	1	3	2		3	4
2	3	1	4	2		4	
2	3	1	5	4			
2	3	2		5			
3	4	2					
4		2					
4		3					
5		3					
		3					
		3					
		3					
		4					
		4					
		4					
		4					
		5					

## 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
 WHERE WORKDEPT LIKE '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)
```

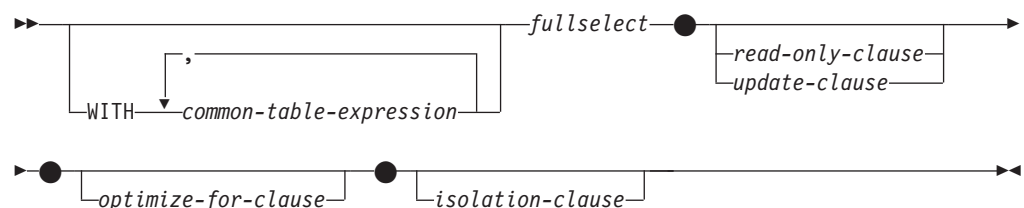
If no NULL values are involved, this example returns the same result as

```
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.

---

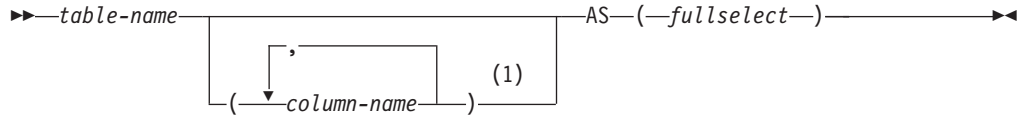
## Select-statement



## 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.

### common-table-expression



#### 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*.

If the fullselect of a common table expression contains a *data-change-table-reference* in the FROM clause, the common table expression is said to modify data. A common table expression that modifies data is always evaluated when the statement is processed, regardless of whether the common table expression is used anywhere else in the statement. If there is at least one common table expression that reads or modifies data, all common table expressions are processed in the order in which they occur, and each common table expression that reads or modifies data is completely executed, including all constraints and triggers, before any subsequent common table expressions are executed.

The common table expression is also optional prior to the fullselect in the CREATE VIEW and INSERT statements.

A common table expression can be used:

- 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
- When multiple SQL data change statements need to be processed within the query

If the fullselect of a common table expression contains a reference to itself in a FROM clause, the common table expression is a *recursive common table expression*. Queries using recursion are useful in supporting applications such as bill of materials (BOM), reservation systems, and network planning.

The following must be true of a recursive common table expression:

- Each fullselect that is part of the recursion cycle must start with SELECT or SELECT ALL. Use of SELECT DISTINCT is not allowed (SQLSTATE 42925). Furthermore, the unions must use UNION ALL (SQLSTATE 42925).
- The column names must be specified following the *table-name* of the common table expression (SQLSTATE 42908).
- The first fullselect of the first union (the initialization fullselect) must not include a reference to any column of the common table expression in any FROM clause (SQLSTATE 42836).
- If a column name of the common table expression is referred to in the iterative fullselect, the data type, length, and code page for the column are determined based on the initialization fullselect. The corresponding column in the iterative fullselect must have the same data type and length as the data type and length determined based on the initialization fullselect and the code page must match (SQLSTATE 42825). However, for character string types, the length of the two data types 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).
- The iterative fullselect and the overall recursive fullselect must not include an order-by-clause (SQLSTATE 42836).
- Subqueries (scalar or quantified) must not be part of any recursion cycles (SQLSTATE 42836).

When developing recursive common table expressions, remember that an infinite recursion cycle (loop) can be created. Check that recursion cycles will 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".

## Select-statement

A warning is issued if this syntax is not found in the recursive common table expression (SQLSTATE 01605).

### Recursion example: bill of materials

Bill of materials (BOM) applications are a common requirement in many business environments. To illustrate the capability of a recursive common table expression for BOM applications, consider a table of parts with associated subparts and the quantity of subparts required by the part. For this example, create the table as follows:

```
CREATE TABLE PARTLIST
(PART VARCHAR(8),
SUBPART VARCHAR(8),
QUANTITY INTEGER);
```

To give query results for this example, assume that the PARTLIST table is populated with the following values:

PART	SUBPART	QUANTITY
00	01	5
00	05	3
01	02	2
01	03	3
01	04	4
01	06	3
02	05	7
02	06	6
03	07	6
04	08	10
04	09	11
05	10	10
05	11	10
06	12	10
06	13	10
07	14	8
07	12	8

#### Example 1: Single level explosion

The first example is called single level explosion. It answers the question, "What parts are needed to build the part identified by '01'?" The list will include the direct subparts, subparts of the subparts and so on. However, if a part is used multiple times, its subparts are only listed once.

```
WITH RPL (PART, SUBPART, QUANTITY) AS
(SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
 FROM PARTLIST ROOT
 WHERE ROOT.PART = '01'
 UNION ALL
 SELECT CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
 FROM RPL PARENT, PARTLIST CHILD
 WHERE PARENT.SUBPART = CHILD.PART
)
SELECT DISTINCT PART, SUBPART, QUANTITY
FROM RPL
ORDER BY PART, SUBPART, QUANTITY;
```

The 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 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.

## Select-statement

### 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 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.



## Select-statement

►► FOR READ  
FETCH ONLY ◀◀

The FOR READ ONLY clause indicates that the result table is read-only and therefore the cursor cannot be referred to in Positioned UPDATE and DELETE statements. FOR FETCH ONLY has the same meaning.

Some result tables are read-only by nature. (For example, a table based on a read-only view.) FOR READ ONLY can still be specified for such tables, but the specification has no effect.

For result tables in which updates and deletes are allowed, specifying FOR READ ONLY (or FOR FETCH ONLY) can possibly improve the performance of FETCH operations by allowing the database manager to do blocking. For example, in programs that contain dynamic SQL statements without the FOR READ ONLY or ORDER BY clause, the database manager might open cursors as if the FOR UPDATE clause were specified. It is recommended, therefore, that the FOR READ ONLY clause be used to improve performance, except in cases where queries will be used in positioned UPDATE or DELETE statements.

A read-only result table must not be referred to in a Positioned UPDATE or DELETE statement, whether it is read-only by nature or specified as FOR READ ONLY (FOR FETCH ONLY).

### optimize-for-clause

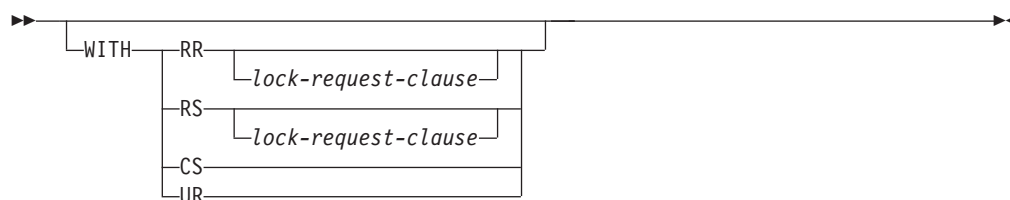
►► OPTIMIZE FOR *integer* ROWS  
ROW ◀◀

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 of *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 (that is, 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 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, the OPTIMIZE FOR clause will have no impact on the data buffers. The size of the communication buffer is defined by the `rqrioblk` or the `aslheapsz` configuration parameter.

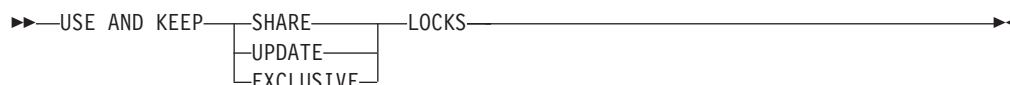


**isolation-clause**

The optional *isolation-clause* specifies the isolation level at which the statement is executed, and whether a specific type of lock is to be acquired.

- RR - Repeatable Read
- RS - Read Stability
- CS - Cursor Stability
- UR - Uncommitted Read

The default isolation level of the statement is the isolation level of the package in which the statement is bound. When a nickname is used in a *select-statement* to access data in DB2 family and Microsoft SQL Server data sources, the *isolation-clause* can be included in the statement to specify the statement isolation level. If the *isolation-clause* is included in statements that access other data sources, the specified isolation level is ignored. The current isolation level on the federated server is mapped to a corresponding isolation level at the data source on each connection to the data source. After a connection is made to a data source, the isolation level cannot be changed for the duration of the connection.

**lock-request-clause**

The optional *lock-request-clause* specifies the type of lock that the database manager is to acquire and hold:

**SHARE**

Concurrent processes can acquire SHARE or UPDATE locks on the data.

**UPDATE**

Concurrent processes can acquire SHARE locks on the data, but no concurrent process can acquire an UPDATE or EXCLUSIVE lock.

**EXCLUSIVE**

Concurrent processes cannot acquire a lock on the data.

The *lock-request-clause* applies to all base table and index scans required by the query, including those within subqueries, SQL functions and SQL methods. It has no affect on locks placed by procedures, external functions, or external methods. Any SQL function or SQL method invoked (directly or indirectly) by the statement must be created with INHERIT ISOLATION LEVEL WITH LOCK REQUEST (SQLSTATE 42601). The *lock-request-clause* cannot be used with a modifying query that might invoke triggers or that requires referential integrity checks (SQLSTATE 42601).

## Select-statement

### 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.

```
WITH
 DINFO (DEPTNO, AVGSALARY, EMPCOUNT) AS
 (SELECT OTHERS.WORKDEPT, AVG(OTHERS.SALARY), COUNT(*)
 FROM EMPLOYEE OTHERS
 GROUP BY OTHERS.WORKDEPT
),
 DINFOMAX AS
 (SELECT MAX(AVGSALARY) AS AVGMAX FROM DINFO)
SELECT THIS_EMP.EMPNO, THIS_EMP.SALARY,
 DINFO.AVGSALARY, DINFO.EMPCOUNT, DINFOMAX.AVGMAX
FROM EMPLOYEE THIS_EMP, DINFO, DINFOMAX
WHERE THIS_EMP.JOB = 'SALESREP'
AND THIS_EMP.WORKDEPT = DINFO.DEPTNO
```

*Example 7:* Given two tables, EMPLOYEE and PROJECT, replace employee SALLY with a new employee GEORGE, assign all projects lead by SALLY to GEORGE, and return the names of the updated projects.

```

WITH
 NEWEMP AS (SELECT EMPNO FROM NEW TABLE
 (INSERT INTO EMPLOYEE(EMPNO, FIRSTNME)
 VALUES(NEXT VALUE FOR EMPNO_SEQ, 'GEORGE'))),
 OLDEMP AS (SELECT EMPNO FROM EMPLOYEE WHERE FIRSTNME = 'SALLY'),
 UP PROJ AS (SELECT PROJNAME FROM NEW TABLE
 (UPDATE PROJECT
 SET RESPEMP = (SELECT EMPNO FROM NEWEMP)
 WHERE RESPEMP = (SELECT EMPNO FROM OLDEMP))),
 DELEMP AS (SELECT EMPNO FROM OLD TABLE
 (DELETE FROM EMPLOYEE
 WHERE EMPNO = (SELECT EMPNO FROM OLDEMP)))
SELECT PROJNAME FROM UP PROJ;

```

*Example 8:* Retrieve data from the DEPT table. That data will later be updated with a searched update, and should be locked when the query executes.

```

SELECT DEPTNO, DEPTNAME, MGRNO
FROM DEPT
WHERE ADMRDEPT = 'A00'
FOR READ ONLY WITH RS USE AND KEEP EXCLUSIVE LOCKS

```



---

## Appendix A. SQL and XML limits

The following tables describe certain SQL and XML limits. Adhering to the most restrictive case can help you to design application programs that are easily portable.

Table 42 lists limits in bytes. These limits are enforced after conversion from the application code page to the database code page when creating identifiers. The limits are also enforced after conversion from the database code page to the application code page when retrieving identifiers from the database. If, during either of these processes, the identifier length limit is exceeded, truncation occurs or an error is returned.

Character limits vary depending on the code page of the database and the code page of the application. For example, because the width of a UTF-8 character can range from 1 to 4 bytes, the character limit for an identifier in a Unicode table whose limit is 128 bytes will range from 32 to 128 characters, depending on which characters are used. If an attempt is made to create an identifier whose name is longer than the limit for this table after conversion to the database code page, an error is returned.

Applications that store identifier names must be able to handle the potentially increased size of identifiers after code page conversion has occurred. When identifiers are retrieved from the catalog, they are converted to the application code page. Conversion from the database code page to the application code page can result in an identifier becoming longer than the byte limit for the table. If a host variable declared by the application cannot store the entire identifier after code page conversion, it is truncated. If that is unacceptable, the host variable can be increased in size to be able to accept the entire identifier name.

The same rules apply to DB2 utilities retrieving data and converting it to a user-specified code page. If a DB2 utility, such as export, is retrieving the data and forcing conversion to a user-specified code page (using the export CODEPAGE modifier or the **DB2CODEPAGE** registry variable), and the identifier expands beyond the limit that is documented in this table because of code page conversion, an error might be returned or the identifier might be truncated.

*Table 42. Identifier Length Limits*

Description	Maximum in Bytes
Alias name	128
Attribute name	128
Audit policy name	128
Authorization name (can only be single-byte characters)	128
Buffer pool name	18
Column name <sup>2</sup>	128
Constraint name	128
Correlation name	128
Cursor name	128
Data partition name	128

## SQL and XML limits

Table 42. Identifier Length Limits (continued)

Description	Maximum in Bytes
Data source column name	255
Data source index name	128
Data source name	128
Data source table name ( <i>remote-table-name</i> )	128
Database partition group name	128
Database partition name	128
Event monitor name	128
External program name	128
Function mapping name	128
Group name	128
Host identifier <sup>1</sup>	255
Identifier for a data source user ( <i>remote-authorization-name</i> )	128
Identifier in an SQL procedure (condition name, for loop identifier, label, result set locator, statement name, variable name)	128
Index name	128
Index extension name	18
Index specification name	128
Label name	128
Namespace uniform resource identifier (URI)	1000
Nickname	128
Package name	128
Package version ID	64
Parameter name	128
Password to access a data source	32
Procedure name	128
Role name	128
Savepoint name	128
Schema name <sup>2</sup>	128
Security label component name	128
Security label name	128
Security policy name	128
Sequence name	128
Server (database alias) name	8
Specific name	128
SQL condition name	128
SQL variable name	128
Statement name	128
Table name	128
Table space name	18

Table 42. Identifier Length Limits (continued)

Description	Maximum in Bytes
Transform group name	18
Trigger name	128
Trusted context name	128
Type mapping name	18
User-defined function name	128
User-defined method name	128
User-defined type name <sup>2</sup>	128
View name	128
Wrapper name	128
XML element name, attribute name, or prefix name	1000
XML schema location uniform resource identifier (URI)	1000
<b>Note:</b>	
1. Individual host language compilers might have a more restrictive limit on variable names.	
2. The SQLDA structure is limited to storing 30-byte column names, 18-byte user-defined type names, and 8-byte schema names for user-defined types. Because the SQLDA is used in the DESCRIBE statement, embedded SQL applications that use the DESCRIBE statement to retrieve column or user-defined type name information must conform to these limits.	

Table 43. Numeric Limits

Description	Limit
Smallest SMALLINT value	-32 768
Largest SMALLINT value	+32 767
Smallest INTEGER value	-2 147 483 648
Largest INTEGER value	+2 147 483 647
Smallest BIGINT value	-9 223 372 036 854 775 808
Largest BIGINT value	+9 223 372 036 854 775 807
Largest decimal precision	31
Maximum exponent ( $E_{max}$ ) for REAL values	38
Smallest REAL value	-3.402E+38
Largest REAL value	+3.402E+38
Minimum exponent ( $E_{min}$ ) for REAL values	-37
Smallest positive REAL value	+1.175E-37
Largest negative REAL value	-1.175E-37
Maximum exponent ( $E_{max}$ ) for DOUBLE values	308
Smallest DOUBLE value	-1.79769E+308
Largest DOUBLE value	+1.79769E+308
Minimum exponent ( $E_{min}$ ) for DOUBLE values	-307





Table 44. String Limits (continued)

Description	Limit
Maximum length of CLOB (in bytes)	2 147 483 647
Maximum length of serialized XML (in bytes)	2 147 483 647
Maximum length of GRAPHIC (in double-byte characters)	127
Maximum length of VARGRAPHIC (in double-byte characters)	16 336
Maximum length of LONG VARGRAPHIC (in double-byte characters) <sup>1</sup>	16 350
Maximum length of DBCLOB (in double-byte characters)	1 073 741 823
Maximum length of BLOB (in bytes)	2 147 483 647
Maximum length of character constant	32 672
Maximum length of graphic constant	16 336
Maximum length of concatenated character string	2 147 483 647
Maximum length of concatenated graphic string	1 073 741 823
Maximum length of concatenated binary string	2 147 483 647
Maximum number of hexadecimal constant digits	32 672
Largest instance of a structured type column object at run time (in gigabytes)	1
Maximum size of a catalog comment (in bytes)	254
<b>Note:</b>	
1. The LONG VARCHAR and LONG VARGRAPHIC data types are deprecated, not recommended, and might be removed in a future release.	

Table 45. XML Limits

Description	Limit
Maximum depth of an XML document (in levels)	125
Maximum size of an XML schema document (in bytes)	31 457 280

Table 46. Datetime Limits

Description	Limit
Smallest DATE value	0001-01-01
Largest DATE value	9999-12-31
Smallest TIME value	00:00:00
Largest TIME value	24:00:00
Smallest TIMESTAMP value	0001-01-01-00.00.00.000000
Largest TIMESTAMP value	9999-12-31-24.00.00.000000

Table 47. Database Manager Limits

Description	Limit
<b>Applications</b>	
Maximum number of host variable declarations in a precompiled program <sup>3</sup>	storage
Maximum length of a host variable value (in bytes)	2 147 483 647

Table 47. Database Manager Limits (continued)

<b>Description</b>	<b>Limit</b>
Maximum number of declared cursors in a program	storage
Maximum number of rows changed in a unit of work	storage
Maximum number of cursors opened at one time	storage
Maximum number of connections per process within a DB2 client	512
Maximum number of simultaneously opened LOB locators in a transaction	4 194 304
Maximum size of an SQLDA (in bytes)	storage
Maximum number of prepared statements	storage
<b>Buffer Pools</b>	
Maximum NPAGES in a buffer pool for 32-bit releases	1 048 576
Maximum NPAGES in a buffer pool for 64-bit releases	2 147 483 647
Maximum total size of all buffer pool slots (4K)	2 147 483 646
<b>Concurrency</b>	
Maximum number of concurrent users of a server <sup>4</sup>	64 000
Maximum number of concurrent users per instance	64 000
Maximum number of concurrent applications per database	60 000
Maximum number of databases per instance concurrently in use	256
<b>Constraints</b>	
Maximum number of constraints on a table	storage
Maximum number of columns in a UNIQUE constraint (supported through a UNIQUE index)	64
Maximum combined length of columns in a UNIQUE constraint (supported through a UNIQUE index, in bytes) <sup>9</sup>	8192
Maximum number of referencing columns in a foreign key	64
Maximum combined length of referencing columns in a foreign key (in bytes) <sup>9</sup>	8192
Maximum length of a check constraint specification (in bytes)	65 535
<b>Databases</b>	
Maximum database partition number	999
<b>Indexes</b>	
Maximum number of indexes on a table	32 767 or storage
Maximum number of columns in an index key	64
Maximum length of an index key including all overhead <sup>7 9</sup>	<i>indexpagesize/4</i>
Maximum length of a variable index key part (in bytes) <sup>8</sup>	1022 or storage
Maximum size of an index per database partition in an SMS table space (in gigabytes) <sup>7</sup>	16 384
Maximum size of an index per database partition in a regular DMS table space (in gigabytes) <sup>7</sup>	512
Maximum size of an index per database partition in a large DMS table space (in gigabytes) <sup>7</sup>	16 384

Table 47. Database Manager Limits (continued)

Description	Limit
Maximum size of an index over XML data per database partition (in terabytes)	2
Maximum length of a variable index key part for an index over XML data (in bytes) <sup>7</sup>	<i>pagesize/4 - 207</i>
<b>Log records</b>	
Maximum Log Sequence Number	281 474 976 710 655
<b>Monitoring</b>	
Maximum number of simultaneously active event monitors	32
<b>Routines</b>	
Maximum number of parameters in a procedure	32 767
Maximum number of parameters in a user-defined function	90
Maximum number of nested levels for routines	64
Maximum number of schemas in the SQL path	64
Maximum length of the SQL path (in bytes)	2048
<b>Security</b>	
Maximum number of elements in a security label component of type set or tree	64
Maximum number of elements in a security label component of type array	65 535
Maximum number of security label components in a security policy	16
<b>SQL</b>	
Maximum total length of an SQL statement (in bytes)	2 097 152
Maximum number of tables referenced in an SQL statement or a view	storage
Maximum number of host variable references in an SQL statement	32 767
Maximum number of constants in a statement	storage
Maximum number of elements in a select list <sup>7</sup>	1012
Maximum number of predicates in a WHERE or HAVING clause	storage
Maximum number of columns in a GROUP BY clause <sup>7</sup>	1012
Maximum total length of columns in a GROUP BY clause (in bytes) <sup>7</sup>	32 677
Maximum number of columns in an ORDER BY clause <sup>7</sup>	1012
Maximum total length of columns in an ORDER BY clause (in bytes) <sup>7</sup>	32 677
Maximum level of subquery nesting	storage
Maximum number of subqueries in a single statement	storage
Maximum number of values in an insert operation <sup>7</sup>	1012
Maximum number of SET clauses in a single update operation <sup>7</sup>	1012
<b>Tables and Views</b>	

Table 47. Database Manager Limits (continued)

Description	Limit
Maximum number of columns in a table <sup>7</sup>	1012
Maximum number of columns in a view <sup>1</sup>	5000
Maximum number of columns in a data source table or view that is referenced by a nickname	5000
Maximum number of columns in a distribution key <sup>5</sup>	500
Maximum length of a row including all overhead <sup>2 7</sup>	32 677
Maximum number of rows in a non-partitioned table, per database partition	128 x 10 <sup>10</sup>
Maximum number of rows in a data partition, per database partition	128 x 10 <sup>10</sup>
Maximum size of a table per database partition in a regular table space (in gigabytes) <sup>3 7</sup>	512
Maximum size of a table per database partition in a large DMS table space (in gigabytes) <sup>7</sup>	16 384
Maximum number of data partitions for a single table	32 767
Maximum number of table partitioning columns	16
<b>Table Spaces</b>	
Maximum size of a LOB object (in terabytes)	4
Maximum size of a LF object (in terabytes)	2
Maximum number of table spaces in a database	32 768
Maximum number of tables in an SMS table space	65 534
Maximum size of a regular DMS table space (in gigabytes) <sup>3 7</sup>	512
Maximum size of a large DMS table space (in terabytes) <sup>3 7</sup>	16
Maximum size of a temporary DMS table space (in terabytes) <sup>37</sup>	16
Maximum number of table objects in a DMS table space <sup>6</sup>	51 000
Maximum number of storage paths in an automatic storage database	128
Maximum length of a storage path that is associated with an automatic storage database (in bytes)	175
<b>Triggers</b>	
Maximum run-time depth of cascading triggers	16
<b>User-defined Types</b>	
Maximum number of attributes in a structured type	4082

Table 47. Database Manager Limits (continued)

Description	Limit
<b>Note:</b>	
1. 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.	
2. 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.	
3. The numbers shown are architectural limits and approximations. The practical limits may be less.	
4. The actual value is controlled by the <b>max_connections</b> and <b>max_coordagents</b> database manager configuration parameters.	
5. This is an architectural limit. The limit on the most columns in an index key should be used as the practical limit.	
6. Table objects include data, indexes, LONG VARCHAR or 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.	
7. For page size-specific values, see Table 48.	
8. This is limited only by the longest index key, including all overhead (in bytes). As the number of index key parts increases, the maximum length of each key part decreases.	
9. The maximum can be less, depending on index options.	

Table 48. Database Manager Page Size-specific Limits

Description	4K page size limit	8K page size limit	16K page size limit	32K page size limit
Maximum number of columns in a table	500	1012	1012	1012
Maximum length of a row including all overhead	4005	8101	16 293	32 677
Maximum size of a table per database partition in a regular table space (in gigabytes)	64	128	256	512
Maximum size of a table per database partition in a large DMS table space (in gigabytes)	2048	4096	8192	16 384
Maximum length of an index key including all overhead (in bytes)	1024	2048	4096	8192
Maximum size of an index per database partition in an SMS table space (in gigabytes)	2048	4096	8192	16 384
Maximum size of an index per database partition in a regular DMS table space (in gigabytes)	64	128	256	512

## SQL and XML limits

Table 48. Database Manager Page Size-specific Limits (continued)

Description	4K page size limit	8K page size limit	16K page size limit	32K page size limit
Maximum size of an index per database partition in a large DMS table space (in gigabytes)	2048	4096	8192	16 384
Maximum size of an index over XML data per database partition (in terabytes)	2	2	2	2
Maximum size of a regular DMS table space (in gigabytes)	64	128	256	512
Maximum size of a large DMS table space (in gigabytes)	2048	4096	8192	16 384
Maximum size of a temporary DMS table space (in terabytes)	2	4	8	16
Maximum number of elements in a select list	500	1012	1012	1012
Maximum number of columns in a GROUP BY clause	500	1012	1012	1012
Maximum total length of columns in a GROUP BY clause (in bytes)	4005	8101	16 293	32 677
Maximum number of columns in an ORDER BY clause	500	1012	1012	1012
Maximum total length of columns in an ORDER BY clause (in bytes)	4005	8101	16 293	32 677
Maximum number of values in an insert operation	500	1012	1012	1012
Maximum number of SET clauses in a single update operation	500	1012	1012	1012

---

## Appendix B. SQLCA (SQL communications area)

An SQLCA is a collection of variables that is updated at the end of the execution of every SQL statement. A program that contains executable SQL statements and is precompiled with option LANGLEVEL SAA1 (the default) or MIA must provide exactly one SQLCA, though more than one SQLCA is possible by having one SQLCA per thread in a multi-threaded application.

When a program is precompiled with option LANGLEVEL SQL92E, an SQLCODE or SQLSTATE variable may be declared in the SQL declare section or an SQLCODE variable can be declared somewhere in the program.

An SQLCA should not be provided when using LANGLEVEL SQL92E. The SQL INCLUDE statement can be used to provide the declaration of the SQLCA in all languages but REXX. The SQLCA is automatically provided in REXX.

To display the SQLCA after each command executed through the command line processor, issue the command `db2 -a`. The SQLCA is then provided as part of the output for subsequent commands. The SQLCA is also dumped in the `db2diag.log` file.

### SQLCA field descriptions

*Table 49. Fields of the SQLCA.* The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement.

Name	Data Type	Field Values
sqlcaid	CHAR(8)	An "eye catcher" for storage dumps containing 'SQLCA'. The sixth byte is 'L' if line number information is returned from parsing an SQL procedure body.
sqlcabc	INTEGER	Contains the length of the SQLCA, 136.
sqlcode	INTEGER	Contains the SQL return code.  <b>Code</b> <b>Means</b>  <b>0</b> Successful execution (although one or more SQLWARN indicators may be set).  <b>positive</b> Successful execution, but with a warning condition.  <b>negative</b> Error condition.
sqlerrml	SMALLINT	Length indicator for <i>sqlerrmc</i> , in the range 0 through 70. 0 means that the value of <i>sqlerrmc</i> is not relevant.

## SQLCA (SQL communications area)

Table 49. Fields of the SQLCA (continued). The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement.

Name	Data Type	Field Values
sqlerrmc	VARCHAR (70)	<p>Contains one or more tokens, separated by X'FF', which are substituted for variables in the descriptions of error conditions.</p> <p>This field is also used when a successful connection is completed.</p> <p>When a NOT ATOMIC compound SQL statement is issued, it may contain information on up to seven errors.</p> <p>The last token might be followed by X'FF'. The <i>sqlerrml</i> value will include any trailing X'FF'.</p>
sqlerrp	CHAR(8)	<p>Begins with a three-letter identifier indicating the product, followed by five characters indicating the version, release, and modification level of the product. For example, SQL09010 means DB2 V9.1 (version 9, release 1, modification level 0).</p> <p>If SQLCODE indicates an error condition, this field identifies the module that returned the error.</p> <p>This field is also used when a successful connection is completed.</p>
sqlerrd	ARRAY	<p>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.</p>
sqlerrd(1)	INTEGER	<p>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.</p> <p>On successful return from an SQL procedure, contains the return status value from the SQL procedure.</p>
sqlerrd(2)	INTEGER	<p>If connection is invoked and successful, contains the maximum expected difference in length of mixed character data (CHAR data types) when converted to the application code page from the database code page. A value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction. If the SQLCA results from a NOT ATOMIC compound SQL statement that encountered one or more errors, the value is set to the number of statements that failed.</p>



Table 49. Fields of the SQLCA (continued). The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement.

Name	Data Type	Field Values
sqlerrd(3)	INTEGER	<p>If PREPARE is invoked and successful, contains an estimate of the number of rows that will be returned. After INSERT, UPDATE, DELETE, or MERGE, contains the actual number of rows that qualified for the operation. If compound SQL is invoked, contains an accumulation of all sub-statement rows. If CONNECT is invoked, contains 1 if the database can be updated, or 2 if the database is read only.</p> <p>If the OPEN statement is invoked, and the cursor contains SQL data change statements, this field contains the sum of the number of rows that qualified for the embedded insert, update, delete, or merge operations.</p> <p>If CREATE PROCEDURE for an SQL procedure is invoked, and an error is encountered when parsing the SQL procedure body, contains the line number where the error was encountered. The sixth byte of sqlcaid must be 'L' for this to be a valid line number.</p>
sqlerrd(4)	INTEGER	<p>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.</p>
sqlerrd(5)	INTEGER	<p>Contains the total number of rows deleted, inserted, or updated as a result of both:</p> <ul style="list-style-type: none"> <li>• The enforcement of constraints after a successful delete operation</li> <li>• The processing of triggered SQL statements from activated triggers</li> </ul> <p>If compound SQL is invoked, contains an accumulation of the number of such rows for all sub-statements. In some cases, when an error is encountered, this field contains a negative value that is an internal error pointer. If CONNECT is invoked, contains an authentication type value of 0 for server authentication; 1 for client authentication; 2 for authentication using DB2 Connect; 4 for SERVER_ENCRYPT authentication; 5 for authentication using DB2 Connect with encryption; 7 for KERBEROS authentication; 9 for GSSPLUGIN authentication; 11 for DATA_ENCRYPT authentication; and 255 for unspecified authentication.</p>
sqlerrd(6)	INTEGER	<p>For a partitioned database, contains the partition number of the database partition that encountered the error or warning. If no errors or warnings were encountered, this field contains the partition number of the coordinator partition. The number in this field is the same as that specified for the database partition in the db2nodes.cfg file.</p>

## SQLCA (SQL communications area)

Table 49. Fields of the SQLCA (continued). The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement.

Name	Data Type	Field Values
sqlwarn	Array	A set of warning indicators, each containing a blank or W. If compound SQL is invoked, contains an accumulation of the warning indicators set for all sub-statements.
sqlwarn0	CHAR(1)	Blank if all other indicators are blank; contains 'W' if at least one other indicator is not blank.
sqlwarn1	CHAR(1)	Contains 'W' if the value of a string column was truncated when assigned to a host variable. Contains 'N' if the null terminator was truncated. Contains 'A' if the CONNECT or ATTACH is successful, and the authorization name for the connection is longer than 8 bytes. Contains 'P' if the PREPARE statement relative cost estimate stored in sqlerrd(4) exceeded the value that could be stored in an INTEGER or was less than 1, and either the CURRENT EXPLAIN MODE or the CURRENT EXPLAIN SNAPSHOT special register is set to a value other than NO.
sqlwarn2	CHAR(1)	Contains 'W' if null values were eliminated from the argument of a column function. <sup>a</sup>  If CONNECT is invoked and successful, contains 'D' if the database is in quiesce state, or 'I' if the instance is in quiesce state.
sqlwarn3	CHAR(1)	Contains 'W' if the number of columns is not equal to the number of host variables. Contains 'Z' if the number of result set locators specified on the ASSOCIATE LOCATORS statement is less than the number of result sets returned by a procedure.
sqlwarn4	CHAR(1)	Contains 'W' if a prepared UPDATE or DELETE statement does not include a WHERE clause.
sqlwarn5	CHAR(1)	Contains 'E' if an error was tolerated during SQL statement execution.
sqlwarn6	CHAR(1)	Contains 'W' if the result of a date calculation was adjusted to avoid an impossible date.
sqlwarn7	CHAR(1)	Reserved for future use.  If CONNECT is invoked and successful, contains 'E' if the <b>dyn_query_mgmt</b> database configuration parameter is enabled.
sqlwarn8	CHAR(1)	Contains 'W' if a character that could not be converted was replaced with a substitution character. Contains 'Y' if there was an unsuccessful attempt to establish a trusted connection.
sqlwarn9	CHAR(1)	Contains 'W' if arithmetic expressions with errors were ignored during 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.

<sup>a</sup> Some functions may not set SQLWARN2 to W, even though null values were eliminated, because the result was not dependent on the elimination of null values.

## Error reporting

The order of error reporting is 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.

In a partitioned database system, the exception to this rule occurs if a data manipulation operation is invoked against a table that is empty on one database partition, but has data on other database partitions. SQLCODE +100 is only returned to the application if agents from all database partitions return SQL0100W, either because the table is empty on all database partitions, or there are no more rows that satisfy the WHERE clause in an UPDATE statement.

## SQLCA usage in partitioned database systems

In partitioned database systems, one SQL statement may be executed by a number of agents on different database partitions, and each agent may return a different SQLCA for different errors or warnings. The coordinator agent also has its own SQLCA.

To provide a consistent view for applications, all SQLCA values are merged into one structure, and SQLCA fields indicate global counts, such that:

- For all errors and warnings, the *sqlwarn* field contains the warning flags received from all agents.
- Values in the *sqlerrd* fields indicating row counts are accumulations from all agents.

Note that SQLSTATE 09000 may not be returned every time an error occurs during the processing of a triggered SQL statement.



---

## Appendix C. SQLDA (SQL descriptor area)

An SQLDA is a collection of variables that is required for execution of the SQL DESCRIBE statement. The SQLDA variables are options that can be used by the PREPARE, OPEN, FETCH, and EXECUTE statements. An SQLDA communicates with dynamic SQL; it can be used in a DESCRIBE statement, modified with the addresses of host variables, and then reused in a FETCH or EXECUTE statement.

SQLDAs are supported for all languages, but predefined declarations are provided only for C, REXX, FORTRAN, and COBOL.

The meaning of the information in an SQLDA depends on its use. In PREPARE and DESCRIBE, an SQLDA provides information to an application program about a prepared statement. In OPEN, EXECUTE, and FETCH, an SQLDA describes host variables.

In DESCRIBE and PREPARE, if any one of the columns being described is either a LOB type (LOB locators and file reference variables do not require doubled SQLDAs), reference type, or a user-defined type, the number of SQLVAR entries for the entire SQLDA will be doubled. For example:

- When describing a table with 3 VARCHAR columns and 1 INTEGER column, there will be 4 SQLVAR entries
- When describing a table with 2 VARCHAR columns, 1 CLOB column, and 1 integer column, there will be 8 SQLVAR entries

In EXECUTE, FETCH, and OPEN, if any one of the variables being described is a LOB type (LOB locators and file reference variables do not require doubled SQLDAs) or a structured type, the number of SQLVAR entries for the entire SQLDA must be doubled. (Distinct types and reference types are not relevant in these cases, because the additional information in the double entries is not required by the database.)

### SQLDA field descriptions

An SQLDA consists of four variables followed by an arbitrary number of occurrences of a sequence of variables collectively named SQLVAR. In OPEN, FETCH, and EXECUTE, each occurrence of SQLVAR describes a host variable. In DESCRIBE and PREPARE, each occurrence of SQLVAR describes a column of a result table or a parameter marker. There are two types of SQLVAR entries:

- **Base SQLVARs:** These entries are always present. They contain the base information about the column, parameter marker, or host variable such as data type code, length attribute, column name, host variable address, and indicator variable address.
- **Secondary SQLVARs:** These entries are only present if the number of SQLVAR entries is doubled as per the rules outlined above. For user-defined types (distinct or structured), they contain the user-defined type name. For reference types, they contain the target type of the reference. For LOBs, they contain the length attribute of the host variable and a pointer to the buffer that contains the actual length. (The distinct type and LOB information does not overlap, so distinct types can be based on LOBs without forcing the number of SQLVAR entries on a DESCRIBE to be tripled.) If locators or file reference variables are used to represent LOBs, these entries are not necessary.

## SQLDA (SQL descriptor area)

In SQLDAs that contain both types of entries, the base SQLVARs are in a block before the block of secondary SQLVARs. In each, the number of entries is equal to the value in SQLD (even though many of the secondary SQLVAR entries may be unused).

The circumstances under which the SQLVAR entries are set by DESCRIBE is detailed in “Effect of DESCRIBE on the SQLDA” on page 522.

### Fields in the SQLDA header

Table 50. Fields in the SQLDA Header

C Name	SQL Data Type	Usage in DESCRIBE and PREPARE (set by the database manager except for SQLN)	Usage in FETCH, OPEN, and EXECUTE (set by the application 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 522 for details on when SQLDOUBLED is set.	The seventh byte of this field is used when the number of SQLVARs is doubled. It is named SQLDOUBLED. If any of the host variables being described is a structured type, BLOB, CLOB, or DBCLOB, the seventh byte must be set to the character '2'; otherwise it can be set to any character but the use of a blank is recommended.
sqldabc	INTEGER	For 32 bit, the length of the SQLDA, equal to SQLN*44+16. For 64 bit, the length of the SQLDA, equal to SQLN*56+16	For 32 bit, the length of the SQLDA, >= to SQLN*44+16. For 64 bit, the length of the SQLDA, >= to SQLN*56+16.
sqln	SMALLINT	Unchanged by the database manager. Must be set to a value greater than or equal to zero before the DESCRIBE statement is executed. Indicates the total number of occurrences of SQLVAR.	Total number of occurrences of SQLVAR provided in the SQLDA. SQLN must be set to a value greater than or equal to zero.
sqld	SMALLINT	Set by the database manager to the number of columns in the result table or to the number of parameter markers.	The number of host variables described by occurrences of SQLVAR.

## Fields in an occurrence of a base SQLVAR

Table 51. Fields in a Base SQLVAR

Name	Data Type	Usage in DESCRIBE and PREPARE	Usage in FETCH, OPEN, and EXECUTE
sqltype	SMALLINT	<p>Indicates the data type of the column or parameter marker, and whether it can contain nulls. (Parameter markers are always considered nullable.) Table 53 on page 523 lists the allowable values and their meanings.</p> <p>Note that for a distinct or reference type, the data type of the base type is placed into this field. For a structured type, the data type of the result of the FROM SQL transform function of the transform group (based on the CURRENT DEFAULT TRANSFORM GROUP special register) for the type is placed into this field. There is no indication in the base SQLVAR that it is part of the description of a user-defined type or reference type.</p>	<p>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.</p>
sqllen	SMALLINT	<p>The length attribute of the column or parameter marker. For datetime columns and parameter markers, the length of the string representation of the values. See Table 53 on page 523.</p> <p>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).</p>	<p>The length attribute of the host variable. See Table 53 on page 523.</p> <p>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.</p>
sqldata	pointer	<p>For string SQLVARs, sqldata contains the code page. For character-string SQLVARs where the column is defined with the FOR BIT DATA attribute, sqldata contains 0. For other character-string SQLVARs, sqldata contains either the SBCS code page for SBCS data, or the SBCS code page associated with the composite MBCS code page for MBCS data. For Japanese EUC, Traditional Chinese EUC, and Unicode UTF-8 character-string SQLVARs, sqldata contains 954, 964, and 1208 respectively.</p> <p>For all other column types, sqldata is undefined.</p>	<p>Contains the address of the host variable (where the fetched data will be stored).</p>
sqlind	pointer	<p>For character-string SQLVARs, sqlind contains 0, except for MBCS data, when sqlind contains the DBCS code page associated with the composite MBCS code page.</p> <p>For all other types, sqlind is undefined.</p>	<p>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.</p>

## SQLDA (SQL descriptor area)

Table 51. Fields in a Base SQLVAR (continued)

Name	Data Type	Usage in DESCRIBE and PREPARE	Usage in FETCH, OPEN, and EXECUTE
sqlname	VARCHAR (30)	<p>Contains the unqualified name of the column or parameter marker.</p> <p>For columns and parameter markers that have a system-generated name, the thirtieth byte is set to X'FF'. For column names specified by the AS clause, this byte is X'00'.</p>	<p>When connecting to a host database, sqlname can be set to indicate a FOR BIT DATA string as follows:</p> <ul style="list-style-type: none"> <li>• The sixth byte of the SQLDAID in the SQLDA header is set to '+'</li> <li>• The length of sqlname is 8</li> <li>• The first two bytes of sqlname are X'0000'</li> <li>• The third and fourth bytes of sqlname are X'0000'</li> <li>• The remaining four bytes of sqlname are reserved and should be set to X'00000000'</li> </ul> <p>When working with XML data, sqlname can be set to indicate an XML subtype as follows:</p> <ul style="list-style-type: none"> <li>• The length of sqlname is 8</li> <li>• The first two bytes of sqlname are X'0000'</li> <li>• The third and fourth bytes of sqlname are X'0000'</li> <li>• The fifth byte of sqlname is X'01'</li> <li>• The remaining three bytes of sqlname are reserved and should be set to X'000000'</li> </ul>

## Fields in an occurrence of a secondary SQLVAR

Table 52. Fields in a Secondary SQLVAR

Name	Data Type	Usage in DESCRIBE and PREPARE	Usage in FETCH, OPEN, and EXECUTE
len.sqllonglen	INTEGER	The length attribute of a BLOB, CLOB, or DBCLOB column or parameter marker.	The length attribute of a BLOB, CLOB, or DBCLOB host variable. The database manager ignores the SQLLEN field in the Base SQLVAR for the data types. The length attribute stores the number of bytes for a BLOB or CLOB, and the number of double-byte characters for a DBCLOB.
reserve2	CHAR(3) for 32 bit, and CHAR(11) for 64 bit.	Not used.	Not used.



Table 52. Fields in a Secondary SQLVAR (continued)

Name	Data Type	Usage in DESCRIBE and PREPARE	Usage in FETCH, OPEN, and EXECUTE
sqlflag4	CHAR(1)	The value is X'01' if the SQLVAR represents a reference type with a target type named in sqldatatype_name. The value is X'12' if the SQLVAR represents a structured type, with the user-defined type name in sqldatatype_name. Otherwise, the value is X'00'.	Set to X'01' if the SQLVAR represents a reference type with a target type named in sqldatatype_name. Set to X'12' if the SQLVAR represents a structured type, with the user-defined type name in sqldatatype_name. Otherwise, the value is X'00'.
sqldatalen	pointer	Not used.	Used for BLOB, CLOB, and DBCLOB host variables only.  If this field is NULL, then the actual length (in double-byte characters) should be stored in the 4 bytes immediately before the start of the data and SQLDATA should point to the first byte of the field length.  If this field is not NULL, it contains a pointer to a 4 byte long buffer that contains the actual length <i>in bytes</i> (even for DBCLOB) of the data in the buffer pointed to from the SQLDATA field in the matching base SQLVAR.  Note that, whether or not this field is used, the len.sqllonglen field must be set.
sqldatatype_name	VARCHAR(27)	For a user-defined type, the database manager sets this to the fully qualified user-defined type name. <sup>1</sup> For a reference type, the database manager sets this to the fully qualified type name of the target type of the reference.	For structured types, set to the fully qualified user-defined type name in the format indicated in the table note. <sup>1</sup>
reserved	CHAR(3)	Not used.	Not used.

<sup>1</sup> The first 8 bytes contain the schema name of the type (extended to the right with spaces, if necessary). Byte 9 contains a dot (.). Bytes 10 to 27 contain the low order portion of the type name, which is *not* extended to the right with spaces.

Note that, although the prime purpose of this field is for the name of user-defined types, the field is also set for IBM predefined data types. In this case, the schema name is SYSIBM, and the low order portion of the name is the name stored in the TYPENAME column of the DATATYPES catalog view. For example:

type name	length	sqldatatype_name
-----	-----	-----
A.B	10	A .B
INTEGER	16	SYSIBM .INTEGER
"Frank's".SMINT	13	Frank's .SMINT
MY."type "	15	MY .type

### Effect of DESCRIBE on the SQLDA

For a DESCRIBE OUTPUT or PREPARE OUTPUT INTO statement, the database manager always sets SQLD to the number of columns in the result set, or the number of output parameter markers. For a DESCRIBE INPUT or PREPARE INPUT INTO statement, the database manager always sets SQLD to the number of input parameter markers in the statement. Note that a parameter marker that corresponds to an INOUT parameter in a CALL statement is described in both the input and output descriptors.

The SQLVARs in the SQLDA are set in the following cases:

- $SQLN \geq SQLD$  and no entry is either a LOB, user-defined type or reference type  
The first SQLD SQLVAR entries are set and SQLDOUBLED is set to blank.
- $SQLN \geq 2*SQLD$  and at least one entry is a LOB, user-defined type or reference type  
Two times SQLD SQLVAR entries are set, and SQLDOUBLED is set to '2'.
- $SQLD \leq SQLN < 2*SQLD$  and at least one entry is a distinct type or reference type, but there are no LOB entries or structured type entries  
The first SQLD SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +237 (SQLSTATE 01594) is issued.

The SQLVARs in the SQLDA are NOT set (requiring allocation of additional space and another DESCRIBE) in the following cases:

- $SQLN < SQLD$  and no entry is either a LOB, user-defined type or reference type  
No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +236 (SQLSTATE 01005) is issued.  
Allocate SQLD SQLVARs for a successful DESCRIBE.
- $SQLN < SQLD$  and at least one entry is a distinct type or reference type, but there are no LOB entries or structured type entries  
No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +239 (SQLSTATE 01005) is issued.  
Allocate  $2*SQLD$  SQLVARs for a successful DESCRIBE including the names of the distinct types and target types of reference types.
- $SQLN < 2*SQLD$  and at least one entry is a LOB or a structured type  
No SQLVAR entries are set and SQLDOUBLED is set to blank. A warning SQLCODE +238 (SQLSTATE 01005) is issued (regardless of the setting of the SQLWARN bind option).  
Allocate  $2*SQLD$  SQLVARs for a successful DESCRIBE.

References in the above lists to LOB entries include distinct type entries whose source type is a LOB type.

The SQLWARN option of the BIND or PREP command is used to control whether the DESCRIBE (or PREPARE INTO) will return the warning SQLCODEs +236, +237, +239. It is recommended that your application code always consider that these SQLCODEs could be returned. The warning SQLCODE +238 is always returned when there are LOB or structured type entries in the select list and there are insufficient SQLVARs in the SQLDA. This is the only way the application can know that the number of SQLVARs must be doubled because of a LOB or structured type entry in the result set.

If a structured type entry is being described, but no FROM SQL transform is defined (either because no TRANSFORM GROUP was specified using the CURRENT DEFAULT TRANSFORM GROUP special register (SQLSTATE 42741) or because the name group does not have a FROM SQL transform function defined (SQLSTATE 42744)), the DESCRIBE will return an error. This error is the same error returned for a DESCRIBE of a table with a structured type entry.

If the database manager returns identifiers that are longer than those that can be stored in the SQLDA, the identifier is truncated and a warning is returned (SQLSTATE 01665); however, when the name of a structured type is truncated, an error is returned (SQLSTATE 42622). For details on identifier length limitations, see “SQL and XQuery limits” .

## SQLTYPE and SQLLEN

Table 53 shows the values that may appear in the SQLTYPE and SQLLEN fields of the SQLDA. In DESCRIBE and PREPARE INTO, an even value of SQLTYPE means that the column does not allow nulls, and an odd value means the column does allow nulls. In FETCH, OPEN, and EXECUTE, an even value of SQLTYPE means that no indicator variable is provided, and an odd value means that SQLIND contains the address of an indicator variable.

Table 53. SQLTYPE and SQLLEN values for DESCRIBE, FETCH, OPEN, and EXECUTE

SQLTYPE	For DESCRIBE and PREPARE INTO		For FETCH, OPEN, and EXECUTE	
	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
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
400/401	N/A	N/A	NULL-terminated graphic string	length attribute of the host variable
404/405	BLOB	0 *	BLOB	Not used. *
408/409	CLOB	0 *	CLOB	Not used. *
412/413	DBCLOB	0 *	DBCLOB	Not used. *
448/449	varying-length character string	length attribute of the column	varying-length character string	length attribute of the host variable
452/453	fixed-length character string	length attribute of the column	fixed-length character string	length attribute of the host variable
456/457	long varying-length character string	length attribute of the column	long varying-length character string	length attribute of the host variable
460/461	not applicable	not applicable	NULL-terminated character string	length attribute of the host variable
464/465	varying-length graphic string	length attribute of the column	varying-length graphic string	length attribute of the host variable
468/469	fixed-length graphic string	length attribute of the column	fixed-length graphic string	length attribute of the host variable

## SQLDA (SQL descriptor area)

Table 53. *SQLTYPE* and *SQLLEN* values for *DESCRIBE*, *FETCH*, *OPEN*, and *EXECUTE* (continued)

SQLTYPE	For DESCRIBE and PREPARE INTO		For FETCH, OPEN, and EXECUTE	
	Column Data Type	SQLLEN	Host Variable Data Type	SQLLEN
472/473	long varying-length graphic string	length attribute of the column	long graphic string	length attribute of the host variable
480/481	floating-point	8 for double precision, 4 for single precision	floating-point	8 for double precision, 4 for single precision
484/485	packed decimal	precision in byte 1; scale in byte 2	packed decimal	precision in byte 1; scale in byte 2
492/493	big integer	8	big integer	8
496/497	large integer	4	large integer	4
500/501	small integer	2	small integer	2
916/917	not applicable	not applicable	BLOB file reference variable	267
920/921	not applicable	not applicable	CLOB file reference variable	267
924/925	not applicable	not applicable	DBCLOB file reference variable.	267
960/961	not applicable	not applicable	BLOB locator	4
964/965	not applicable	not applicable	CLOB locator	4
968/969	not applicable	not applicable	DBCLOB locator	4
988/989	XML	0	not applicable; use an XML AS <string or binary LOB type> host variable instead	not used
996	decimal floating-point	8 for DECFLOAT(16), 16 for DECFLOAT(34)	decimal floating-point	8 for DECFLOAT(16), 16 for DECFLOAT(34)

### Note:

- The `len.sqlqlonglen` 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) 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.

Data Type	Compatible Data Type
BIGINT	DECIMAL(19, 0)
ROWID <sup>1</sup>	VARCHAR(40) FOR BIT DATA

<sup>1</sup> ROWID is supported by DB2 Universal Database for z/OS Version 8.

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 54. Values for Sign Indicator of a Packed Decimal Number

Sign	Representation		
	Binary	Decimal	Hexadecimal
Positive (+)	1100	12	C
Negative (-)	1101	13	D

In summary:

- To store any value, allocate  $p/2+1$  bytes, where  $p$  is precision.
- Assign the nybbles from left to right to represent the value. If a number has an even precision, a leading zero nybble is added. This assignment includes leading (insignificant) and trailing (significant) zero digits.
- The sign nybble will be the second nybble of the last byte.

For example:

Column	Value	Nybbles in Hexadecimal Grouped by Bytes
DEC(8,3)	6574.23	00 65 74 23 0C
DEC(6,2)	-334.02	00 33 40 2D
DEC(7,5)	5.2323	05 23 23 0C
DEC(5,2)	-23.5	02 35 0D

## SQLDA (SQL descriptor area)

### 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].sqlen))[0] = precision;
((char *)&(sqlda->sqlvar[i].sqlen))[1] = scale;
```

---

## Appendix D. System catalog views

The database manager creates and maintains two sets of system catalog views that are defined on top of the base system catalog tables.

- SYSCAT views are read-only catalog views that are found in the SYSCAT schema. SELECT privilege on these views is granted to PUBLIC by default.
- SYSSTAT views are updatable catalog views that are found in the SYSSTAT schema. The updatable views contain statistical information that is used by the optimizer. The values in some columns in these views can be changed to test performance. (Before changing any statistics, it is recommended that the RUNSTATS command be invoked so that all the statistics reflect the current state.)

Applications should be written to the SYSCAT and SYSSTAT views rather than the base catalog tables.

All the system catalog views are created at database creation time. The catalog views cannot be explicitly created or dropped. In a Unicode database, the catalog views are created with IDENTITY collation. In non-Unicode databases, the catalog views are created with the database collation. The views are updated during normal operation in response to SQL data definition statements, environment routines, and certain utilities. Data in the system catalog views is available through normal SQL query facilities. The system catalog views (with the exception of some updatable catalog views) cannot be modified using normal SQL data manipulation statements.

An object (table, column, function, or index) will appear in a user's updatable catalog view only if that user created the object, holds CONTROL privilege on the object, or holds explicit DBADM authority. This also applies to a user with SYSADM authority who has implicit DBADM authority. In order for a SYSADM user to view records in all SYSSTAT views for objects it does not own, the user must be explicitly granted CONTROL privilege on the object, or explicitly granted DBADM authority.

The order of columns in the views may change from release to release. To prevent this from affecting programming logic, specify the columns in a select list explicitly, and avoid using SELECT \*. Columns have consistent names based on the types of objects that they describe.

### **Described Object**

#### **Column Names**

**Table** TABSCHEMA, TABNAME

**Index** INDSHEMA, INDNAME

**View** VIEWSCHEMA, VIEWNAME

### **Constraint**

CONSTSCHEMA, CONSTNAME

### **Trigger**

TRIGSCHEMA, TRIGNAME

### **Package**

PKGSHEMA, PKGNAME

## System catalog views

<b>Type</b>	TYPESHEMA, TYPENAME, TYPEID
<b>Function</b>	ROUTINESHEMA, ROUTINENAME, ROUTINEID
<b>Method</b>	ROUTINESHEMA, ROUTINENAME, ROUTINEID
<b>Procedure</b>	ROUTINESHEMA, ROUTINENAME, ROUTINEID
<b>Column</b>	COLNAME
<b>Schema</b>	SCHEMANAME
<b>Table Space</b>	TBSPACE
<b>Database partition group</b>	NGNAME
<b>Buffer pool</b>	BPNAME
<b>Event Monitor</b>	EVMONNAME
<b>Creation Timestamp</b>	CREATE_TIME

---

## Road map to the catalog views

Table 55. Road map to the read-only catalog views

Description	Catalog View
attributes of structured data types	"SYSCAT.ATTRIBUTES" on page 532
audit policies	"SYSCAT.AUDITPOLICIES" on page 533
	"SYSCAT.AUDITUSE" on page 535
authorities on database	"SYSCAT.DBAUTH" on page 552
buffer pool configuration on database partition group	"SYSCAT.BUFFERPOOLS" on page 536
buffer pool size on database partition	"SYSCAT.BUFFERPOOLDBPARTITIONS" on page 535
cast functions	"SYSCAT.CASTFUNCTIONS" on page 536
check constraints	"SYSCAT.CHECKS" on page 537
column privileges	"SYSCAT.COLAUTH" on page 538
columns	"SYSCAT.COLUMNS" on page 543
columns referenced by check constraints	"SYSCAT.COLCHECKS" on page 539
columns used in dimensions	"SYSCAT.COLUSE" on page 547
columns used in keys	"SYSCAT.KEYCOLUSE" on page 573
constraint dependencies	"SYSCAT.CONSTDEP" on page 548
database partition group database partitions	"SYSCAT.DBPARTITIONGROUPDEF" on page 554
database partition group definitions	"SYSCAT.DBPARTITIONGROUPS" on page 554
data partitions	"SYSCAT.DATAPARTITIONEXPRESSION" on page 549
	"SYSCAT.DATAPARTITIONS" on page 549



Table 55. Road map to the read-only catalog views (continued)

Description	Catalog View
data types	"SYSCAT.DATATYPES" on page 550
detailed column group statistics	"SYSCAT.COLGROUPOCOLS" on page 540
	"SYSCAT.COLGROUPODIST" on page 540
	"SYSCAT.COLGROUPODISTCOUNTS" on page 541
	"SYSCAT.COLGROUPOPS" on page 541
detailed column options	"SYSCAT.COLOPTIONS" on page 542
detailed column statistics	"SYSCAT.COLDIST" on page 539
distribution maps	"SYSCAT.PARTITIONMAPS" on page 584
event monitor definitions	"SYSCAT.EVENTMONITORS" on page 555
events currently monitored	"SYSCAT.EVENTS" on page 556
	"SYSCAT.EVENTTABLES" on page 557
function dependencies <sup>1</sup>	"SYSCAT.ROUTINEDEP" on page 587
function mapping	"SYSCAT.FUNCMAPPINGS" on page 559
function mapping options	"SYSCAT.FUNCMAPOPTIONS" on page 559
function parameter mapping options	"SYSCAT.FUNCMAPPARMOPTIONS" on page 559
function parameters <sup>1</sup>	"SYSCAT.ROUTINEPARMS" on page 589
functions <sup>1</sup>	"SYSCAT.ROUTINES" on page 591
global variables	"SYSCAT.VARIABLEAUTH" on page 629
	"SYSCAT.VARIABLEDEP" on page 629
	"SYSCAT.VARIABLES" on page 630
hierarchies (types, tables, views)	"SYSCAT.HIERARCHIES" on page 560
	"SYSCAT.FULLHIERARCHIES" on page 558
identity columns	"SYSCAT.COLIDENTATTRIBUTES" on page 542
index columns	"SYSCAT.INDEXCOLUSE" on page 562
index dependencies	"SYSCAT.INDEXDEP" on page 563
index exploitation	"SYSCAT.INDEXEXPLOITRULES" on page 569
index extension dependencies	"SYSCAT.INDEXEXTENSIONDEP" on page 570
index extension parameters	"SYSCAT.INDEXEXTENSIONPARMS" on page 571
index extension search methods	"SYSCAT.INDEXEXTENSIONMETHODS" on page 571
index extensions	"SYSCAT.INDEXEXTENSIONS" on page 572
index options	"SYSCAT.INDEXOPTIONS" on page 572
index privileges	"SYSCAT.INDEXAUTH" on page 562
indexes	"SYSCAT.INDEXES" on page 564
method dependencies <sup>1</sup>	"SYSCAT.ROUTINEDEP" on page 587
method parameters <sup>1</sup>	"SYSCAT.ROUTINES" on page 591
methods <sup>1</sup>	"SYSCAT.ROUTINES" on page 591
nicknames	"SYSCAT.NICKNAMES" on page 574
object mapping	"SYSCAT.NAMEMAPPINGS" on page 574
package dependencies	"SYSCAT.PACKAGEDEP" on page 578
package privileges	"SYSCAT.PACKAGEAUTH" on page 577

## Road map to the catalog views

Table 55. Road map to the read-only catalog views (continued)

Description	Catalog View
packages	"SYSCAT.PACKAGES" on page 579
partitioned tables	"SYSCAT.TABDETACHEDDEP" on page 613
pass-through privileges	"SYSCAT.PASSTHROUGH" on page 584
predicate specifications	"SYSCAT.PREDICATESPECS" on page 584
procedure options	"SYSCAT.ROUTINEOPTIONS" on page 588
procedure parameter options	"SYSCAT.ROUTINEPARMOPTIONS" on page 589
procedure parameters <sup>1</sup>	"SYSCAT.ROUTINEPARMS" on page 589
procedures <sup>1</sup>	"SYSCAT.ROUTINES" on page 591
protected tables	"SYSCAT.SECURITYLABELACCESS" on page 601
	"SYSCAT.SECURITYLABELCOMPONENTELEMENTS" on page 601
	"SYSCAT.SECURITYLABELCOMPONENTS" on page 602
	"SYSCAT.SECURITYLABELS" on page 602
	"SYSCAT.SECURITYPOLICIES" on page 602
	"SYSCAT.SECURITYPOLICYCOMPONENTRULES" on page 603
	"SYSCAT.SECURITYPOLICYEXEMPTIONS" on page 604
	"SYSCAT.SURROGATEAUTHIDS" on page 609
provides DB2 for z/OS compatibility	"SYSIBM.SYSDUMMY1" on page 642
referential constraints	"SYSCAT.REFERENCES" on page 585
remote table options	"SYSCAT.TABOPTIONS" on page 621
roles	"SYSCAT.ROLEAUTH" on page 586
	"SYSCAT.ROLES" on page 586
routine dependencies	"SYSCAT.ROUTINEDEP" on page 587
routine parameters <sup>1</sup>	"SYSCAT.ROUTINEPARMS" on page 589
routine privileges	"SYSCAT.ROUTINEAUTH" on page 586
routines <sup>1</sup>	"SYSCAT.ROUTINES" on page 591
	"SYSCAT.ROUTINESFEDERATED" on page 598
schema privileges	"SYSCAT.SCHEMAAUTH" on page 600
schemas	"SYSCAT.SCHEMATA" on page 600
sequence privileges	"SYSCAT.SEQUENCEAUTH" on page 605
sequences	"SYSCAT.SEQUENCES" on page 605
server options	"SYSCAT.SERVEROPTIONS" on page 607
server-specific user options	"SYSCAT.USEROPTIONS" on page 628
statements in packages	"SYSCAT.STATEMENTS" on page 608
stored procedures	"SYSCAT.ROUTINES" on page 591
system servers	"SYSCAT.SERVERS" on page 607
table constraints	"SYSCAT.TABCONST" on page 611
table dependencies	"SYSCAT.TABDEP" on page 612
table privileges	"SYSCAT.TABAUTH" on page 610

Table 55. Road map to the read-only catalog views (continued)

Description	Catalog View
table space use privileges	"SYSCAT.TBSPACEAUTH" on page 621
table spaces	"SYSCAT.TABLESPACES" on page 619
tables	"SYSCAT.TABLES" on page 614
transforms	"SYSCAT.TRANSFORMS" on page 623
trigger dependencies	"SYSCAT.TRIGDEP" on page 624
triggers	"SYSCAT.TRIGGERS" on page 624
trusted contexts	"SYSCAT.CONTEXTATTRIBUTES" on page 548 "SYSCAT.CONTEXTS" on page 548
type mapping	"SYSCAT.TYPEMAPPINGS" on page 626
user-defined functions	"SYSCAT.ROUTINES" on page 591
view dependencies	"SYSCAT.TABDEP" on page 612
views	"SYSCAT.TABLES" on page 614 "SYSCAT.VIEWS" on page 631
workload management	"SYSCAT.HISTOGRAMTEMPLATEBINS" on page 561 "SYSCAT.HISTOGRAMTEMPLATES" on page 561 "SYSCAT.HISTOGRAMTEMPLATEUSE" on page 561 "SYSCAT.SERVICECLASSES" on page 607 "SYSCAT.THRESHOLDS" on page 621 "SYSCAT.WORKACTIONS" on page 632 "SYSCAT.WORKACTIONSETS" on page 634 "SYSCAT.WORKCLASSES" on page 634 "SYSCAT.WORKCLASSETS" on page 635 "SYSCAT.WORKLOADAUTH" on page 636 "SYSCAT.WORKLOADCONNATTR" on page 636 "SYSCAT.WORKLOADS" on page 637
wrapper options	"SYSCAT.WRAPOPTIONS" on page 638
wrappers	"SYSCAT.WRAPPERS" on page 638
XML values index	"SYSCAT.INDEXXMLPATTERNS" on page 573
XSR objects	"SYSCAT.XDBMAPGRAPHS" on page 638 "SYSCAT.XDBMAPSHREDTREES" on page 638 "SYSCAT.XSROBJECTAUTH" on page 639 "SYSCAT.XSROBJECTCOMPONENTS" on page 639 "SYSCAT.XSROBJECTDEP" on page 640 "SYSCAT.XSROBJECTHIERARCHIES" on page 641 "SYSCAT.XSROBJECTS" on page 641

## Road map to the catalog views

Table 55. Road map to the read-only catalog views (continued)

Description	Catalog View
-------------	--------------

<sup>1</sup> The following catalog views for functions, methods, and procedures defined in DB2 Version 7.1 and earlier are still available:

Functions: SYSCAT.FUNCTIONS, SYSCAT.FUNCDEP, SYSCAT.FUNCPARMS

Methods: SYSCAT.FUNCTIONS, SYSCAT.FUNCDEP, SYSCAT.FUNCPARMS

Procedures: SYSCAT.PROCEDURES, SYSCAT.PROCPARMS

However, these views have not been updated since DB2 Version 7.1. Use the SYSCAT.ROUTINES, SYSCAT.ROUTINEDEP, or SYSCAT.ROUTINEPARMS catalog view instead.

Table 56. Road map to the updatable catalog views

Description	Catalog View
-------------	--------------

columns	"SYSSTAT.COLUMNS" on page 644
detailed column group statistics	"SYSSTAT.COLGROUPDIST" on page 643 "SYSSTAT.COLGROUPDISTCOUNTS" on page 643 "SYSSTAT.COLGROUPS" on page 644
detailed column statistics	"SYSSTAT.COLDIST" on page 642
indexes	"SYSSTAT.INDEXES" on page 645
routines <sup>1</sup>	"SYSSTAT.ROUTINES" on page 648
tables	"SYSSTAT.TABLES" on page 649

<sup>1</sup> The SYSSTAT.FUNCTIONS catalog view still exists for updating the statistics for functions and methods. This view, however, does not reflect any changes since DB2 Version 7.1.

## SYSCAT.ATTRIBUTES

Each row represents an attribute that is defined for a user-defined structured data type. Includes inherited attributes of subtypes.

Table 57. SYSCAT.ATTRIBUTES Catalog View

Column Name	Data Type	Nullable	Description
TYPESHEMA	VARCHAR (128)		Schema name of the structured data type that includes the attribute.
TYPENAME	VARCHAR (128)		Unqualified name of the structured data type that includes the attribute.
ATTR_NAME	VARCHAR (128)		Attribute name.
ATTR_TYPESHEMA	VARCHAR (128)		Schema name of the data type of an attribute.
ATTR_TYPENAME	VARCHAR (128)		Unqualified name of the data type of an attribute.
TARGET_TYPESHEMA	VARCHAR (128)	Y	Schema name of the target row type. Applies to reference types only; null value otherwise.
TARGET_TYPENAME	VARCHAR (128)	Y	Unqualified name of the target row type. Applies to reference types only; null value otherwise.

Table 57. SYSCAT.ATTRIBUTES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SOURCE_TYPESHEMA	VARCHAR (128)		For inherited attributes, the schema name of the data type with which the attribute was first defined. For non-inherited attributes, this column is the same as TYPESHEMA.
SOURCE_TYPENAME	VARCHAR (128)		For inherited attributes, the unqualified name of the data type with which the attribute was first defined. For non-inherited attributes, this column is the same as TYPENAME.
ORDINAL	SMALLINT		Position of the attribute in the definition of the structured data type, starting with 0.
LENGTH	INTEGER		For string types, contains the maximum length. For decimal type, contains the precision (number of digits); 0 otherwise.
SCALE	SMALLINT		For the decimal type, contains the scale (number of digits to the right of the decimal point); 0 otherwise.
CODEPAGE	SMALLINT		For string types, denotes the code page; 0 indicates FOR BIT DATA; 0 for non-string types.
COLLATIONSHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the attribute; null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the attribute; null value otherwise.
LOGGED	CHAR (1)		Applies to LOB types only; blank otherwise. <ul style="list-style-type: none"> <li>• N = Changes are not logged</li> <li>• Y = Changes are logged</li> </ul>
COMPACT	CHAR (1)		Applies to LOB types only; blank otherwise. <ul style="list-style-type: none"> <li>• N = Stored in non-compact format</li> <li>• Y = Stored in compact format</li> </ul>
DL_FEATURES	CHAR(10)		This column is no longer used and will be removed in a future release.
JAVA_FIELDNAME	VARCHAR (256)	Y	Reserved for future use.

## SYSCAT.AUDITPOLICIES

Each row represents an audit policy.

Table 58. SYSCAT.AUDITPOLICIES Catalog View

Column Name	Data Type	Nullable	Description
AUDITPOLICYNAME	VARCHAR (128)		Name of the audit policy.
AUDITPOLICYID	INTEGER		Identifier for the audit policy.
CREATE_TIME	TIMESTAMP		Time at which the audit policy was created.
ALTER_TIME	TIMESTAMP		Time at which the audit policy was last altered.

## SYSCAT.AUDITPOLICIES

Table 58. SYSCAT.AUDITPOLICIES Catalog View (continued)

Column Name	Data Type	Nullable	Description
AUDITSTATUS	CHAR (1)		Status for the AUDIT category. <ul style="list-style-type: none"><li>• B = Both</li><li>• F = Failure</li><li>• N = None</li><li>• S = Success</li></ul>
CONTEXTSTATUS	CHAR (1)		Status for the CONTEXT category. <ul style="list-style-type: none"><li>• B = Both</li><li>• F = Failure</li><li>• N = None</li><li>• S = Success</li></ul>
VALIDATESTATUS	CHAR (1)		Status for the VALIDATE category. <ul style="list-style-type: none"><li>• B = Both</li><li>• F = Failure</li><li>• N = None</li><li>• S = Success</li></ul>
CHECKINGSTATUS	CHAR (1)		Status for the CHECKING category. <ul style="list-style-type: none"><li>• B = Both</li><li>• F = Failure</li><li>• N = None</li><li>• S = Success</li></ul>
SECMAINTSTATUS	CHAR (1)		Status for the SECMAINT category. <ul style="list-style-type: none"><li>• B = Both</li><li>• F = Failure</li><li>• N = None</li><li>• S = Success</li></ul>
OBJMAINTSTATUS	CHAR (1)		Status for the OBJMAINT category. <ul style="list-style-type: none"><li>• B = Both</li><li>• F = Failure</li><li>• N = None</li><li>• S = Success</li></ul>
SYSADMINSTATUS	CHAR (1)		Status for the SYSADMIN category. <ul style="list-style-type: none"><li>• B = Both</li><li>• F = Failure</li><li>• N = None</li><li>• S = Success</li></ul>
EXECUTESTATUS	CHAR (1)		Status for the EXECUTE category. <ul style="list-style-type: none"><li>• B = Both</li><li>• F = Failure</li><li>• N = None</li><li>• S = Success</li></ul>
EXECUTEWITHDATA	CHAR (1)		Host variables and parameter markers logged with EXECUTE category. <ul style="list-style-type: none"><li>• N = No</li><li>• Y = Yes</li></ul>

Table 58. SYSCAT.AUDITPOLICIES Catalog View (continued)

Column Name	Data Type	Nullable	Description
ERRORTYPE	CHAR (1)		The audit error type. <ul style="list-style-type: none"> <li>• A = Audit</li> <li>• N = Normal</li> </ul>
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSCAT.AUDITUSE

Each row represents an audit policy that is associated with a non-database object, such as USER, GROUP, or authority (SYSADM, SYSCTRL, SYSMAINT).

Table 59. SYSCAT.AUDITUSE Catalog View

Column Name	Data Type	Nullable	Description
AUDITPOLICYNAME	VARCHAR (128)		Name of the audit policy.
AUDITPOLICYID	INTEGER		Identifier for the audit policy.
OBJECTTYPE	CHAR(1)		The type of object with which this audit policy is associated. <ul style="list-style-type: none"> <li>• S = MQT</li> <li>• T = Table</li> <li>• g = Authority</li> <li>• i = Authorization ID</li> <li>• x = Trusted context</li> <li>• Blank = Database</li> </ul>
SUBOBJECTTYPE	CHAR(1)		If OBJECTTYPE is 'i', this is the type that the authorization ID represents. <ul style="list-style-type: none"> <li>• G = Group</li> <li>• R = Role</li> <li>• U = User</li> <li>• Blank = Not applicable</li> </ul>
OBJECTSCHEMA	VARCHAR (128)		Schema name of the object for which the audit policy is in use. OBJECTSCHEMA is null if OBJECTTYPE identifies an object to which a schema does not apply.
OBJECTNAME	VARCHAR (128)		Unqualified name of the object for which this audit policy is in use.

## SYSCAT.BUFFERPOOLDBPARTITIONS

Each row represents a combination of a buffer pool and a database partition, in which the size of the buffer pool on that partition is different from its default size for other partitions in the same database partition group (as represented in SYSCAT.BUFFERPOOLS).

Table 60. SYSCAT.BUFFERPOOLDBPARTITIONS Catalog View

Column Name	Data Type	Nullable	Description
BUFFERPOOLID	INTEGER		Internal buffer pool identifier.
DBPARTITIONNUM	SMALLINT		Database partition number.

## SYSCAT.BUFFERPOOLDBPARTITIONS

Table 60. SYSCAT.BUFFERPOOLDBPARTITIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
NPAGES	INTEGER		Number of pages in this buffer pool on this database partition.

## SYSCAT.BUFFERPOOLS

Each row represents the configuration of a buffer pool on one database partition group of a database, or on all database partitions of a database.

Table 61. SYSCAT.BUFFERPOOLS Catalog View

Column Name	Data Type	Nullable	Description
BPNAME	VARCHAR (128)		Name of the buffer pool.
BUFFERPOOLID	INTEGER		Identifier for the buffer pool.
DBPGNAME	VARCHAR (128)	Y	Name of the database partition group (null if the buffer pool exists on all database partitions in the database).
NPAGES	INTEGER		Default number of pages in this buffer pool on database partitions in this database partition group.
PAGESIZE	INTEGER		Page size for this buffer pool on database partitions in this database partition group.
ESTORE	CHAR (1)		Always 'N'. Extended storage no longer applies.
NUMBLOCKPAGES	INTEGER		Number of pages of the buffer pool that are to be in a block-based area. A <i>block-based</i> area of the buffer pool is only used by prefetchers doing a sequential prefetch.
BLOCKSIZE	INTEGER		Number of pages in a <i>block</i> .
NGNAME <sup>1</sup>	VARCHAR (128)	Y	Name of the database partition group (null if the buffer pool exists on all database partitions in the database).

**Note:**

1. The NGNAME column is included for backwards compatibility. See DBPGNAME.

## SYSCAT.CASTFUNCTIONS

Each row represents a cast function, not including built-in cast functions.

Table 62. SYSCAT.CASTFUNCTIONS Catalog View

Column Name	Data Type	Nullable	Description
FROM_TYPESHEMA	VARCHAR (128)		Schema name of the data type of the parameter.
FROM_TYPENAME	VARCHAR (128)		Name of the data type of the parameter.
TO_TYPESHEMA	VARCHAR (128)		Schema name of the data type of the result after casting.
TO_TYPENAME	VARCHAR (128)		Name of the data type of the result after casting.



Table 62. SYSCAT.CASTFUNCTIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
FUNCSHEMA	VARCHAR (128)		Schema name of the function.
FUNCNAME	VARCHAR (128)		Unqualified name of the function.
SPECIFICNAME	VARCHAR (128)		Name of the routine instance (might be system-generated).
ASSIGN_FUNCTION	CHAR (1)		<ul style="list-style-type: none"> <li>• N = Not an assignment function</li> <li>• Y = Implicit assignment function</li> </ul>

## SYSCAT.CHECKS

Each row represents a check constraint or a derived column in a materialized query table. For table hierarchies, each check constraint is recorded only at the level of the hierarchy where the constraint was created.

Table 63. SYSCAT.CHECKS Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR (128)		Name of the check constraint.
OWNER	VARCHAR (128)		Authorization ID under which the check constraint was created.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• S = The owner is the system</li> <li>• U = The owner is an individual user</li> </ul>
TABSHEMA	VARCHAR (128)		Schema name of the table to which this constraint applies.
TABNAME	VARCHAR (128)		Name of the table to which this constraint applies.
CREATE_TIME	TIMESTAMP		Time at which the constraint was defined. Used in resolving functions that are part of this constraint. Functions that were created after the constraint was defined are not chosen.
QUALIFIER	VARCHAR (128)		Value of the default schema at the time of object definition. Used to complete any unqualified references.
TYPE	CHAR (1)		Type of check constraint: <ul style="list-style-type: none"> <li>• C = Check constraint</li> <li>• F = Functional dependency</li> <li>• O = Constraint is an object property</li> <li>• S = System-generated check constraint for a GENERATED ALWAYS column</li> </ul>
FUNC_PATH	CLOB (2K)		SQL path in effect when the constraint was defined; used to resolve functions and types that are part of the constraint.
TEXT	CLOB (2M)		Text of the check condition or definition of the derived column. <sup>1</sup>
PERCENTVALID	SMALLINT		Number of rows for which the informational constraint is valid, expressed as a percentage of the total.

## SYSCAT.CHECKS

Table 63. SYSCAT.CHECKS Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLLATIONSHEMA	VARCHAR (128)		Schema name of the collation for the constraint.
COLLATIONNAME	VARCHAR (128)		Unqualified name of the collation for the constraint.
COLLATIONSHEMA_ORDERBY	VARCHAR (128)		Schema name of the collation for ORDER BY clauses in the constraint.
COLLATIONNAME_ORDERBY	VARCHAR (128)		Unqualified name of the collation for ORDER BY clauses in the constraint.
DEFINER <sup>2</sup>	VARCHAR (128)		Authorization ID under which the check constraint was created.

**Note:**

1. In the catalog view, the text of the check condition is always shown in the database code page and can contain substitution characters. The check constraint will always be applied in the code page of the target table, and will not contain any substitution characters when applied. (The check constraint will be applied based on the original text in the code page of the target table, which might not include the substitution characters.)
2. The DEFINER column is included for backwards compatibility. See OWNER.

## SYSCAT.COLAUTH

Each row represents a user or a group that has been granted one or more privileges on a column.

Table 64. SYSCAT.COLAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of a privilege.
GRANTORTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• S = Grantor is the system</li><li>• U = Grantor is an individual user</li></ul>
GRANTEE	VARCHAR (128)		Holder of a privilege.
GRANTEETYPE	CHAR (1)		<ul style="list-style-type: none"><li>• G = Grantee is a group</li><li>• R = Grantee is a role</li><li>• U = Grantee is an individual user</li></ul>
TABSCHEMA	VARCHAR (128)		Schema name of the table or view on which the privilege is held.
TABNAME	VARCHAR (128)		Unqualified name of the table or view on which the privilege is held.
COLNAME	VARCHAR (128)		Name of the column to which this privilege applies.
COLNO	SMALLINT		Column number of this column within the table (starting with 0).
PRIVTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• R = Reference privilege</li><li>• U = Update privilege</li></ul>
GRANTABLE	CHAR (1)		<ul style="list-style-type: none"><li>• G = Privilege is grantable</li><li>• N = Privilege is not grantable</li></ul>

**Note:**

1. Privileges can be granted by column, but can be revoked only on a table-wide basis.

---

## SYSCAT.COLCHECKS

Each row represents a column that is referenced by a check constraint or by the definition of a materialized query table. For table hierarchies, each check constraint is recorded only at the level of the hierarchy where the constraint was created.

Table 65. SYSCAT.COLCHECKS Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR (128)		Name of the check constraint.
TABSCHEMA	VARCHAR (128)		Schema name of the table containing the referenced column.
TABNAME	VARCHAR (128)		Unqualified name of the table containing the referenced column.
COLNAME	VARCHAR (128)		Name of the column.
USAGE	CHAR (1)		<ul style="list-style-type: none"> <li>• D = Column is the child in a functional dependency</li> <li>• P = Column is the parent in a functional dependency</li> <li>• R = Column is referenced in the check constraint</li> <li>• S = Column is a source in the system-generated column check constraint that supports a materialized query table</li> <li>• T = Column is a target in the system-generated column check constraint that supports a materialized query table</li> </ul>

---

## SYSCAT.COLDIST

Each row represents the  $n$ th most frequent value of some column, or the  $n$ th quantile (cumulative distribution) value of the column. Applies to columns of real tables only (not views). No statistics are recorded for inherited columns of typed tables.

Table 66. SYSCAT.COLDIST Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the table to which the statistics apply.
TABNAME	VARCHAR (128)		Unqualified name of the table to which the statistics apply.
COLNAME	VARCHAR (128)		Name of the column to which the statistics apply.
TYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• F = Frequency value</li> <li>• Q = Quantile value</li> </ul>
SEQNO	SMALLINT		If TYPE = 'F', $n$ in this column identifies the $n$ th most frequent value. If TYPE = 'Q', $n$ in this column identifies the $n$ th quantile value.

## SYSCAT.COLDIST

Table 66. SYSCAT.COLDIST Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLVALUE <sup>1</sup>	VARCHAR (254)	Y	Data value as a character literal or a null value.
VALCOUNT	BIGINT		If TYPE = 'F', VALCOUNT is the number of occurrences of COLVALUE in the column. If TYPE = 'Q', VALCOUNT is the number of rows whose value is less than or equal to COLVALUE.
DISTCOUNT <sup>2</sup>	BIGINT	Y	If TYPE = 'Q', this column records the number of distinct values that are less than or equal to COLVALUE (null if unavailable).

**Note:**

1. In the catalog view, the value of COLVALUE is always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column's table, and will therefore use actual column values when applied during query optimization.
2. DISTCOUNT is collected only for columns that are the first key column in an index.

## SYSCAT.COLGROUPCOLS

Each row represents a column that makes up a column group.

Table 67. SYSCAT.COLGROUPCOLS Catalog View

Column Name	Data Type	Nullable	Description
COLGROUPEID	INTEGER		Identifier for the column group.
COLNAME	VARCHAR (128)		Name of the column in the column group.
TABSHEMA	VARCHAR (128)		Schema name of the table for the column in the column group.
TABNAME	VARCHAR (128)		Unqualified name of the table for the column in the column group.
ORDINAL	SMALLINT		Ordinal number of the column in the column group.

## SYSCAT.COLGROUPDIST

Each row represents the value of the column in a column group that makes up the *n*th most frequent value of the column group or the *n*th quantile value of the column group.

Table 68. SYSCAT.COLGROUPDIST Catalog View

Column Name	Data Type	Nullable	Description
COLGROUPEID	INTEGER		Identifier for the column group.
TYPE	CHAR (1)		<ul style="list-style-type: none"><li>• F = Frequency value</li><li>• Q = Quantile value</li></ul>
ORDINAL	SMALLINT		Ordinal number of the column in the column group.
SEQNO	SMALLINT		If TYPE = 'F', <i>n</i> in this column identifies the <i>n</i> th most frequent value. If TYPE = 'Q', <i>n</i> in this column identifies the <i>n</i> th quantile value.

Table 68. SYSCAT.COLGROUPDIST Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLVALUE <sup>1</sup>	VARCHAR (254)		Data value as a character literal or a null value.

**Note:**

1. In the catalog view, the value of COLVALUE is always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column's table, and will therefore use actual column values when applied during query optimization.

## SYSCAT.COLGROUPDISTCOUNTS

Each row represents the distribution statistics that apply to the *n*th most frequent value of a column group or the *n*th quantile of a column group.

Table 69. SYSCAT.COLGROUPDISTCOUNTS Catalog View

Column Name	Data Type	Nullable	Description
COLGROUPID	INTEGER		Identifier for the column group.
TYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• F = Frequency value</li> <li>• Q = Quantile value</li> </ul>
SEQNO	SMALLINT		Sequence number <i>n</i> representing the <i>n</i> th TYPE value.
VALCOUNT	BIGINT		If TYPE = 'F', VALCOUNT is the number of occurrences of COLVALUE for the column group with this SEQNO. If TYPE = 'Q', VALCOUNT is the number of rows whose value is less than or equal to COLVALUE for the column group with this SEQNO.
DISTCOUNT	BIGINT		If TYPE = 'Q', this column records the number of distinct values that are less than or equal to COLVALUE for the column group with this SEQNO (null if unavailable).

## SYSCAT.COLGROUPS

Each row represents a column group and statistics that apply to the entire column group.

Table 70. SYSCAT.COLGROUPS Catalog View

Column Name	Data Type	Nullable	Description
COLGROUPSCHEMA	VARCHAR (128)		Schema name of the column group.
COLGROUPNAME	VARCHAR (128)		Unqualified name of the column group.
COLGROUPID	INTEGER		Identifier for the column group.
COLGROUPCARD	BIGINT		Cardinality of the column group.
NUMFREQ_VALUES	SMALLINT		Number of frequent values collected for the column group.
NUMQUANTILES	SMALLINT		Number of quantiles collected for the column group.

## SYSCAT.COLIDENTATTRIBUTES

---

## SYSCAT.COLIDENTATTRIBUTES

Each row represents an identity column that is defined for a table.

Table 71. SYSCAT.COLIDENTATTRIBUTES Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the table or view that contains the column.
TABNAME	VARCHAR (128)		Unqualified name of the table or view that contains the column.
COLNAME	VARCHAR (128)		Name of the column.
START	DECIMAL (31,0)		Start value of the sequence.
INCREMENT	DECIMAL (31,0)		Increment value.
MINVALUE	DECIMAL (31,0)		Minimum value of the sequence.
MAXVALUE	DECIMAL (31,0)		Maximum value of the sequence.
CYCLE	CHAR (1)		Indicates whether or not the sequence can continue to generate values after reaching its maximum or minimum value. <ul style="list-style-type: none"><li>• N = Sequence cannot cycle</li><li>• Y = Sequence can cycle</li></ul>
CACHE	INTEGER		Number of sequence values to pre-allocate in memory for faster access. 0 indicates that values of the sequence are not to be preallocated. In a partitioned database, this value applies to each database partition.
ORDER	CHAR (1)		Indicates whether or not the sequence numbers must be generated in order of request. <ul style="list-style-type: none"><li>• N = Sequence numbers are not required to be generated in order of request</li><li>• Y = Sequence numbers must be generated in order of request</li></ul>
NEXTCACHEFIRSTVALUE	DECIMAL (31,0)	Y	The first value available to be assigned in the next cache block. If no caching, the next value available to be assigned.
SEQID	INTEGER		Identifier for the sequence.

---

## SYSCAT.COLOPTIONS

Each row contains column specific option values.

Table 72. SYSCAT.COLOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of a nickname.
TABNAME	VARCHAR (128)		Nickname for the column for which options are set.
COLNAME	VARCHAR (128)		Local column name.
OPTION	VARCHAR (128)		Name of the column option.
SETTING	CLOB (32K)		Value.

---

**SYSCAT.COLUMNS**

Each row represents a column defined for a table, view, or nickname.

Table 73. SYSCAT.COLUMNS Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the table, view, or nickname that contains the column.
TABNAME	VARCHAR (128)		Unqualified name of the table, view, or nickname that contains the column.
COLNAME	VARCHAR (128)		Name of the column.
COLNO	SMALLINT		Number of this column in the table (starting with 0).
TYPESCHEMA	VARCHAR (128)		Schema name of the data type for the column.
TYPENAME	VARCHAR (128)		Unqualified name of the data type for the column.
LENGTH	INTEGER		Maximum length of the data; 0 for distinct types. The LENGTH column indicates precision for DECIMAL fields, and indicates the number of bytes of storage required for decimal floating-point columns; that is, 8 and 16 for DECFLOAT(16) and DECFLOAT(34), respectively.
SCALE	SMALLINT		Scale if the column type is DECIMAL; 0 otherwise.
DEFAULT <sup>1</sup>	VARCHAR (254)	Y	Default value for the column of a table expressed as a constant, special register, or cast-function appropriate for the data type of the column. Can also be the keyword NULL. Values might be converted from what was specified as a default value. For example, date and time constants are shown in ISO format, cast-function names are qualified with schema names, and identifiers are delimited. Null value if a DEFAULT clause was not specified or the column is a view column.
NULLS <sup>2</sup>	CHAR (1)		Nullability attribute for the column. <ul style="list-style-type: none"> <li>• N = Column is not nullable</li> <li>• Y = Column is nullable</li> </ul> <p>The value can be 'N' for a view column that is derived from an expression or function. Nevertheless, such a column allows null values when the statement using the view is processed with warnings for arithmetic errors.</p>
CODEPAGE	SMALLINT		Code page used for data in this column; 0 if the column is defined as FOR BIT DATA or is not a string type.

## SYSCAT.COLUMNS

Table 73. SYSCAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLLATIONSHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the column; null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the column; null value otherwise.
LOGGED	CHAR (1)		Applies only to columns whose type is LOB or distinct based on LOB; blank otherwise. <ul style="list-style-type: none"> <li>• N = Column is not logged</li> <li>• Y = Column is logged</li> </ul>
COMPACT	CHAR (1)		Applies only to columns whose type is LOB or distinct based on LOB; blank otherwise. <ul style="list-style-type: none"> <li>• N = Column is not compacted</li> <li>• Y = Column is compacted in storage</li> </ul>
COLCARD	BIGINT		Number of distinct values in the column; -1 if statistics are not collected; -2 for inherited columns and columns of hierarchy tables.
HIGH2KEY <sup>3</sup>	VARCHAR (254)	Y	Second-highest data value. Representation of numeric data changed to character literals. Empty if statistics are not collected. Empty for inherited columns and columns of hierarchy tables.
LOW2KEY <sup>3</sup>	VARCHAR (254)	Y	Second-lowest data value. Representation of numeric data changed to character literals. Empty if statistics are not collected. Empty for inherited columns and columns of hierarchy tables.
AVGCOLLEN	INTEGER		Average space (in bytes) required for the column; -1 if a long field or LOB, or statistics have not been collected; -2 for inherited columns and columns of hierarchy tables.
KEYSEQ	SMALLINT	Y	The column's numerical position within the table's primary key. Null for columns of subtables and hierarchy tables.
PARTKEYSEQ	SMALLINT	Y	The column's numerical position within the table's distribution key; 0 or the null value if the column is not in the distribution key. Null for columns of subtables and hierarchy tables.
NQUANTILES	SMALLINT		Number of quantile values recorded in SYSCAT.COLDIST for this column; -1 if statistics are not gathered; -2 for inherited columns and columns of hierarchy tables.
NMOSTFREQ	SMALLINT		Number of most-frequent values recorded in SYSCAT.COLDIST for this column; -1 if statistics are not gathered; -2 for inherited columns and columns of hierarchy tables.
NUMNULLS	BIGINT		Number of null values in the column; -1 if statistics are not collected.



Table 73. SYSCAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description
TARGET_TYPESHEMA	VARCHAR (128)	Y	Schema name of the target row type, if the type of this column is REFERENCE; null value otherwise.
TARGET_TYPENAME	VARCHAR (128)	Y	Unqualified name of the target row type, if the type of this column is REFERENCE; null value otherwise.
SCOPE_TABSCHEMA	VARCHAR (128)	Y	Schema name of the scope (target table), if the type of this column is REFERENCE; null value otherwise.
SCOPE_TABNAME	VARCHAR (128)	Y	Unqualified name of the scope (target table), if the type of this column is REFERENCE; null value otherwise.
SOURCE_TABSCHEMA	VARCHAR (128)	Y	For columns of typed tables or views, the schema name of the table or view in which the column was first introduced. For non-inherited columns, this is the same as TABSCHEMA. Null for columns of non-typed tables and views.
SOURCE_TABNAME	VARCHAR (128)	Y	For columns of typed tables or views, the unqualified name of the table or view in which the column was first introduced. For non-inherited columns, this is the same as TABNAME. Null for columns of non-typed tables and views.
DL_FEATURES	CHAR (10)	Y	This column is no longer used and will be removed in a future release.
SPECIAL_PROPS	CHAR (8)	Y	Applies to REFERENCE type columns only; blanks otherwise. Each byte position is defined as follows: <ul style="list-style-type: none"> <li>• 1 = Object identifier (OID) column ('Y' for yes; 'N' for no)</li> <li>• 2 = User-generated or system-generated ('U' for user; 'S' for system)</li> </ul> Bytes 3 through 8 are reserved for future use.
HIDDEN	CHAR (1)		Type of hidden column. <ul style="list-style-type: none"> <li>• I = Column is defined as IMPLICITLY HIDDEN</li> <li>• S = System-managed hidden column</li> <li>• Blank = Column is not hidden</li> </ul>
INLINE_LENGTH	INTEGER		Maximum size in bytes of the internal representation of an instance of an XML document or a structured type that can be stored in the base table; 0 when not applicable.
IDENTITY	CHAR (1)		<ul style="list-style-type: none"> <li>• N = Not an identity column</li> <li>• T = Row change timestamp column</li> <li>• Y = Identity column</li> </ul>

## SYSCAT.COLUMNS

Table 73. SYSCAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description
ROWCHANGETIMESTAMP	CHAR (1)		<ul style="list-style-type: none"><li>• N = Not a row change timestamp column</li><li>• Y = Row change timestamp column</li></ul>
GENERATED	CHAR (1)		Type of generated column. <ul style="list-style-type: none"><li>• A = Column value is always generated</li><li>• D = Column value is generated by default</li><li>• Blank = Column is not generated</li></ul>
TEXT	CLOB (2M)	Y	For columns defined as generated as expression, this field contains the text of the generated column expression, starting with the keyword AS.
COMPRESS	CHAR (1)		<ul style="list-style-type: none"><li>• O = Compress off</li><li>• S = Compress system default values</li></ul>
AVGDISTINCTPERPAGE	DOUBLE	Y	For future use.
PAGEVARIANCERATIO	DOUBLE	Y	For future use.
SUB_COUNT	SMALLINT		Average number of sub-elements in the column. Applicable to character string columns only.
SUB_DELIM_LENGTH	SMALLINT		Average length of the delimiters that separate each sub-element in the column. Applicable to character string columns only.
AVGCOLLENCHAR	INTEGER		Average number of characters (based on the collation in effect for the column) required for the column; -1 if a long field or LOB, or statistics have not been collected; -2 for inherited columns and columns of hierarchy tables.
IMPLICITVALUE <sup>4</sup>	VARCHAR (254)	Y	For a column that was added to a table after the table was created, stores the default value at the time the column was added. For a column that was defined when the table was created, stores the null value.
SECLABELNAME	VARCHAR(128)	Y	Name of the security label that is associated with the column if it is a protected column; null value otherwise.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

Table 73. SYSCAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description
<b>Note:</b>			
1.	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.		
2.	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.		
3.	In the catalog view, the values of HIGH2KEY and LOW2KEY are always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column's table, and will therefore use actual column values when applied during query optimization.		
4.	Attaching a data partition is allowed unless IMPLICITVALUE for a specific column is a non-null value for both the source column and the target column, and the values do not match. In this case, you must drop the source table and then recreate it. A column can have a non-null value in the IMPLICITVALUE field if one of the following conditions is met: <ul style="list-style-type: none"> <li>• The column is created as the result of an ALTER TABLE...ADD COLUMN statement</li> <li>• The IMPLICITVALUE field is propagated from a source table during attach</li> <li>• The IMPLICITVALUE field is inherited from a source table during detach</li> <li>• The IMPLICITVALUE field is set during migration from Version 8 to Version 9, where it is determined to be an added column, or might be an added column. If the database is not certain whether the column is added or not, it is treated as added. An added column is a column that was created as the result of an ALTER TABLE...ADD COLUMN statement.</li> </ul>		

To avoid these inconsistencies during non-migration scenarios, it is recommended that you always create the tables that you are going to attach with all the columns already defined. That is, never use the ALTER TABLE statement to add columns to a table before attaching it.

## SYSCAT.COLUSE

Each row represents a column that is referenced in the DIMENSIONS clause of a CREATE TABLE statement.

Table 74. SYSCAT.COLUSE Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the table containing the column.
TABNAME	VARCHAR (128)		Unqualified name of the table containing the column.
COLNAME	VARCHAR (128)		Name of the column.
DIMENSION	SMALLINT		Dimension number, based on the order of dimensions specified in the DIMENSIONS clause (initial position is 0). For a composite dimension, this value will be the same for each component of the dimension.
COLSEQ	SMALLINT		Numeric position of the column in the dimension to which it belongs (initial position is 0). The value is 0 for the single column in a noncomposite dimension.
TYPE	CHAR (1)		Type of dimension. <ul style="list-style-type: none"> <li>• C = Clustering or multidimensional clustering</li> <li>• P = Partitioning</li> </ul>

## SYSCAT.CONSTDEP

---

## SYSCAT.CONSTDEP

Each row represents a dependency of a constraint on some other object. The constraint depends on the object of type BTYPE of name BNAME, so a change to the object affects the constraint.

Table 75. SYSCAT.CONSTDEP Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR (128)		Unqualified name of the constraint.
TABSCHEMA	VARCHAR (128)		Schema name of the table to which the constraint applies.
TABNAME	VARCHAR (128)		Unqualified name of the table to which the constraint applies.
BTYPE	CHAR (1)		Type of object on which the constraint depends. Possible values are: <ul style="list-style-type: none"><li>• F = Routine instance</li><li>• I = Index</li><li>• R = Structured type</li></ul>
BSCHEMA	VARCHAR (128)		Schema name of the object on which the constraint depends.
BNAME	VARCHAR (128)		Unqualified name of the object on which the constraint depends.

---

## SYSCAT.CONTEXTATTRIBUTES

Each row represents a trusted context attribute.

Table 76. SYSCAT.CONTEXTATTRIBUTES Catalog View

Column Name	Data Type	Nullable	Description
CONTEXTNAME	VARCHAR (128)		Name of the trusted context.
ATTR_NAME	VARCHAR (128)		Name of the attribute. One of: <ul style="list-style-type: none"><li>• ADDRESS</li><li>• ENCRYPTION</li></ul>
ATTR_VALUE	VARCHAR (128)		Value of the attribute.
ATTR_OPTIONS	VARCHAR (128)	Y	If ATTR_NAME is 'ADDRESS', specifies the level of encryption required for this specific address. A null value indicates that the global ENCRYPTION attribute applies.

---

## SYSCAT.CONTEXTS

Each row represents a trusted context.

Table 77. SYSCAT.CONTEXTS Catalog View

Column Name	Data Type	Nullable	Description
CONTEXTNAME	VARCHAR (128)		Name of the trusted context.
CONTEXTID	INTEGER		Identifier for the trusted context.

Table 77. SYSCAT.CONTEXTS Catalog View (continued)

Column Name	Data Type	Nullable	Description
SYSTEMAUTHID	VARCHAR (128)		The system authorization ID associated with the trusted context.
DEFAULTCONTEXTROLE	VARCHAR (128)	Y	The default role for the context.
CREATE_TIME	TIMESTAMP		Time at which the trusted context was created.
ALTER_TIME	TIMESTAMP		Time at which the trusted context was last altered.
ENABLED	CHAR (1)		Trusted context state. <ul style="list-style-type: none"> <li>• N = Disabled</li> <li>• Y = Enabled</li> </ul>
AUDITPOLICYID	INTEGER	Y	Identifier for the audit policy.
AUDITPOLICYNAME	VARCHAR (128)	Y	Name of the audit policy.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSCAT.DATAPARTITIONEXPRESSION

Each row represents an expression for that part of the table partitioning key.

Table 78. SYSCAT.DATAPARTITIONEXPRESSION Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the partitioned table.
TABNAME	VARCHAR (128)		Unqualified name of the partitioned table.
DATAPARTITIONKEYSEQ	INTEGER		Expression key part sequence ID, starting from 1.
DATAPARTITIONEXPRESSION	CLOB (32K)		Expression for this entry in the sequence, in SQL syntax.
NULLSFIRST	CHAR (1)		<ul style="list-style-type: none"> <li>• N = Null values in this expression compare high</li> <li>• Y = Null values in this expression compare low</li> </ul>

## SYSCAT.DATAPARTITIONS

Each row represents a data partition.

Table 79. SYSCAT.DATAPARTITIONS Catalog View

Column Name	Data Type	Nullable	Description
DATAPARTITIONNAME	VARCHAR (128)		Name of the data partition.
TABSCHEMA	VARCHAR (128)		Schema name of the table to which this data partition belongs.
TABNAME	VARCHAR (128)		Unqualified name of the table to which this data partition belongs.
DATAPARTITIONID	INTEGER		Identifier for the data partition.

## SYSCAT.DATAPARTITIONS

Table 79. SYSCAT.DATAPARTITIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
TBSPACEID	INTEGER	Y	Identifier for the table space in which this data partition is stored. Null when STATUS is 'I'.
PARTITIONOBJECTID	INTEGER	Y	Identifier for the data partition within the table space.
LONG_TBSPACEID	INTEGER	Y	Identifier for the table space in which long data is stored. Null when STATUS is 'I'.
ACCESS_MODE	CHAR (1)		Access restriction state of the data partition. These states only apply to objects that are in set integrity pending state or to objects that were processed by a SET INTEGRITY statement. Possible values are: <ul style="list-style-type: none"><li>• D = No data movement</li><li>• F = Full access</li><li>• N = No access</li><li>• R = Read-only access</li></ul>
STATUS	VARCHAR (32)		<ul style="list-style-type: none"><li>• A = Data partition is newly attached</li><li>• D = Data partition is detached</li><li>• I = Detached data partition whose entry in the catalog is maintained only during asynchronous index cleanup; rows with a STATUS value of 'I' are removed when all index records referring to the detached partition have been deleted</li><li>• Empty string = Data partition is visible (normal status)</li></ul> Bytes 2 through 32 are reserved for future use.
SEQNO	INTEGER		Data partition sequence number (starting from 0).
LOWINCLUSIVE	CHAR (1)		<ul style="list-style-type: none"><li>• N = Low key value is not inclusive</li><li>• Y = Low key value is inclusive</li></ul>
LOWVALUE	VARCHAR (512)		Low key value (a string representation of an SQL value) for this data partition.
HIGHINCLUSIVE	CHAR (1)		<ul style="list-style-type: none"><li>• N = High key value is not inclusive</li><li>• Y = High key value is inclusive</li></ul>
HIGHVALUE	VARCHAR (512)		High key value (a string representation of an SQL value) for this data partition.

## SYSCAT.DATATYPES

Each row represents a built-in or user-defined data type.

Table 80. SYSCAT.DATATYPES Catalog View

Column Name	Data Type	Nullable	Description
TYPESHEMA	VARCHAR (128)		Schema name of the data type. The schema name for built-in types is 'SYSIBM'.

Table 80. SYSCAT.DATATYPES Catalog View (continued)

Column Name	Data Type	Nullable	Description
TYPENAME	VARCHAR (128)		Unqualified name of the data type.
OWNER	VARCHAR (128)		Authorization ID under which the type was created.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• S = The owner is the system</li> <li>• U = The owner is an individual user</li> </ul>
SOURCESHEMA	VARCHAR (128)	Y	Schema name of the source type for distinct types. For user-defined structured types, this is the type schema of the reference representation type. Null for other types.
SOURCENAME	VARCHAR (128)	Y	Unqualified name of the source type for distinct types. For user-defined structured types, this is the type name of the reference representation type. Null for other types.
METATYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• A = Array type</li> <li>• R = User-defined structured type</li> <li>• S = System predefined type</li> <li>• T = User-defined distinct type</li> </ul>
TYPEID	SMALLINT		Identifier for the data type.
SOURCETYPEID	SMALLINT	Y	Identifier for the source type (null for built-in types). For user-defined structured types, this is the identifier of the reference representation type.
LENGTH	INTEGER		Maximum length of the type. 0 for built-in parameterized types (for example, DECIMAL and VARCHAR ). For user-defined structured types, this is the length of the reference representation type.
SCALE	SMALLINT		Scale for distinct types or reference representation types based on the built-in DECIMAL type; 0 for all other types (including DECIMAL itself). For user-defined structured types, this indicates the length of the reference representation type.
CODEPAGE	SMALLINT		Database code page for string types, distinct types based on string types, or reference representation types; 0 otherwise.
COLLATIONSHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the data type; null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the data type; null value otherwise.
ARRAY_LENGTH	INTEGER	Y	Maximum cardinality of the array. Null if the type is not an array type.
CREATE_TIME	TIMESTAMP		Creation time of the data type.
ATTRCOUNT	SMALLINT		Number of attributes in the data type.
INSTANTIABLE	CHAR (1)		<ul style="list-style-type: none"> <li>• N = Type cannot be instantiated</li> <li>• Y = Type can be instantiated</li> </ul>

## SYSCAT.DATATYPES

Table 80. SYSCAT.DATATYPES Catalog View (continued)

Column Name	Data Type	Nullable	Description
WITH_FUNC_ACCESS	CHAR (1)		<ul style="list-style-type: none"><li>• N = Methods for this type cannot be invoked using function notation</li><li>• Y = All the methods for this type can be invoked using function notation</li></ul>
FINAL	CHAR (1)		<ul style="list-style-type: none"><li>• N = The user-defined type can have subtypes</li><li>• Y = The user-defined type cannot have subtypes</li></ul>
INLINE_LENGTH	INTEGER		Maximum length of a structured type that can be kept with a base table row; 0 otherwise.
NATURAL_INLINE_LENGTH	INTEGER	Y	System-generated natural inline length of a structured type instance. Null if this type is not a structured type.
JARSCHEMA	VARCHAR (128)	Y	Schema name of the JAR_ID that identifies the Jar file containing the Java class that implements the SQL type. Null if the EXTERNAL NAME clause is not specified.
JAR_ID	VARCHAR (128)	Y	Identifier for the Jar file that contains the Java class that implements the SQL type. Null if the EXTERNAL NAME clause is not specified.
CLASS	VARCHAR (384)	Y	Java class that implements the SQL type. Null if the EXTERNAL NAME clause is not specified.
SQLJ_REPRESENTATION	CHAR (1)	Y	SQLJ "representation_spec" of the Java class that implements the SQL type. Null if the EXTERNAL NAME ... LANGUAGE JAVA REPRESENTATION SPEC clause is not specified. <ul style="list-style-type: none"><li>• D = SQL data</li><li>• S = Serializable</li></ul>
ALTER_TIME	TIMESTAMP		Time at which the data type was last altered.
DEFINER <sup>1</sup>	VARCHAR (128)		Authorization ID under which the type was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

**Note:**

1. The DEFINER column is included for backwards compatibility. See OWNER.

## SYSCAT.DBAUTH

Each row represents a user or a group that has been granted one or more database-level authorities.

Table 81. SYSCAT.DBAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the authority.



Table 81. SYSCAT.DBAUTH Catalog View (continued)

Column Name	Data Type	Nullable	Description
GRANTORTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• S = Grantor is the system</li> <li>• U = Grantor is an individual user</li> </ul>
GRANTEE	VARCHAR (128)		Holder of the authority.
GRANTEETYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• G = Grantee is a group</li> <li>• R = Grantee is a role</li> <li>• U = Grantee is an individual user</li> </ul>
BINDADDAUTH	CHAR (1)		Authority to create packages. <ul style="list-style-type: none"> <li>• N = Not held</li> <li>• Y = Held</li> </ul>
CONNECTAUTH	CHAR (1)		Authority to connect to the database. <ul style="list-style-type: none"> <li>• N = Not held</li> <li>• Y = Held</li> </ul>
CREATETABAUTH	CHAR (1)		Authority to create tables. <ul style="list-style-type: none"> <li>• N = Not held</li> <li>• Y = Held</li> </ul>
DBADMAUTH	CHAR (1)		DBADM authority. <ul style="list-style-type: none"> <li>• N = Not held</li> <li>• Y = Held</li> </ul>
EXTERNALROUTINEAUTH	CHAR (1)		Authority to create external routines. <ul style="list-style-type: none"> <li>• N = Not held</li> <li>• Y = Held</li> </ul>
IMPLSCHEMAAUTH	CHAR (1)		Authority to implicitly create schemas by creating objects in non-existent schemas. <ul style="list-style-type: none"> <li>• N = Not held</li> <li>• Y = Held</li> </ul>
LOADAUTH	CHAR (1)		Authority to use the DB2 load utility. <ul style="list-style-type: none"> <li>• N = Not held</li> <li>• Y = Held</li> </ul>
NOFENCEAUTH	CHAR (1)		Authority to create non-fenced user-defined functions. <ul style="list-style-type: none"> <li>• N = Not held</li> <li>• Y = Held</li> </ul>
QUIESCECONNECTAUTH	CHAR (1)		Authority to access the database when it is quiesced. <ul style="list-style-type: none"> <li>• N = Not held</li> <li>• Y = Held</li> </ul>
LIBRARYADMAUTH	CHAR (1)		Reserved for future use.
SECURITYADMAUTH	CHAR (1)		Security Administrator authority. <ul style="list-style-type: none"> <li>• N = Not held</li> <li>• Y = Held</li> </ul>

## SYSCAT.DBPARTITIONGROUPDEF

---

## SYSCAT.DBPARTITIONGROUPDEF

Each row represents a database partition that is contained in a database partition group.

Table 82. SYSCAT.DBPARTITIONGROUPDEF Catalog View

Column Name	Data Type	Nullable	Description
DBPGNAME	VARCHAR (128)		Name of the database partition group that contains the database partition.
DBPARTITIONNUM	SMALLINT		Partition number of a database partition that is contained in the database partition group. A valid partition number is between 0 and 999, inclusive.
IN_USE	CHAR (1)		Status of the database partition. <ul style="list-style-type: none"><li>• A = The newly added database partition is not in the distribution map, but the containers for the table spaces in the database partition group have been created; the database partition is added to the distribution map when a redistribute database partition group operation has completed successfully</li><li>• D = The database partition will be dropped when a redistribute database partition group operation has completed successfully</li><li>• T = The newly added database partition is not in the distribution map, and it was added using the WITHOUT TABLESPACES clause; containers must be added to the table spaces in the database partition group</li><li>• Y = The database partition is in the distribution map</li></ul>

---

## SYSCAT.DBPARTITIONGROUPS

Each row represents a database partition group.

Table 83. SYSCAT.DBPARTITIONGROUPS Catalog View

Column Name	Data Type	Nullable	Description
DBPGNAME	VARCHAR (128)		Name of the database partition group.
OWNER	VARCHAR (128)		Authorization ID under which the database partition group was created.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• S = The owner is the system</li><li>• U = The owner is an individual user</li></ul>
PMAP_ID	SMALLINT		Identifier for the distribution map in the SYSCAT.PARTITIONMAPS catalog view.
REDISTRIBUTE_PMAP_ID	SMALLINT		Identifier for the distribution map currently being used for redistribution; -1 if redistribution is currently not in progress.

Table 83. SYSCAT.DBPARTITIONGROUPS Catalog View (continued)

Column Name	Data Type	Nullable	Description
CREATE_TIME	TIMESTAMP		Creation time of the database partition group.
DEFINER <sup>1</sup>	VARCHAR (128)		Authorization ID under which the database partition group was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

**Note:**

1. The DEFINER column is included for backwards compatibility. See OWNER.

## SYSCAT.EVENTMONITORS

Each row represents an event monitor.

Table 84. SYSCAT.EVENTMONITORS Catalog View

Column Name	Data Type	Nullable	Description
EVMONNAME	VARCHAR (128)		Name of the event monitor.
OWNER	VARCHAR (128)		Authorization ID under which the event monitor was created.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• S = The owner is the system</li> <li>• U = The owner is an individual user</li> </ul>
TARGET_TYPE	CHAR (1)		Type of target to which event data is written. <ul style="list-style-type: none"> <li>• F = File</li> <li>• P = Pipe</li> <li>• T = Table</li> </ul>
TARGET	VARCHAR (762)		Name of the target to which file or pipe event monitor data is written. For files, it can be either an absolute path name or a relative path name (relative to the database path for the database; this can be seen using the LIST ACTIVE DATABASES command). For pipes, it can be an absolute path name.
MAXFILES	INTEGER	Y	Maximum number of event files that this event monitor permits in an event path. Null if there is no maximum, or if TARGET_TYPE is not 'F' (file).
MAXFILESIZE	INTEGER	Y	Maximum size (in 4K pages) that each event file can attain before the event monitor creates a new file. Null if there is no maximum, or if TARGET_TYPE is not 'F' (file).
BUFFERSIZE	INTEGER	Y	Size of the buffer (in 4K pages) that is used by event monitors with file targets; null value otherwise.
IO_MODE	CHAR (1)	Y	Mode of file input/output (I/O). <ul style="list-style-type: none"> <li>• B = Blocked</li> <li>• N = Not blocked</li> <li>• Null = TARGET_TYPE is not 'F' (file) or 'T' (table)</li> </ul>

## SYSCAT.EVENTMONITORS

Table 84. SYSCAT.EVENTMONITORS Catalog View (continued)

Column Name	Data Type	Nullable	Description
WRITE_MODE	CHAR (1)	Y	Indicates how this event monitor handles existing event data when the monitor is activated. <ul style="list-style-type: none"><li>• A = Append</li><li>• R = Replace</li><li>• Null = TARGET_TYPE is not 'F' (file)</li></ul>
AUTOSTART	CHAR (1)		Indicates whether this event monitor is to be activated automatically when the database starts. <ul style="list-style-type: none"><li>• N = No</li><li>• Y = Yes</li></ul>
DBPARTITIONNUM	SMALLINT		Number of the database partition on which the event monitor runs and logs events.
MONSCOPE	CHAR (1)		Monitoring scope. <ul style="list-style-type: none"><li>• G = Global</li><li>• L = Local</li><li>• T = Each database partition on which the table space exists</li><li>• Blank = WRITE TO TABLE event monitor</li></ul>
EVMON_ACTIVATES	INTEGER		Number of times the event monitor has been activated.
NODENUM <sup>1</sup>	SMALLINT		Number of the database partition on which the event monitor runs and logs events.
DEFINER <sup>2</sup>	VARCHAR (128)		Authorization ID under which the event monitor was created.
REMARKS	VARCHAR (254)	Y	Reserved for future use.

**Note:**

1. The NODENUM column is included for backwards compatibility. See DBPARTITIONNUM.
2. The DEFINER column is included for backwards compatibility. See OWNER.

## SYSCAT.EVENTS

Each row represents an event that is being monitored. An event monitor, in general, monitors multiple events.

Table 85. SYSCAT.EVENTS Catalog View

Column Name	Data Type	Nullable	Description
EVMONNAME	VARCHAR (128)		Name of the event monitor that is monitoring this event.

Table 85. SYSCAT.EVENTS Catalog View (continued)

Column Name	Data Type	Nullable	Description
TYPE	VARCHAR (128)		Type of event being monitored. Possible values are: <ul style="list-style-type: none"> <li>• ACTIVITIES</li> <li>• CONNECTIONS</li> <li>• DATABASE</li> <li>• DEADLOCKS</li> <li>• DETAILDEADLOCKS</li> <li>• STATEMENTS</li> <li>• TABLES</li> <li>• TABLESPACES</li> <li>• THRESHOLD_VIOLATIONS</li> <li>• TRANSACTIONS</li> <li>• STATISTICS</li> </ul>
FILTER	CLOB (64K)	Y	Full text of the WHERE clause that applies to this event.

## SYSCAT.EVENTTABLES

Each row represents the target table of an event monitor that writes to SQL tables.

Table 86. SYSCAT.EVENTTABLES Catalog View

Column Name	Data Type	Nullable	Description
EVMONNAME	VARCHAR (128)		Name of the event monitor.

## SYSCAT.EVENTTABLES

Table 86. SYSCAT.EVENTTABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
LOGICAL_GROUP	VARCHAR (128)		Name of the logical data group. Possible values are: <ul style="list-style-type: none"><li>• ACTIVITYHISTORY</li><li>• BUFFERPOOL</li><li>• CONN</li><li>• CONNHEADER</li><li>• CONTROL</li><li>• DATAVAL</li><li>• DB</li><li>• DEADLOCK</li><li>• DLCONN</li><li>• DLLOCK</li><li>• SCSTATS</li><li>• STMT</li><li>• STMTHIST</li><li>• STMTVALS</li><li>• SUBSECTION</li><li>• TABLE</li><li>• TABLESPACE</li><li>• THRESHOLDVIOLATIONS</li><li>• WCSTATS</li><li>• WLSTATS</li><li>• XACT</li></ul>
TABSCHEMA	VARCHAR (128)		Schema name of the target table.
TABNAME	VARCHAR (128)		Unqualified name of the target table.
PCTDEACTIVATE	SMALLINT		A percent value that specifies how full a DMS table space must be before an event monitor automatically deactivates. Set to 100 for SMS table spaces.

## 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 87. SYSCAT.FULLHIERARCHIES Catalog View

Column Name	Data Type	Nullable	Description
METATYPE	CHAR (1)		Relationship type. <ul style="list-style-type: none"><li>• R = Between structured types</li><li>• U = Between typed tables</li><li>• W = Between typed views</li></ul>
SUB_SCHEMA	VARCHAR (128)		Schema name of the subtype, subtable, or subview.
SUB_NAME	VARCHAR (128)		Unqualified name of the subtype, subtable, or subview.

Table 87. SYSCAT.FULLHIERARCHIES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SUPER_SCHEMA	VARCHAR (128)	Y	Schema name of the supertype, supertable, or superview.
SUPER_NAME	VARCHAR (128)	Y	Unqualified name of the supertype, supertable, or superview.
ROOT_SCHEMA	VARCHAR (128)		Schema name of the table, view, or type that is at the root of the hierarchy.
ROOT_NAME	VARCHAR (128)		Unqualified name of the table, view, or type that is at the root of the hierarchy.

## SYSCAT.FUNCMAPOPTIONS

Each row represents a function mapping option value.

Table 88. SYSCAT.FUNCMAPOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
FUNCTION_MAPPING	VARCHAR (128)		Name of the function mapping.
OPTION	VARCHAR (128)		Name of the function mapping option.
SETTING	VARCHAR (2048)		Value of the function mapping option.

## SYSCAT.FUNCMAPPARMOPTIONS

Each row represents a function mapping parameter option value.

Table 89. SYSCAT.FUNCMAPPARMOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
FUNCTION_MAPPING	VARCHAR (128)		Name of the function mapping.
ORDINAL	SMALLINT		Position of the parameter.
LOCATION	CHAR (1)		Location of the parameter. <ul style="list-style-type: none"> <li>• L = Local parameter</li> <li>• R = Remote parameter</li> </ul>
OPTION	VARCHAR (128)		Name of the function mapping parameter option.
SETTING	VARCHAR (2048)		Value of the function mapping parameter option.

## SYSCAT.FUNCMAPPINGS

Each row represents a function mapping.

Table 90. SYSCAT.FUNCMAPPINGS Catalog View

Column Name	Data Type	Nullable	Description
FUNCTION_MAPPING	VARCHAR (128)		Name of the function mapping (might be system-generated).
FUNCSHEMA	VARCHAR (128)	Y	Schema name of the function. If null, the function is assumed to be a built-in function.

## SYSCAT.FUNCMAPPINGS

Table 90. SYSCAT.FUNCMAPPINGS Catalog View (continued)

Column Name	Data Type	Nullable	Description
FUNCNAME	VARCHAR (1024)	Y	Unqualified name of the user-defined or built-in function.
FUNCID	INTEGER	Y	Identifier for the function.
SPECIFICNAME	VARCHAR (128)	Y	Name of the routine instance (might be system-generated).
OWNER	VARCHAR (128)		Authorization ID under which the mapping was created. 'SYSIBM' indicates that this is a built-in function.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• S = The owner is the system</li><li>• U = The owner is an individual user</li></ul>
WRAPNAME	VARCHAR (128)	Y	Wrapper to which this mapping applies.
SERVERNAME	VARCHAR (128)	Y	Name of the data source.
SERVERTYPE	VARCHAR (30)	Y	Type of data source to which this mapping applies.
SERVERVERSION	VARCHAR (18)	Y	Version of the server type to which this mapping applies.
CREATE_TIME	TIMESTAMP		Time at which the mapping was created.
DEFINER <sup>1</sup>	VARCHAR (128)		Authorization ID under which the mapping was created. 'SYSIBM' indicates that this is a built-in function.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

**Note:**

1. The DEFINER column is included for backwards compatibility. See OWNER.

## SYSCAT.HIERARCHIES

Each row represents the relationship between a subtable and its immediate supertable, a subtype and its immediate supertype, or a subview and its immediate superview. Only immediate hierarchical relationships are included in this view.

Table 91. SYSCAT.HIERARCHIES Catalog View

Column Name	Data Type	Nullable	Description
METATYPE	CHAR (1)		Relationship type. <ul style="list-style-type: none"><li>• R = Between structured types</li><li>• U = Between typed tables</li><li>• W = Between typed views</li></ul>
SUB_SCHEMA	VARCHAR (128)		Schema name of the subtype, subtable, or subview.
SUB_NAME	VARCHAR (128)		Unqualified name of the subtype, subtable, or subview.
SUPER_SCHEMA	VARCHAR (128)		Schema name of the supertype, supertable, or superview.
SUPER_NAME	VARCHAR (128)		Unqualified name of the supertype, supertable, or superview.



Table 91. SYSCAT.HIERARCHIES Catalog View (continued)

Column Name	Data Type	Nullable	Description
ROOT_SCHEMA	VARCHAR (128)		Schema name of the table, view, or type that is at the root of the hierarchy.
ROOT_NAME	VARCHAR (128)		Unqualified name of the table, view, or type that is at the root of the hierarchy.

## SYSCAT.HISTOGRAMTEMPLATEBINS

Each row represents a histogram template bin.

Table 92. SYSCAT.HISTOGRAMTEMPLATEBINS Catalog View

Column Name	Data Type	Nullable	Description
TEMPLATENAME	VARCHAR (128)	Y	Name of the histogram template.
TEMPLATEID	INTEGER		Identifier for the histogram template.
BINID	INTEGER		Identifier for the histogram template bin.
BINUPPERVALUE	BIGINT		The upper value for a single bin in the histogram template.

## SYSCAT.HISTOGRAMTEMPLATES

Each row represents a histogram template.

Table 93. SYSCAT.HISTOGRAMTEMPLATES Catalog View

Column Name	Data Type	Nullable	Description
TEMPLATEID	INTEGER		Identifier for the histogram template.
TEMPLATENAME	VARCHAR (128)		Name of the histogram template.
CREATE_TIME	TIMESTAMP		Time at which the histogram template was created.
ALTER_TIME	TIMESTAMP		Time at which the histogram template was last altered.
NUMBINS	INTEGER		Number of bins in the histogram template, including the last bin that has an unbounded top value.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSCAT.HISTOGRAMTEMPLATEUSE

Each row represents a relationship between a workload management object that can use histogram templates and a histogram template.

Table 94. SYSCAT.HISTOGRAMTEMPLATEUSE Catalog View

Column Name	Data Type	Nullable	Description
TEMPLATENAME	VARCHAR (128)	Y	Name of the histogram template.
TEMPLATEID	INTEGER		Identifier for the histogram template.

## SYSCAT.HISTOGRAMTEMPLATEUSE

Table 94. SYSCAT.HISTOGRAMTEMPLATEUSE Catalog View (continued)

Column Name	Data Type	Nullable	Description
HISTOGRAMTYPE	CHAR (1)		The type of information collected by histograms based on this template. <ul style="list-style-type: none"><li>• C = Activity estimated cost histogram</li><li>• E = Activity execution time histogram</li><li>• I = Activity interarrival time histogram</li><li>• L = Activity life time histogram</li><li>• Q = Activity queue time histogram</li><li>• R = Request execution time histogram</li></ul>
OBJECTTYPE	CHAR (1)		The type of WLM object. <ul style="list-style-type: none"><li>• b = Service class</li><li>• k = Work action</li></ul>
OBJECTID	INTEGER		Identifier of the WLM object.
SERVICECLASSNAME	VARCHAR (128)	Y	Name of the service class.
PARENTSERVICECLASSNAME	VARCHAR (128)	Y	Name of the parent service class.
WORKACTIONNAME	VARCHAR (128)	Y	Name of the work action.
WORKACTIONSETNAME	VARCHAR (128)	Y	Name of the work action set.

## SYSCAT.INDEXAUTH

Each row represents a user or group that has been granted CONTROL privilege on an index.

Table 95. SYSCAT.INDEXAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• S = Grantor is the system</li><li>• U = Grantor is an individual user</li></ul>
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		<ul style="list-style-type: none"><li>• G = Grantee is a group</li><li>• R = Grantee is a role</li><li>• U = Grantee is an individual user</li></ul>
INDSCHEMA	VARCHAR (128)		Schema name of the index.
INDNAME	VARCHAR (128)		Unqualified name of the index.
CONTROLAUTH	CHAR (1)		CONTROL privilege. <ul style="list-style-type: none"><li>• N = Not held</li><li>• Y = Held</li></ul>

## SYSCAT.INDEXCOLUSE

Each row represents a column that participates in an index.

Table 96. SYSCAT.INDEXCOLUSE Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR (128)		Schema name of the index.
INDNAME	VARCHAR (128)		Unqualified name of the index.
COLNAME	VARCHAR (128)		Name of the column.
COLSEQ	SMALLINT		Numeric position of the column in the index (initial position is 1).
COLORDER	CHAR (1)		Order of the values in this index column. Possible values are: <ul style="list-style-type: none"> <li>• A = Ascending</li> <li>• D = Descending</li> <li>• I = INCLUDE column (ordering ignored)</li> </ul>
COLLATIONSCHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the column; null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the column; null value otherwise.

## SYSCAT.INDEXDEP

Each row represents a dependency of an index on some other object. The index depends on an object of type BTYPE and name BNAME, so a change to the object affects the index.

Table 97. SYSCAT.INDEXDEP Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR (128)		Schema name of the index.
INDNAME	VARCHAR (128)		Unqualified name of the index.

## SYSCAT.INDEXDEP

Table 97. SYSCAT.INDEXDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: <ul style="list-style-type: none"><li>• A = Alias</li><li>• B = Trigger</li><li>• F = Routine instance</li><li>• H = Hierachy table</li><li>• K = Package</li><li>• L = Detached table</li><li>• O = Privilege dependency on all subtables or subviews in a table or view hierarchy</li><li>• Q = Sequence</li><li>• R = Structured type</li><li>• S = Materialized query table</li><li>• T = Table (not typed)</li><li>• U = Typed table</li><li>• V = View (not typed)</li><li>• W = Typed view</li><li>• X = Index extension</li><li>• Z = XSR object</li><li>• v = Global variable</li></ul>
BSHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.
TABAUTH	SMALLINT	Y	If BTYPE = 'O', 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by the dependent index; null value otherwise.

## SYSCAT.INDEXES

Each row represents an index. Indexes on typed tables are represented by two rows: one for the "logical index" on the typed table, and one for the "H-index" on the hierarchy table.

Table 98. SYSCAT.INDEXES Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR (128)		Schema name of the index.
INDNAME	VARCHAR (128)		Unqualified name of the index.
OWNER	VARCHAR (128)		Authorization ID under which the index was created.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• S = The owner is the system</li><li>• U = The owner is an individual user</li></ul>
TABSHEMA	VARCHAR (128)		Schema name of the table or nickname on which the index is defined.

Table 98. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
TABNAME	VARCHAR (128)		Unqualified name of the table or nickname on which the index is defined.
COLNAMES	VARCHAR (640)		This column is no longer used and will be removed in the next release. Use SYSCAT.INDEXCOLUSE for this information.
UNIQUERULE	CHAR (1)		Unique rule. <ul style="list-style-type: none"> <li>• D = Permits duplicates</li> <li>• U = Unique</li> <li>• P = Implements primary key</li> </ul>
MADE_UNIQUE	CHAR (1)		<ul style="list-style-type: none"> <li>• N = Index remains as it was created</li> <li>• Y = This index was originally non-unique but was converted to a unique index to support a unique or primary key constraint. If the constraint is dropped, the index reverts to being non-unique.</li> </ul>
COLCOUNT	SMALLINT		Number of columns in the key, plus the number of include columns, if any.
UNIQUE_COLCOUNT	SMALLINT		Number of columns required for a unique key. It is always <= COLCOUNT, and < COLCOUNT only if there are include columns; -1 if the index has no unique key (that is, it permits duplicates).
INDEXTYPE <sup>5</sup>	CHAR (4)		Type of index. <ul style="list-style-type: none"> <li>• BLOK = Block index</li> <li>• CLUS = Clustering index (controls the physical placement of newly inserted rows)</li> <li>• DIM = Dimension block index</li> <li>• REG = Regular index</li> <li>• XPTH = XML path index</li> <li>• XRGN = XML region index</li> <li>• XVIL = Index over XML column (logical)</li> <li>• XVIP = Index over XML column (physical)</li> </ul>
ENTRYTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• H = This row represents an index on a hierarchy table</li> <li>• L = This row represents a logical index on a typed table</li> <li>• Blank = This row represents an index on an untyped table</li> </ul>
PCTFREE	SMALLINT		Percentage of each index page to be reserved during the initial building of the index. This space is available for data insertions after the index has been built.
IID	SMALLINT		Identifier for the index.
NLEAF	BIGINT		Number of leaf pages; -1 if statistics are not collected.

## SYSCAT.INDEXES

Table 98. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
NLEVELS	SMALLINT		Number of index levels; -1 if statistics are not collected.
FIRSTKEYCARD	BIGINT		Number of distinct first-key values; -1 if statistics are not collected.
FIRST2KEYCARD	BIGINT		Number of distinct keys using the first two columns of the index; -1 if statistics are not collected, or if not applicable.
FIRST3KEYCARD	BIGINT		Number of distinct keys using the first three columns of the index; -1 if statistics are not collected, or if not applicable.
FIRST4KEYCARD	BIGINT		Number of distinct keys using the first four columns of the index; -1 if statistics are not collected, or if not applicable.
FULLKEYCARD	BIGINT		Number of distinct full-key values; -1 if statistics are not collected.
CLUSTERRATIO <sup>3</sup>	SMALLINT		Degree of data clustering with the index; -1 if statistics are not collected or if detailed index statistics are collected (in which case, CLUSTERFACTOR will be used instead).
CLUSTERFACTOR <sup>3</sup>	DOUBLE		Finer measurement of the degree of clustering; -1 if statistics are not collected or if the index is defined on a nickname.
SEQUENTIAL_PAGES	BIGINT		Number of leaf pages located on disk in index key order with few or no large gaps between them; -1 if statistics are not collected.
DENSITY	INTEGER		Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percent (integer between 0 and 100); -1 if statistics are not collected.
USER_DEFINED	SMALLINT		1 if this index was defined by a user and has not been dropped; 0 otherwise.
SYSTEM_REQUIRED	SMALLINT		<ul style="list-style-type: none"> <li>• 1 if one or the other of the following conditions is met: <ul style="list-style-type: none"> <li>– This index is required for a primary or unique key constraint, or this index is a dimension block index or composite block index for a multidimensional clustering (MDC) table.</li> <li>– This is the index on the object identifier (OID) column of a typed table.</li> </ul> </li> <li>• 2 if both of the following conditions are met: <ul style="list-style-type: none"> <li>– This index is required for a primary or unique key constraint, or this index is a dimension block index or composite block index for an MDC table.</li> <li>– This is the index on the OID column of a typed table.</li> </ul> </li> <li>• 0 otherwise.</li> </ul>

Table 98. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
CREATE_TIME	TIMESTAMP		Time when the index was created.
STATS_TIME	TIMESTAMP	Y	Last time that any change was made to the recorded statistics for this index. Null if no statistics are available.
PAGE_FETCH_PAIRS <sup>3</sup>	VARCHAR (520)		A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the table with this index using that hypothetical buffer. Zero-length string if no data is available.
MINPCTUSED	SMALLINT		A non-zero integer value indicates that the index is enabled for online defragmentation, and represents the minimum percentage of used space on a page before a page merge can be attempted. A zero value indicates that no page merge is attempted.
REVERSE_SCANS	CHAR (1)		<ul style="list-style-type: none"> <li>• N = Index does not support reverse scans</li> <li>• Y = Index supports reverse scans</li> </ul>
INTERNAL_FORMAT	SMALLINT		Possible values are: <ul style="list-style-type: none"> <li>• 1 = Index does not have backward pointers</li> <li>• 2 or greater = Index has backward pointers</li> <li>• 6 = Index is a composite block index</li> </ul>
IESHEMA	VARCHAR (128)	Y	Schema name of the index extension. Null for ordinary indexes.
IENAME	VARCHAR (128)	Y	Unqualified name of the index extension. Null for ordinary indexes.
IEARGUMENTS	CLOB (64K)	Y	External information of the parameter specified when the index is created. Null for ordinary indexes.
INDEX_OBJECTID	INTEGER		Identifier for the index object.
NUMRIDS	BIGINT		Total number of row identifiers (RIDs) or block identifiers (BIDs) in the index; -1 if not known.
NUMRIDS_DELETED	BIGINT		Total number of row identifiers (or block identifiers) in the index that are marked deleted, excluding those identifiers on leaf pages on which all the identifiers are marked deleted.
NUM_EMPTY_LEAFS	BIGINT		Total number of index leaf pages that have all of their row identifiers (or block identifiers) marked deleted.
AVERAGE_RANDOM_FETCH_PAGES <sup>1,2</sup>	DOUBLE		Average number of random table pages between sequential page accesses when fetching using the index; -1 if not known.
AVERAGE_RANDOM_PAGES <sup>2</sup>	DOUBLE		Average number of random table pages between sequential page accesses; -1 if not known.

## SYSCAT.INDEXES

Table 98. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
AVERAGE_SEQUENCE_GAP <sup>2</sup>	DOUBLE		Gap between index page sequences. Detected through a scan of index leaf pages, each gap represents the average number of index pages that must be randomly fetched between sequences of index pages; -1 if not known.
AVERAGE_SEQUENCE_FETCH_GAP <sup>1,2</sup>	DOUBLE		Gap between table page sequences when fetching using the index. Detected through a scan of index leaf pages, each gap represents the average number of table pages that must be randomly fetched between sequences of table pages; -1 if not known.
AVERAGE_SEQUENCE_PAGES <sup>2</sup>	DOUBLE		Average number of index pages that are accessible in sequence (that is, the number of index pages that the prefetchers would detect as being in sequence); -1 if not known.
AVERAGE_SEQUENCE_FETCH_PAGES <sup>1,2</sup>	DOUBLE		Average number of table pages that are accessible in sequence (that is, the number of table pages that the prefetchers would detect as being in sequence) when fetching using the index; -1 if not known.
TBSPACEID	INTEGER		Identifier for the index table space.
LEVEL2PCTFREE	SMALLINT		Percentage of each index level 2 page to be reserved during initial building of the index. This space is available for future inserts after the index has been built.
PAGESPLIT	CHAR (1)		Index page split behavior. <ul style="list-style-type: none"> <li>• H = High</li> <li>• L = Low</li> <li>• S = Symmetric</li> </ul>
AVGPARTITION_CLUSTERRATIO <sup>3</sup>	SMALLINT		Degree of data clustering within a single data partition. -1 if the table is not partitioned, if statistics are not collected, or if detailed statistics are collected (in which case AVGPARTITION_CLUSTERFACTOR will be used instead).
AVGPARTITION_CLUSTERFACTOR <sup>3</sup>	DOUBLE		Finer measurement of the degree of clustering within a single data partition. -1 if the table is not partitioned, if statistics are not collected, or if the index is defined on a nickname.
AVGPARTITION_PAGE_FETCH_PAIRS <sup>3</sup>	VARCHAR (520)		A list of paired integers in character form. Each pair represents a potential buffer pool size and the corresponding page fetches required to access a single data partition from the table. Zero-length string if no data is available, or if the table is not partitioned.
DATAPARTITION_CLUSTERFACTOR	DOUBLE		A statistic measuring the "clustering" of the index keys with regard to data partitions. It is a number between 0 and 1, with 1 representing perfect clustering and 0 representing no clustering.



Table 98. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
INDCARD	BIGINT		Cardinality of the index. This might be different from the cardinality of the table for indexes that do not have a one-to-one relationship between the table rows and the index entries.
OS_PTR_SIZE	INTEGER		Platform word size with which the index was created. <ul style="list-style-type: none"> <li>• 32 = 32-bit</li> <li>• 64 = 64-bit</li> </ul>
COLLECTSTATISTICS	CHAR (1)		Specifies how statistics were collected at index creation time. <ul style="list-style-type: none"> <li>• D = Collect detailed index statistics</li> <li>• S = Collect sampled detailed index statistics</li> <li>• Y = Collect basic index statistics</li> <li>• Blank = Do not collect index statistics</li> </ul>
DEFINER <sup>4</sup>	VARCHAR (128)		Authorization ID under which the index was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

**Note:**

1. When using DMS table spaces, this statistic cannot be computed.
2. Prefetch statistics are not gathered during a LOAD...STATISTICS YES, or a CREATE INDEX...COLLECT STATISTICS operation, or when the database configuration parameter *seqdetect* is turned off.
3. AVGPARTITION\_CLUSTERRATIO, AVGPARTITION\_CLUSTERFACTOR, and AVGPARTITION\_PAGE\_FETCH\_PAIRS measure the degree of clustering within a single data partition (local clustering). CLUSTERRATIO, CLUSTERFACTOR, and PAGE\_FETCH\_PAIRS measure the degree of clustering in the entire table (global clustering). Global clustering and local clustering values can diverge significantly if the table partitioning key is not a prefix of the index key, or when the table partitioning key and the index key are logically independent of each other.
4. The DEFINER column is included for backwards compatibility. See OWNER.
5. The XPTH, XRGN, and XVIP indexes are not recognized by any application programming interface that returns index metadata.

## SYSCAT.INDEXEXPLOITRULES

Each row represents an index exploitation rule.

Table 99. SYSCAT.INDEXEXPLOITRULES Catalog View

Column Name	Data Type	Nullable	Description
FUNCID	INTEGER		Identifier for the function.
SPECID	SMALLINT		Number of the predicate specification.
IESHEMA	VARCHAR (128)		Schema name of the index extension.
IENAME	VARCHAR (128)		Unqualified name of the index extension.
RULEID	SMALLINT		Identifier for the exploitation rule.
SEARCHMETHODID	SMALLINT		Identifier for the search method in the specific index extension.
SEARCHKEY	VARCHAR (640)		Key used to exploit the index.

## SYSCAT.INDEXEXPLOITRULES

Table 99. SYSCAT.INDEXEXPLOITRULES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SEARCHARGUMENT	VARCHAR (2700)		Search arguments used to exploit the index.
EXACT	CHAR (1)		<ul style="list-style-type: none"><li>• N = Index lookup is not exact in terms of predicate evaluation</li><li>• Y = Index lookup is exact in terms of predicate evaluation</li></ul>

## SYSCAT.INDEXEXTENSIONDEP

Each row represents a dependency of an index extension on some other object. The index extension depends on the object of type BTYPE of name BNAME, so a change to the object affects the index extension.

Table 100. SYSCAT.INDEXEXTENSIONDEP Catalog View

Column Name	Data Type	Nullable	Description
IESHEMA	VARCHAR (128)		Schema name of the index extension.
IENAME	VARCHAR (128)		Unqualified name of the index extension.
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: <ul style="list-style-type: none"><li>• A = Alias</li><li>• B = Trigger</li><li>• F = Routine instance</li><li>• H = Hierachy table</li><li>• K = Package</li><li>• L = Detached table</li><li>• O = Privilege dependency on all subtables or subviews in a table or view hierarchy</li><li>• Q = Sequence</li><li>• R = Structured type</li><li>• S = Materialized query table</li><li>• T = Table (not typed)</li><li>• U = Typed table</li><li>• V = View (not typed)</li><li>• W = Typed view</li><li>• X = Index extension</li><li>• Z = XSR object</li><li>• v = Global variable</li></ul>
BSHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.
TABAUTH	SMALLINT	Y	If BTYPE = 'O', 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by the dependent index extension; null value otherwise.

## SYSCAT.INDEXEXTENSIONMETHODS

Each row represents a search method. An index extension can contain more than one search method.

Table 101. SYSCAT.INDEXEXTENSIONMETHODS Catalog View

Column Name	Data Type	Nullable	Description
METHODNAME	VARCHAR (128)		Name of the search method.
METHODID	SMALLINT		Number of the method in the index extension.
IESHEMA	VARCHAR (128)		Schema name of the index extension on which this method is defined.
IENAME	VARCHAR (128)		Unqualified name of the index extension on which this method is defined.
RANGEFUNCSHEMA	VARCHAR (128)		Schema name of the range-through function.
RANGEFUNCNAME	VARCHAR (128)		Unqualified name of the range-through function.
RANGESPECIFICNAME	VARCHAR (128)		Function-specific name of the range-through function.
FILTERFUNCSHEMA	VARCHAR (128)	Y	Schema name of the filter function.
FILTERFUNCNAME	VARCHAR (128)	Y	Unqualified name of the filter function.
FILTERSPECIFICNAME	VARCHAR (128)	Y	Function-specific name of the filter function.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSCAT.INDEXEXTENSIONPARMS

Each row represents an index extension instance parameter or source key column.

Table 102. SYSCAT.INDEXEXTENSIONPARMS Catalog View

Column Name	Data Type	Nullable	Description
IESHEMA	VARCHAR (128)		Schema name of the index extension.
IENAME	VARCHAR (128)		Unqualified name of the index extension.
ORDINAL	SMALLINT		Sequence number of the parameter or key column.
PARAMNAME	VARCHAR (128)		Name of the parameter or key column.
TYPESHEMA	VARCHAR (128)		Schema name of the data type of the parameter or key column.
TYPENAME	VARCHAR (128)		Unqualified name of the data type of the parameter or key column.
LENGTH	INTEGER		Data type length of the parameter or key column.
SCALE	SMALLINT		Data type scale of the parameter or key column; 0 if not applicable.
PARMTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• K = Source key column</li> <li>• P = Index extension instance parameter</li> </ul>
CODEPAGE	SMALLINT		Code page of the index extension instance parameter; 0 if not a string type.

## SYSCAT.INDEXEXTENSIONPARMS

Table 102. SYSCAT.INDEXEXTENSIONPARMS Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLLATIONSCHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the parameter; null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the parameter; null value otherwise.

## SYSCAT.INDEXEXTENSIONS

Each row represents an index extension.

Table 103. SYSCAT.INDEXEXTENSIONS Catalog View

Column Name	Data Type	Nullable	Description
IESHEMA	VARCHAR (128)		Schema name of the index extension.
IENAME	VARCHAR (128)		Unqualified name of the index extension.
OWNER	VARCHAR (128)		Authorization ID under which the index extension was created.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• S = The owner is the system</li><li>• U = The owner is an individual user</li></ul>
CREATE_TIME	TIMESTAMP		Time at which the index extension was defined.
KEYGENFUNCSHEMA	VARCHAR (128)		Schema name of the key generation function.
KEYGENFUNCNAME	VARCHAR (128)		Unqualified name of the key generation function.
KEYGENSPECIFICNAME	VARCHAR (128)		Name of the key generation function instance (might be system-generated).
TEXT	CLOB (2M)		Full text of the CREATE INDEX EXTENSION statement.
DEFINER <sup>1</sup>	VARCHAR (128)		Authorization ID under which the index extension was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

**Note:**

1. The DEFINER column is included for backwards compatibility. See OWNER.

## SYSCAT.INDEXOPTIONS

Each row represents an index-specific option value.

Table 104. SYSCAT.INDEXOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR (128)		Schema name of the index.
INDNAME	VARCHAR (128)		Unqualified name of the index.
OPTION	VARCHAR (128)		Name of the index option.
SETTING	VARCHAR (2048)		Value of the index option.

---

## SYSCAT.INDEXXMLPATTERNS

Each row represents a pattern clause in an index over an XML column.

Table 105. SYSCAT.INDEXXMLPATTERNS Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR (128)		Schema name of the logical index.
INDNAME	VARCHAR (128)		Unqualified name of the logical index.
PINDNAME	VARCHAR (128)		Unqualified name of the physical index.
PINDID	SMALLINT		Identifier for the physical index.
TYPEMODEL	CHAR (1)		<ul style="list-style-type: none"> <li>• Q = SQL DATA TYPE (Ignore invalid values)</li> <li>• R = SQL DATA TYPE (Reject invalid values)</li> </ul>
DATATYPE	VARCHAR (128)		Name of the data type.
HASHED	CHAR (1)		Indicates whether or not the value is hashed. <ul style="list-style-type: none"> <li>• N = Not hashed</li> <li>• Y = Hashed</li> </ul>
LENGTH	SMALLINT		VARCHAR ( <i>n</i> ) length; 0 otherwise.
PATTERNID	SMALLINT		Identifier for the pattern.
PATTERN	CLOB (2M)	Y	Definition of the pattern.

**Note:**

1. When indexes over XML columns are created, logical indexes that utilize XML pattern information are created, resulting in the creation of physical B-tree indexes with DB2-generated key columns to support the logical indexes. A physical index is created to support the data type that is specified in the `xmltype`-clause of the `CREATE INDEX` statement.

---

## SYSCAT.KEYCOLUSE

Each row represents a column that participates in a key defined by a unique, primary key, or foreign key constraint.

Table 106. SYSCAT.KEYCOLUSE Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR (128)		Name of the constraint.
TABSCHEMA	VARCHAR (128)		Schema name of the table containing the column.
TABNAME	VARCHAR (128)		Unqualified name of the table containing the column.
COLNAME	VARCHAR (128)		Name of the column.
COLSEQ	SMALLINT		Numeric position of the column in the key (initial position is 1).

## SYSCAT.NAMEMAPPINGS

---

## SYSCAT.NAMEMAPPINGS

Each row represents the mapping between a "logical" object (typed table or view and its columns and indexes, including inherited columns) and the corresponding "implementation" object (hierarchy table or hierarchy view and its columns and indexes) that implements the logical object.

Table 107. SYSCAT.NAMEMAPPINGS Catalog View

Column Name	Data Type	Nullable	Description
TYPE	CHAR (1)		<ul style="list-style-type: none"><li>• C = Column</li><li>• I = Index</li><li>• U = Typed table</li></ul>
LOGICAL_SCHEMA	VARCHAR (128)		Schema name of the logical object.
LOGICAL_NAME	VARCHAR (128)		Unqualified name of the logical object.
LOGICAL_COLNAME	VARCHAR (128)	Y	Name of the logical column if TYPE = 'C'; null value otherwise.
IMPL_SCHEMA	VARCHAR (128)		Schema name of the implementation object that implements the logical object.
IMPL_NAME	VARCHAR (128)		Unqualified name of the implementation object that implements the logical object.
IMPL_COLNAME	VARCHAR (128)	Y	Name of the implementation column if TYPE = 'C'; null value otherwise.

---

## SYSCAT.NICKNAMES

Each row represents a nickname.

Table 108. SYSCAT.NICKNAMES Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the nickname.
TABNAME	VARCHAR (128)		Unqualified name of the nickname.
OWNER	VARCHAR (128)		User who created the nickname.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• S = The owner is the system</li><li>• U = The owner is an individual user</li></ul>
STATUS	CHAR (1)		Status of the object. <ul style="list-style-type: none"><li>• C = Set integrity pending</li><li>• N = Normal</li><li>• X = Inoperative</li></ul>
CREATE_TIME	TIMESTAMP		Time at which the object was created.
STATS_TIME	TIMESTAMP	Y	Time at which any change was last made to recorded statistics for this object. Null if statistics are not collected.
COLCOUNT	SMALLINT		Number of columns, including inherited columns (if any).
TABLEID	SMALLINT		Internal logical object identifier.
TBSPACEID	SMALLINT		Internal logical identifier for the primary table space for this object.

Table 108. SYSCAT.NICKNAMES Catalog View (continued)

Column Name	Data Type	Nullable	Description
CARD	BIGINT		Total number of rows; -1 if statistics are not collected.
NPAGES	BIGINT		Total number of pages on which the rows of the nickname exist; -1 if statistics are not gathered.
FPAGES	BIGINT		Total number of pages; -1 if statistics are not gathered.
OVERFLOW	BIGINT		Total number of overflow records; -1 if statistics are not gathered.
PARENTS	SMALLINT	Y	Number of parent tables for this object; that is, the number of referential constraints in which this object is a dependent.
CHILDREN	SMALLINT	Y	Number of dependent tables for this object; that is, the number of referential constraints in which this object is a parent.
SELFREFS	SMALLINT	Y	Number of self-referencing referential constraints for this object; that is, the number of referential constraints in which this object is both a parent and a dependent.
KEYCOLUMNS	SMALLINT	Y	Number of columns in the primary key.
KEYINDEXID	SMALLINT	Y	Index identifier for the primary key index; 0 or the null value if there is no primary key.
KEYUNIQUE	SMALLINT		Number of unique key constraints (other than the primary key constraint) defined on this object.
CHECKCOUNT	SMALLINT		Number of check constraints defined on this object.
DATA_CAPTURE	CHAR (1)		<ul style="list-style-type: none"> <li>• L = Nickname participates in data replication, including replication of LONG VARCHAR and LONG VARGRAPHIC columns</li> <li>• N = Nickname does not participate in data replication</li> <li>• Y = Nickname participates in data replication</li> </ul>

## SYSCAT.NICKNAMES

Table 108. SYSCAT.NICKNAMES Catalog View (continued)

Column Name	Data Type	Nullable	Description
CONST_CHECKED	CHAR (32)		<ul style="list-style-type: none"> <li>• Byte 1 represents foreign key constraint.</li> <li>• Byte 2 represents check constraint.</li> <li>• Byte 5 represents materialized query table.</li> <li>• Byte 6 represents generated column.</li> <li>• Byte 7 represents staging table.</li> <li>• Byte 8 represents data partitioning constraint.</li> <li>• Other bytes are reserved for future use.</li> </ul> <p>Possible values are:</p> <ul style="list-style-type: none"> <li>• F = In byte 5, the materialized query table cannot be refreshed incrementally. In byte 7, the content of the staging table is incomplete and cannot be used for incremental refresh of the associated materialized query table.</li> <li>• N = Not checked</li> <li>• U = Checked by user</li> <li>• W = Was in 'U' state when the table was placed in set integrity pending state</li> <li>• Y = Checked by system</li> </ul>
PARTITION_MODE	CHAR (1)		Reserved for future use.
STATISTICS_PROFILE	CLOB (10M)	Y	RUNSTATS command used to register a statistical profile for the object.
ACCESS_MODE	CHAR (1)		<p>Access restriction state of the object. These states only apply to objects that are in set integrity pending state or to objects that were processed by a SET INTEGRITY statement. Possible values are:</p> <ul style="list-style-type: none"> <li>• D = No data movement</li> <li>• F = Full access</li> <li>• N = No access</li> <li>• R = Read-only access</li> </ul>
CODEPAGE	SMALLINT		Code page of the object. This is the default code page used for all character columns, triggers, check constraints, and expression-generated columns.
REMOTE_TABLE	VARCHAR (128)	Y	Unqualified name of the specific data source object (such as a table or a view) for which the nickname was created.
REMOTE_SCHEMA	VARCHAR (128)	Y	Schema name of the specific data source object (such as a table or a view) for which the nickname was created.
SERVERNAME	VARCHAR (128)	Y	Name of the data source that contains the table or view for which the nickname was created.



Table 108. SYSCAT.NICKNAMES Catalog View (continued)

Column Name	Data Type	Nullable	Description
REMOTE_TYPE	CHAR (1)	Y	Type of object at the data source. <ul style="list-style-type: none"> <li>• A = Alias</li> <li>• N = Nickname</li> <li>• S = Materialized query table</li> <li>• T = Table (untyped)</li> <li>• V = View (untyped)</li> </ul>
CACHINGALLOWED	VARCHAR (1)		<ul style="list-style-type: none"> <li>• N = Caching is not allowed</li> <li>• Y = Caching is allowed</li> </ul>
DEFINER <sup>1</sup>	VARCHAR (128)		Authorization ID under which the table, view, alias, or nickname was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

**Note:**

1. The DEFINER column is included for backwards compatibility. See OWNER.

## SYSCAT.PACKAGEAUTH

Each row represents a user or group that has been granted one or more privileges on a package.

Table 109. SYSCAT.PACKAGEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• S = Grantor is the system</li> <li>• U = Grantor is an individual user</li> </ul>
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• G = Grantee is a group</li> <li>• R = Grantee is a role</li> <li>• U = Grantee is an individual user</li> </ul>
PKGSHEMA	VARCHAR (128)		Schema name of the package.
PKGNAME	VARCHAR (128)		Unqualified name of the package.
CONTROLAUTH	CHAR (1)		CONTROL privilege. <ul style="list-style-type: none"> <li>• N = Not held</li> <li>• Y = Held</li> </ul>
BINDAUTH	CHAR (1)		Privilege to bind the package. <ul style="list-style-type: none"> <li>• G = Held and grantable</li> <li>• N = Not held</li> <li>• Y = Held</li> </ul>
EXECUTEAUTH	CHAR (1)		Privilege to execute the package. <ul style="list-style-type: none"> <li>• G = Held and grantable</li> <li>• N = Not held</li> <li>• Y = Held</li> </ul>

---

**SYSCAT.PACKAGEDEP**

Each row represents a dependency of a package on some other object. The package depends on the object of type BTYPE of name BNAME, so a change to the object affects the package.

Table 110. SYSCAT.PACKAGEDEP Catalog View

Column Name	Data Type	Nullable	Description
PKGSHEMA	VARCHAR (128)		Schema name of the package.
PKGNAME	VARCHAR (128)		Unqualified name of the package.
BINDER	VARCHAR (128)		Binder of the package.
BINDERTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• U = Binder is an individual user</li> </ul>
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: <ul style="list-style-type: none"> <li>• A = Alias</li> <li>• B = Trigger</li> <li>• D = Server definition</li> <li>• F = Routine instance</li> <li>• I = Index</li> <li>• M = Function mapping</li> <li>• N = Nickname</li> <li>• O = Privilege dependency on all subtables or subviews in a table or view hierarchy</li> <li>• P = Page size</li> <li>• Q = Sequence object</li> <li>• R = User-defined data type</li> <li>• S = Materialized query table</li> <li>• T = Table (untyped)</li> <li>• U = Typed table</li> <li>• V = View (untyped)</li> <li>• W = Typed view</li> <li>• Z = XSR object</li> <li>• v = Global variable</li> </ul>
BSCHEMA	VARCHAR (128)		Schema name of an object on which the package depends.
BNAME	VARCHAR (128)		Unqualified name of an object on which the package depends.
TABAUTH	SMALLINT	Y	If BTYPE is 'O', 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges that are required by this package (SELECT, INSERT, UPDATE, or DELETE).
UNIQUE_ID	CHAR (8) FOR BIT DATA		Identifier for a specific package when multiple packages having the same name exist.
PKGVERSION	VARCHAR (64)	Y	Version identifier for the package.

**Note:**

1. If a function instance with dependencies is dropped, the package is put into an "inoperative" state, and it must be explicitly rebound. If any other object with dependencies is dropped, the package is put into an "invalid" state, and the system will attempt to rebind the package automatically when it is first referenced.

---

## SYSCAT.PACKAGES

Each row represents a package that has been created by binding an application program.

Table 111. SYSCAT.PACKAGES Catalog View

Column Name	Data Type	Nullable	Description
PKGSHEMA	VARCHAR (128)		Schema name of the package.
PKGNAME	VARCHAR (128)		Unqualified name of the package.
BOUNDBY	VARCHAR (128)		Authorization ID of the binder of the package.
BOUNDBYTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>U = The binder is an individual user</li> </ul>
OWNER	VARCHAR (128)		Authorization ID under which the package was bound.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>S = The owner is the system</li> <li>U = The owner is an individual user</li> </ul>
DEFAULT_SCHEMA	VARCHAR (128)		Default schema name used for unqualified names in static SQL statements.
VALID <sup>1</sup>	CHAR (1)		<ul style="list-style-type: none"> <li>N = Needs rebinding</li> <li>V = Validate at run time</li> <li>X = Package is inoperative because some function instance on which it depends has been dropped; explicit rebind is needed</li> <li>Y = Valid</li> </ul>
UNIQUE_ID	CHAR (8) FOR BIT DATA		Identifier for a specific package when multiple packages having the same name exist.
TOTAL_SECT	SMALLINT		Number of sections in the package.
FORMAT	CHAR (1)		<p>Date and time format associated with the package.</p> <ul style="list-style-type: none"> <li>0 = Format associated with the territory code of the client</li> <li>1 = USA</li> <li>2 = EUR</li> <li>3 = ISO</li> <li>4 = JIS</li> <li>5 = LOCAL</li> </ul>
ISOLATION	CHAR (2)	Y	<p>Isolation level.</p> <ul style="list-style-type: none"> <li>CS = Cursor Stability</li> <li>RR = Repeatable Read</li> <li>RS = Read Stability</li> <li>UR = Uncommitted Read</li> </ul>
BLOCKING	CHAR (1)	Y	<p>Cursor blocking option.</p> <ul style="list-style-type: none"> <li>B = Block all cursors</li> <li>N = No blocking</li> <li>U = Block unambiguous cursors</li> </ul>

## SYSCAT.PACKAGES

Table 111. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
INSERT_BUF	CHAR (1)		Setting of the INSERT bind option (applies to partitioned database systems). <ul style="list-style-type: none"> <li>• N = Inserts are not buffered</li> <li>• Y = Inserts are buffered at the coordinator database partition to minimize traffic among database partitions</li> </ul>
LANG_LEVEL	CHAR (1)	Y	Setting of the LANGLEVEL bind option. <ul style="list-style-type: none"> <li>• 0 = SAA1</li> <li>• 1 = MIA</li> <li>• 2 = SQL92E</li> </ul>
FUNC_PATH	CLOB (2K)		SQL path used by the last bind operation for this package. This is used as the default path for rebind operations. 'SYSIBM' for pre-Version 2 packages.
QUERYOPT	INTEGER		Optimization class under which this package was bound. Used for rebind operations.
EXPLAIN_LEVEL	CHAR (1)		Indicates whether Explain was requested using the EXPLAIN or EXPLSNAP bind option. <ul style="list-style-type: none"> <li>• P = Package selection level</li> <li>• Blank = No Explain requested</li> </ul>
EXPLAIN_MODE	CHAR (1)		Value of the EXPLAIN bind option. <ul style="list-style-type: none"> <li>• A = ALL</li> <li>• N = No</li> <li>• R = REOPT</li> <li>• Y = Yes</li> </ul>
EXPLAIN_SNAPSHOT	CHAR (1)		Value of the EXPLSNAP bind option. <ul style="list-style-type: none"> <li>• A = ALL</li> <li>• N = No</li> <li>• R = REOPT</li> <li>• Y = Yes</li> </ul>
SQLWARN	CHAR (1)		Indicates whether or not positive SQLCODEs resulting from dynamic SQL statements are returned to the application. <ul style="list-style-type: none"> <li>• N = No, they are suppressed</li> <li>• Y = Yes</li> </ul>
SQLMATHWARN	CHAR (1)		Value of the <i>dft_sqlmathwarn</i> database configuration parameter at bind time. Indicates whether arithmetic and retrieval conversion errors return warnings and null values (indicator -2), allowing query processing to continue whenever possible. <ul style="list-style-type: none"> <li>• N = No, errors are returned</li> <li>• Y = Yes, warnings are returned</li> </ul>

Table 111. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
EXPLICIT_BIND_TIME	TIMESTAMP		Time at which this package was last explicitly bound or rebound. When the package is implicitly rebound, no function instance that was created later than this time will be selected.
LAST_BIND_TIME	TIMESTAMP		Time at which the package was last explicitly or implicitly bound or rebound. Used to check the validity of Explain data.
CODEPAGE	SMALLINT		Application code page at bind time; -1 if not known.
COLLATIONSCHEMA	VARCHAR (128)		Schema name of the collation for the package.
COLLATIONNAME	VARCHAR (128)		Unqualified name of the collation for the package.
COLLATIONSCHEMA_ORDERBY	VARCHAR (128)		Schema name of the collation for ORDER BY clauses in the package.
COLLATIONNAME_ORDERBY	VARCHAR (128)		Unqualified name of the collation for ORDER BY clauses in the package.
DEGREE	CHAR (5)		Degree of intra-partition parallelism that was specified when the package was bound. <ul style="list-style-type: none"> <li>• 1 = No parallelism</li> <li>• 2-32767 = User-specified limit</li> <li>• ANY = Degree determined by the system (no limit specified)</li> </ul>
MULTINODE_PLANS	CHAR (1)		<ul style="list-style-type: none"> <li>• N = Package was not bound in a partitioned database environment</li> <li>• Y = Package was bound in a partitioned database environment</li> </ul>
INTRA_PARALLEL	CHAR (1)		Use of intra-partition parallelism by static SQL statements within the package. <ul style="list-style-type: none"> <li>• F = One or more static SQL statements in this package can use intra-partition parallelism; this parallelism has been disabled for use on a system that is not configured for intra-partition parallelism</li> <li>• N = No static SQL statement uses intra-partition parallelism</li> <li>• Y = One or more static SQL statements in the package use intra-partition parallelism</li> </ul>
VALIDATE	CHAR (1)		Indicates whether validity checking can be deferred until run time. <ul style="list-style-type: none"> <li>• B = All checking must be performed at bind time</li> <li>• R = Validation of tables, views, and privileges that do not exist at bind time is performed at run time</li> </ul>

## SYSCAT.PACKAGES

Table 111. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
DYNAMICRULES	CHAR (1)		<ul style="list-style-type: none"> <li>• B = BIND; dynamic SQL statements are executed with DYNAMICRULES BIND behavior</li> <li>• D = DEFINERBIND; when the package is run within a routine context, dynamic SQL statements in the package are executed with DEFINE behavior; when the package is not run within a routine context, dynamic SQL statements in the package are executed with BIND behavior</li> <li>• E = DEFINERRUN; when the package is run within a routine context, dynamic SQL statements in the package are executed with DEFINE behavior; when the package is not run within a routine context, dynamic SQL statements in the package are executed with RUN behavior</li> <li>• H = INVOKEBIND; when the package is run within a routine context, dynamic SQL statements in the package are executed with INVOKE behavior; when the package is not run within a routine context, dynamic SQL statements in the package are executed with BIND behavior</li> <li>• I = INVOKERUN; when the package is run within a routine context, dynamic SQL statements in the package are executed with INVOKE behavior; when the package is not run within a routine context, dynamic SQL statements in the package are executed with RUN behavior</li> <li>• R = RUN; dynamic SQL statements are executed with RUN behavior; this is the default</li> </ul>
SQLERROR	CHAR (1)		<p>SQLERROR option on the most recent subcommand that bound or rebound the package.</p> <ul style="list-style-type: none"> <li>• C = CONTINUE; creates a package, even if errors occur while binding SQL statements</li> <li>• N = NOPACKAGE; does not create a package or a bind file if an error occurs</li> </ul>
REFRESHAGE	DECIMAL (20,6)		Timestamp duration indicating the maximum length of time between execution of a REFRESH TABLE statement for a materialized query table (MQT) and when that MQT is used in place of a base table.
FEDERATED	CHAR (1)		<ul style="list-style-type: none"> <li>• N = FEDERATED bind or prep option is turned off</li> <li>• U = FEDERATED bind or prep option was not specified</li> <li>• Y = FEDERATED bind or prep option is turned on</li> </ul>

Table 111. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
TRANSFORMGROUP	VARCHAR (1024)	Y	Value of the TRANSFORM GROUP bind option; null if a transform group is not specified.
REOPTVAR	CHAR (1)		Indicates whether the access path is determined again at execution time using input variable values. <ul style="list-style-type: none"> <li>• A = Access path is reoptimized for every OPEN or EXECUTE request</li> <li>• N = Access path is determined at bind time</li> <li>• O = Access path is reoptimized only at the first OPEN or EXECUTE request; it is subsequently cached</li> </ul>
OS_PTR_SIZE	INTEGER		Word size for the platform on which the package was created. <ul style="list-style-type: none"> <li>• 32 = Package is a 32-bit package</li> <li>• 64 = Package is a 64-bit package</li> </ul>
PKGVERSION	VARCHAR (64)		Version identifier for the package.
PKG_CREATE_TIME	TIMESTAMP		Time at which the package was first bound.
STATICREADONLY	CHAR (1)		Indicates whether or not static cursors will be treated as READ ONLY. Possible values are: <ul style="list-style-type: none"> <li>• N = Static cursors take on the attributes that would normally be generated for the given statement text and the setting of the LANGLEVEL precompile option</li> <li>• Y = Any static cursor that does not contain the FOR UPDATE or the FOR READ ONLY clause is considered READ ONLY</li> </ul>
FEDERATED_ASYNCHRONY	INTEGER		Indicates the limit on asynchrony (the number of ATQs in the plan) as a bind option when the package was bound. <ul style="list-style-type: none"> <li>• 0 = No asynchrony</li> <li>• <i>n</i> = User-specified limit (32 767 maximum)</li> <li>• -1 = Degree of asynchrony determined by the system</li> <li>• -2 = Degree of asynchrony not specified</li> </ul> For a non-federated system, the value is 0.
OPTPROFILESCHEMA	VARCHAR (128)	Y	Value of the optimization profile schema specified as part of the OPTPROFILE bind option.
OPTPROFILENAME	VARCHAR (128)	Y	Value of the optimization profile name specified as part of the OPTPROFILE bind option.
DEFINER <sup>2</sup>	VARCHAR (128)		Authorization ID under which the package was bound.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSCAT.PACKAGES

Table 111. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
-------------	-----------	----------	-------------

**Note:**

1. If a function instance with dependencies is dropped, the package is put into an "inoperative" state, and it must be explicitly rebound. If any other object with dependencies is dropped, the package is put into an "invalid" state, and the system will attempt to rebind the package automatically when it is first referenced.
2. The DEFINER column is included for backwards compatibility. See OWNER.

---

## SYSCAT.PARTITIONMAPS

Each row represents a distribution map that is used to distribute the rows of a table among the database partitions in a database partition group, based on hashing the table's distribution key.

Table 112. SYSCAT.PARTITIONMAPS Catalog View

Column Name	Data Type	Nullable	Description
PMAP_ID	SMALLINT		Identifier for the distribution map.
PARTITIONMAP	BLOB (8192)		Distribution map, a vector of 4096 two-byte integers for a multiple partition database partition group. For a single partition database partition group, there is one entry denoting the partition number of the single partition.

---

## SYSCAT.PASSTHROUGHAUTH

Each row represents a user or group that has been granted pass-through authorization to query a data source.

Table 113. SYSCAT.PASSTHROUGHAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• S = Grantor is the system</li><li>• U = Grantor is an individual user</li></ul>
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		<ul style="list-style-type: none"><li>• G = Grantee is a group</li><li>• R = Grantee is a role</li><li>• U = Grantee is an individual user</li></ul>
SERVERNAME	VARCHAR (128)		Name of the data source to which authorization is being granted.

---

## SYSCAT.PREDICATESPECS

Each row represents a predicate specification.



Table 114. SYSCAT.PREDICATESPECS Catalog View

Column Name	Data Type	Nullable	Description
FUNCSCHEMA	VARCHAR (128)		Schema name of the function.
FUNCNAME	VARCHAR (128)		Unqualified name of the function.
SPECIFICNAME	VARCHAR (128)		Name of the function instance.
FUNCID	INTEGER		Identifier for the function.
SPECID	SMALLINT		Number of this predicate specification.
CONTEXTOP	CHAR (8)		Comparison operator, one of the built-in relational operators (=, <, >, >=, and so on).
CONTEXTEXP	CLOB (2M)		Constant, or an SQL expression.
FILTERTEXT	CLOB (32K)	Y	Text of the data filter expression.

## SYSCAT.REFERENCES

Each row represents a referential integrity (foreign key) constraint.

Table 115. SYSCAT.REFERENCES Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR (128)		Name of the constraint.
TABSCHEMA	VARCHAR (128)		Schema name of the dependent table.
TABNAME	VARCHAR (128)		Unqualified name of the dependent table.
OWNER	VARCHAR (128)		Authorization ID under which the constraint was created.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• S = The owner is the system</li> <li>• U = The owner is an individual user</li> </ul>
REFKEYNAME	VARCHAR (128)		Name of the parent key.
REFTABSCHEMA	VARCHAR (128)		Schema name of the parent table.
REFTABNAME	VARCHAR (128)		Unqualified name of the parent table.
COLCOUNT	SMALLINT		Number of columns in the foreign key.
DELETERULE	CHAR (1)		Delete rule. <ul style="list-style-type: none"> <li>• A = NO ACTION</li> <li>• C = CASCADE</li> <li>• N = SET NULL</li> <li>• R = RESTRICT</li> </ul>
UPDATERULE	CHAR (1)		Update rule. <ul style="list-style-type: none"> <li>• A = NO ACTION</li> <li>• R = RESTRICT</li> </ul>
CREATE_TIME	TIMESTAMP		Time at which the constraint was defined.
FK_COLNAMES	VARCHAR (640)		This column is no longer used and will be removed in a future release. Use SYSCAT.KEYCOLUSE for this information.
PK_COLNAMES	VARCHAR (640)		This column is no longer used and will be removed in a future release. Use SYSCAT.KEYCOLUSE for this information.

## SYSCAT.REFERENCES

Table 115. SYSCAT.REFERENCES Catalog View (continued)

Column Name	Data Type	Nullable	Description
DEFINER <sup>1</sup>	VARCHAR (128)		Authorization ID under which the constraint was created.

**Note:**

1. The DEFINER column is included for backwards compatibility. See OWNER.

## SYSCAT.ROLEAUTH

Each row represents a role granted to a user, group, role, or PUBLIC.

Table 116. SYSCAT.ROLEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Authorization ID that granted the role.
GRANTORTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• U = Grantor is an individual user</li></ul>
GRANTEE	VARCHAR (128)		Authorization ID to which the role was granted.
GRANTEETYPE	CHAR (1)		<ul style="list-style-type: none"><li>• G = The grantee is a group</li><li>• R = The grantee is a role</li><li>• U = The grantee is an individual user</li></ul>
ROLENAME	VARCHAR (128)		Name of the role.
ROLEID	INTEGER		Identifier for the role.
ADMIN	CHAR (1)		Privilege to grant or revoke the role to or from others, or to comment on the role. <ul style="list-style-type: none"><li>• N = Not held</li><li>• Y = Held</li></ul>

## SYSCAT.ROLES

Each row represents a role.

Table 117. SYSCAT.ROLES Catalog View

Column Name	Data Type	Nullable	Description
ROLENAME	VARCHAR (128)		Name of the role.
ROLEID	INTEGER		Identifier for the role.
CREATE_TIME	TIMESTAMP		Time when the role was created.
AUDITPOLICYID	INTEGER	Y	Identifier for the audit policy.
AUDITPOLICYNAME	VARCHAR (128)	Y	Name of the audit policy.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSCAT.ROUTINEAUTH

Each row represents a user or group that has been granted EXECUTE privilege on a particular routine (function, method, or procedure), or on all routines in a particular schema in the database.

Table 118. SYSCAT.ROUTINEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege. 'SYSIBM' if the privilege was granted by the system.
GRANTORTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• S = Grantor is the system</li> <li>• U = Grantor is an individual user</li> </ul>
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• G = Grantee is a group</li> <li>• R = Grantee is a role</li> <li>• U = Grantee is an individual user</li> </ul>
SCHEMA	VARCHAR (128)		Schema name of the routine.
SPECIFICNAME	VARCHAR (128)	Y	Specific name of the routine. If SPECIFICNAME is null and ROUTINETYPE is not 'M', the privilege applies to all routines of the type specified in ROUTINETYPE in the schema specified in SCHEMA. If SPECIFICNAME is null and ROUTINETYPE is 'M', the privilege applies to all methods for the subject type specified by TYPENAME in the schema specified by TYPESCHEMA. If SPECIFICNAME is null, ROUTINETYPE is 'M', and both TYPENAME and TYPESCHEMA are null, the privilege applies to all methods for all types in the schema.
TYPESCHEMA	VARCHAR (128)	Y	Schema name of the type for the method. Null if ROUTINETYPE is not 'M'.
TYPENAME	VARCHAR (128)	Y	Unqualified name of the type for the method. Null if ROUTINETYPE is not 'M'. If TYPENAME is null and ROUTINETYPE is 'M', the privilege applies to all methods for any subject type if they are in the schema specified by SCHEMA.
ROUTINETYPE	CHAR (1)		Type of the routine. <ul style="list-style-type: none"> <li>• F = Function</li> <li>• M = Method</li> <li>• P = Procedure</li> </ul>
EXECUTEAUTH	CHAR (1)		Privilege to execute the routine. <ul style="list-style-type: none"> <li>• G = Held and grantable</li> <li>• N = Not held</li> <li>• Y = Held</li> </ul>
GRANT_TIME	TIMESTAMP		Time at which the privilege was granted.

## SYSCAT.ROUTINEDEP

Each row represents a dependency of a routine on some other object. The routine depends on the object of type BTYPE of name BNAME, so a change to the object affects the routine.

## SYSCAT.ROUTINEDEP

Table 119. SYSCAT.ROUTINEDEP Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESHEMA	VARCHAR (128)		Schema name of the routine that has dependencies on another object.
SPECIFICNAME	VARCHAR (128)		Specific name of the routine that has dependencies on another object.
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: <ul style="list-style-type: none"><li>• A = Alias</li><li>• B = Trigger</li><li>• F = Routine instance</li><li>• H = Hierachy table</li><li>• K = Package</li><li>• L = Detached table</li><li>• O = Privilege dependency on all subtables or subviews in a table or view hierarchy</li><li>• Q = Sequence</li><li>• R = Structured type</li><li>• S = Materialized query table</li><li>• T = Table (not typed)</li><li>• U = Typed table</li><li>• V = View (not typed)</li><li>• W = Typed view</li><li>• X = Index extension</li><li>• Z = XSR object</li><li>• v = Global variable</li></ul>
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.
TABAUTH	SMALLINT	Y	If BTYPE = 'O', 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by the dependent routine; null value otherwise.
ROUTINENAME	VARCHAR (128)		This column is no longer used and will be removed in a future release. See SPECIFICNAME.

## SYSCAT.ROUTINEOPTIONS

Each row represents a routine-specific option value.

Table 120. SYSCAT.ROUTINEOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESHEMA	VARCHAR (128)		Schema name of the routine.
ROUTINENAME	VARCHAR (128)		Unqualified name of the routine.
SPECIFICNAME	VARCHAR (128)		Name of the routine instance (might be system-generated).

Table 120. SYSCAT.ROUTINEOPTIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
OPTION	VARCHAR (128)		Name of the federated routine option.
SETTING	VARCHAR (2048)		Value of the federated routine option.

## SYSCAT.ROUTINEPARMOPTIONS

Each row represents a routine parameter-specific option value.

Table 121. SYSCAT.ROUTINEPARMOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESCHEMA	VARCHAR (128)		Schema name of the routine.
ROUTINENAME	VARCHAR (128)		Unqualified name of the routine.
SPECIFICNAME	VARCHAR (128)		Name of the routine instance (might be system-generated).
ORDINAL	SMALLINT		Position of the parameter within the routine signature.
OPTION	VARCHAR (128)		Name of the federated routine option.
SETTING	VARCHAR (2048)		Value of the federated routine option.

## SYSCAT.ROUTINEPARMS

Each row represents a parameter or the result of a routine defined in SYSCAT.ROUTINES.

Table 122. SYSCAT.ROUTINEPARMS Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESCHEMA	VARCHAR (128)	Y	Schema name of the routine.
ROUTINENAME	VARCHAR (128)	Y	Unqualified name of the routine.
SPECIFICNAME	VARCHAR (128)	Y	Name of the routine instance (might be system-generated).
PARAMNAME	VARCHAR (128)	Y	Name of the parameter or result column, or null if no name exists.
ROWTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• B = Both input and output parameter</li> <li>• C = Result after casting</li> <li>• O = Output parameter</li> <li>• P = Input parameter</li> <li>• R = Result before casting</li> </ul>
ORDINAL	SMALLINT		If ROWTYPE = 'B', 'O', or 'P', numerical position of the parameter within the routine signature, starting with 1; if ROWTYPE = 'R' and the routine returns a table, numerical position of a named column in the result table, starting with 1; 0 otherwise.
TYPESCHEMA	VARCHAR (128)	Y	Schema name of the data type. The schema name for built-in types is 'SYSIBM'.
TYPENAME	VARCHAR (128)	Y	Unqualified name of the data type.

## SYSCAT.ROUTINEPARMS

Table 122. SYSCAT.ROUTINEPARMS Catalog View (continued)

Column Name	Data Type	Nullable	Description
LOCATOR	CHAR (1)		<ul style="list-style-type: none"> <li>N = Parameter or result is not passed in the form of a locator</li> <li>Y = Parameter or result is passed in the form of a locator</li> </ul>
LENGTH <sup>1</sup>	INTEGER		Length of the parameter or result; 0 if the parameter or result is a user-defined data type.
SCALE <sup>1</sup>	SMALLINT		Scale of the parameter or result; 0 if the parameter or result is a user-defined data type.
CODEPAGE	SMALLINT		Code page of this parameter or result; 0 denotes either not applicable, or a parameter or result for character data declared with the FOR BIT DATA attribute.
COLLATIONSCHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the parameter; null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the parameter; null value otherwise.
CAST_FUNCSCHEMA	VARCHAR (128)	Y	Schema name of the function used to cast an argument or a result. Applies to sourced and external functions; null value otherwise.
CAST_FUNCSPECIFIC	VARCHAR (128)	Y	Unqualified name of the function used to cast an argument or a result. Applies to sourced and external functions; null value otherwise.
TARGET_TYPESHEMA	VARCHAR (128)	Y	Schema name of the target type, if the type of the parameter or result is REFERENCE. Null value if the type of the parameter or result is not REFERENCE.
TARGET_TYPENAME	VARCHAR (128)	Y	Unqualified name of the target type, if the type of the parameter or result is REFERENCE. Null value if the type of the parameter or result is not REFERENCE.
SCOPE_TABSCHEMA	VARCHAR (128)	Y	Schema name of the scope (target table), if the parameter type is REFERENCE; null value otherwise.
SCOPE_TABNAME	VARCHAR (128)	Y	Unqualified name of the scope (target table), if the parameter type is REFERENCE; null value otherwise.
TRANSFORMGRPNAME	VARCHAR (128)	Y	Name of the transform group for a structured type parameter or result.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

**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.ROUTINES

Each row represents a user-defined routine (scalar function, table function, sourced function, method, or procedure). Does not include built-in functions.

Table 123. SYSCAT.ROUTINES Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESHEMA	VARCHAR (128)		Schema name of the routine.
ROUTINENAME	VARCHAR (128)		Unqualified name of the routine.
ROUTINETYPE	CHAR (1)		Type of routine. <ul style="list-style-type: none"> <li>• F = Function</li> <li>• M = Method</li> <li>• P = Procedure</li> </ul>
OWNER	VARCHAR (128)		Authorization ID under which the routine was created.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• S = The owner is the system</li> <li>• U = The owner is an individual user</li> </ul>
SPECIFICNAME	VARCHAR (128)		Name of the routine instance (might be system-generated).
ROUTINEID	INTEGER		Identifier for the routine.
RETURN_TYPESHEMA	VARCHAR (128)	Y	Schema name of the return type for a scalar function or method.
RETURN_TYPENAME	VARCHAR (128)	Y	Unqualified name of the return type for a scalar function or method.
ORIGIN	CHAR (1)		<ul style="list-style-type: none"> <li>• B = Built-in</li> <li>• E = User-defined, external</li> <li>• M = Template function</li> <li>• F = Federated procedure</li> <li>• Q = SQL-bodied<sup>1</sup></li> <li>• S = System-generated</li> <li>• T = System-generated transform function (not directly invocable)</li> <li>• U = User-defined, based on a source</li> </ul>
FUNCTIONTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• C = Column or aggregate</li> <li>• R = Row</li> <li>• S = Scalar</li> <li>• T = Table</li> <li>• Blank = Procedure</li> </ul>
PARAM_COUNT	SMALLINT		Number of routine parameters.
LANGUAGE	CHAR (8)		Implementation language for the routine body (or for the source function body, if this function is sourced on another function). Possible values are 'C', 'COBOL', 'JAVA', 'OLE', 'OLEDB', or 'SQL'. Blanks if ORIGIN is not 'E' or 'Q'.

## SYSCAT.ROUTINES

Table 123. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SOURCESCHEMA	VARCHAR (128)	Y	If ORIGIN = 'U' and the source function is a user-defined function, contains the schema name of the specific name of the source function. If ORIGIN = 'U' and the source function is a built-in function, contains the value 'SYSIBM'. Null if ORIGIN is not 'U'.
SOURCESPECIFIC	VARCHAR (128)	Y	If ORIGIN = 'U' and the source function is a user-defined function, contains the unqualified specific name of the source function. If ORIGIN = 'U' and the source function is a built-in function, contains the value 'N/A for built-in'. Null if ORIGIN is not 'U'.
DETERMINISTIC	CHAR (1)		<ul style="list-style-type: none"> <li>N = Results are not deterministic (same parameters might give different results in different routine calls)</li> <li>Y = Results are deterministic</li> <li>Blank = ORIGIN is not 'E', 'F', or 'Q'</li> </ul>
EXTERNAL_ACTION	CHAR (1)		<ul style="list-style-type: none"> <li>E = Function has external side-effects (therefore, the number of invocations is important)</li> <li>N = No side-effects</li> <li>Blank = ORIGIN is not 'E', 'F', or 'Q'</li> </ul>
NULLCALL	CHAR (1)		<ul style="list-style-type: none"> <li>N = RETURNS NULL ON NULL INPUT (function result is implicitly the null value if one or more operands are null)</li> <li>Y = CALLED ON NULL INPUT</li> <li>Blank = ORIGIN is not 'E' or 'Q'</li> </ul>
CAST_FUNCTION	CHAR (1)		<ul style="list-style-type: none"> <li>N = Not a cast function</li> <li>Y = Cast function</li> <li>Blank = ROUTINETYPE is not 'F'</li> </ul>
ASSIGN_FUNCTION	CHAR (1)		<ul style="list-style-type: none"> <li>N = Not an assignment function</li> <li>Y = Implicit assignment function</li> <li>Blank = ROUTINETYPE is not 'F'</li> </ul>
SCRATCHPAD	CHAR (1)		<ul style="list-style-type: none"> <li>N = Routine has no scratchpad</li> <li>Y = Routine has a scratchpad</li> <li>Blank = ORIGIN is not 'E' or ROUTINETYPE is 'P'</li> </ul>
SCRATCHPAD_LENGTH	SMALLINT		<p>Size (in bytes) of the scratchpad for the routine.</p> <ul style="list-style-type: none"> <li>-1 = LANGUAGE is 'OLEDB' and SCRATCHPAD is 'Y'</li> <li>0 = SCRATCHPAD is not 'Y'</li> </ul>



Table 123. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
FINALCALL	CHAR (1)		<ul style="list-style-type: none"> <li>• N = No final call is made</li> <li>• Y = Final call is made to this routine at the runtime end-of-statement</li> <li>• Blank = ORIGIN is not 'E' or ROUTINETYPE is 'P'</li> </ul>
PARALLEL	CHAR (1)		<ul style="list-style-type: none"> <li>• N = Routine cannot be executed in parallel</li> <li>• Y = Routine can be executed in parallel</li> <li>• Blank = ORIGIN is not 'E'</li> </ul>
PARAMETER_STYLE	CHAR (8)		<p>Parameter style that was declared when the routine was created. Possible values are:</p> <ul style="list-style-type: none"> <li>• DB2DARI</li> <li>• DB2GENRL</li> <li>• DB2SQL</li> <li>• GENERAL</li> <li>• GNRLNULL</li> <li>• JAVA</li> <li>• SQL</li> <li>• Blanks if ORIGIN is not 'E'</li> </ul>
FENCED	CHAR (1)		<ul style="list-style-type: none"> <li>• N = Not fenced</li> <li>• Y = Fenced</li> <li>• Blank = ORIGIN is not 'E'</li> </ul>
SQL_DATA_ACCESS	CHAR (1)		<p>Indicates what type of SQL statements, if any, the database manager should assume is contained in the routine.</p> <ul style="list-style-type: none"> <li>• C = Contains SQL (simple expressions with no subqueries only)</li> <li>• M = Contains SQL statements that modify data</li> <li>• N = Does not contain SQL statements</li> <li>• R = Contains read-only SQL statements</li> <li>• Blank = ORIGIN is not 'E', 'F', or 'Q'</li> </ul>
DBINFO	CHAR (1)		<p>Indicates whether a DBINFO parameter is passed to an external routine.</p> <ul style="list-style-type: none"> <li>• N = DBINFO is not passed</li> <li>• Y = DBINFO is passed</li> <li>• Blank = ORIGIN is not 'E'</li> </ul>
PROGRAMTYPE	CHAR (1)		<p>Indicates how the external routine is invoked.</p> <ul style="list-style-type: none"> <li>• M = Main</li> <li>• S = Subroutine</li> <li>• Blank = ORIGIN is 'F'</li> </ul>

## SYSCAT.ROUTINES

Table 123. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
COMMIT_ON_RETURN	CHAR (1)		<p>Indicates whether the transaction is committed on successful return from this procedure.</p> <ul style="list-style-type: none"> <li>• N = The unit of work is not committed</li> <li>• Y = The unit of work is committed</li> <li>• Blank = ROUTINETYPE is not 'P'</li> </ul>
RESULT_SETS	SMALLINT		Estimated maximum number of result sets.
SPEC_REG	CHAR (1)		<p>Indicates whether special registers are inherited from the caller or reinitialized to their default values when the routine is called.</p> <ul style="list-style-type: none"> <li>• D = Default special registers</li> <li>• I = Inherited special registers</li> <li>• Blank = PARAMETER_STYLE is 'DB2DARI' or ORIGIN is not 'E' or 'Q'</li> </ul>
FEDERATED	CHAR (1)		<p>Indicates whether or not federated objects can be accessed from the routine.</p> <ul style="list-style-type: none"> <li>• N = Federated objects cannot be accessed</li> <li>• Y = Federated objects can be accessed</li> <li>• Blank = ORIGIN is not 'E', 'F', or 'Q'</li> </ul>
THREADSAFE	CHAR (1)		<p>Indicates whether or not the routine can run in the same process as other routines.</p> <ul style="list-style-type: none"> <li>• N = Routine is not threadsafe</li> <li>• Y = Routine is threadsafe</li> <li>• Blank = ORIGIN is not 'E'</li> </ul>
VALID	CHAR (1)		<p>Applies to LANGUAGE = 'SQL' only; blank otherwise.</p> <ul style="list-style-type: none"> <li>• N = Routine needs rebinding</li> <li>• Y = Routine is valid</li> <li>• X = Routine is inoperative and must be recreated</li> </ul>
METHODIMPLEMENTED	CHAR (1)		<ul style="list-style-type: none"> <li>• N = Method body is not implemented</li> <li>• Y = Method body is implemented</li> <li>• Blank = ROUTINETYPE is not 'M'</li> </ul>
METHODEFFECT	CHAR (2)		<ul style="list-style-type: none"> <li>• CN = Constructor method</li> <li>• MU = Mutator method</li> <li>• OB = Observer method</li> <li>• Blanks = Not a system method</li> </ul>
TYPE_PRESERVING	CHAR (1)		<ul style="list-style-type: none"> <li>• N = Return type is the declared return type of the method</li> <li>• Y = Return type is governed by a "type-preserving" parameter; all system-generated mutator methods are type-preserving</li> <li>• Blank = ROUTINETYPE is not 'M'</li> </ul>

Table 123. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
WITH_FUNC_ACCESS	CHAR (1)		<ul style="list-style-type: none"> <li>• N = This method cannot be invoked by using functional notation</li> <li>• Y = This method can be invoked by using functional notation; that is, the "WITH FUNCTION ACCESS" attribute is specified</li> <li>• Blank = ROUTINETYPE is not 'M'</li> </ul>
OVERRIDDEN_METHODID	INTEGER	Y	Identifier for the overridden method when the OVERRIDING option is specified for a user-defined method. Null if ROUTINETYPE is not 'M'.
SUBJECT_TYPESHEMA	VARCHAR (128)	Y	Schema name of the subject type for the user-defined method. Null if ROUTINETYPE is not 'M'.
SUBJECT_TYPENAME	VARCHAR (128)	Y	Unqualified name of the subject type for the user-defined method. Null if ROUTINETYPE is not 'M'.
CLASS	VARCHAR (384)	Y	For LANGUAGE JAVA, CLR, or OLE, this is the class that implements this routine; null value otherwise.
JAR_ID	VARCHAR (128)	Y	For LANGUAGE JAVA, this is the JAR_ID of the installed jar file that implements this routine if a jar file was specified at routine creation time; null value otherwise. For LANGUAGE CLR, this is the assembly file that implements this routine.
JARSHEMA	VARCHAR (128)	Y	For LANGUAGE JAVA when a JAR_ID is present, this is the schema name of the jar file that implements this routine; null value otherwise.
JAR_SIGNATURE	VARCHAR (2048)	Y	For LANGUAGE JAVA, this is the method signature of this routine's specified Java method. For LANGUAGE CLR, this is a reference field for this CLR routine. Null value otherwise.
CREATE_TIME	TIMESTAMP		Time at which the routine was created.
ALTER_TIME	TIMESTAMP		Time at which the routine was last altered.
FUNC_PATH	CLOB (2K)	Y	SQL path at the time the routine was defined. Null if LANGUAGE is not 'SQL'.
QUALIFIER	VARCHAR (128)		Value of the default schema at the time of object definition. Used to complete any unqualified references.
IOS_PER_INVOC	DOUBLE		Estimated number of inputs/outputs (I/Os) per invocation; 0 is the default; -1 if not known.
INSTS_PER_INVOC	DOUBLE		Estimated number of instructions per invocation; 450 is the default; -1 if not known.

## SYSCAT.ROUTINES

Table 123. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
IOS_PER_ARGBYTE	DOUBLE		Estimated number of I/Os per input argument byte; 0 is the default; -1 if not known.
INSTS_PER_ARGBYTE	DOUBLE		Estimated number of instructions per input argument byte; 0 is the default; -1 if not known.
PERCENT_ARGBYTES	SMALLINT		Estimated average percent of input argument bytes that the routine will actually read; 100 is the default; -1 if not known.
INITIAL_IOS	DOUBLE		Estimated number of I/Os performed the first time that the routine is invoked; 0 is the default; -1 if not known.
INITIAL_INSTS	DOUBLE		Estimated number of instructions executed the first time the routine is invoked; 0 is the default; -1 if not known.
CARDINALITY	BIGINT		Predicted cardinality of a table function; -1 if not known, or if the routine is not a table function.
SELECTIVITY <sup>2</sup>	DOUBLE		For user-defined predicates; -1 if there are no user-defined predicates.
RESULT_COLS	SMALLINT		For a table function (ROUTINETYPE = 'F' and FUNCTIONTYPE = 'T'), contains the number of columns in the result table; for a procedure (ROUTINETYPE = 'P'), contains 0; contains 1 otherwise.
IMPLEMENTATION	VARCHAR (762)	Y	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 value otherwise.
LIB_ID	INTEGER	Y	Reserved for future use.
TEXT_BODY_OFFSET	INTEGER		If LANGUAGE = 'SQL', the offset to the start of the SQL procedure body in the full text of the CREATE statement; -1 if LANGUAGE is not 'SQL'.
TEXT	CLOB (2M)	Y	If LANGUAGE = 'SQL', the full text of the CREATE FUNCTION, CREATE METHOD, or CREATE PROCEDURE statement; null value otherwise.
NEWSAVEPOINTLEVEL	CHAR (1)		Indicates whether the routine initiates a new savepoint level when it is invoked. <ul style="list-style-type: none"> <li>• N = A new savepoint level is not initiated when the routine is invoked; the routine uses the existing savepoint level</li> <li>• Y = A new savepoint level is initiated when the routine is invoked</li> <li>• Blank = Not applicable</li> </ul>

Table 123. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
DEBUG_MODE <sup>3</sup>	VARCHAR (8)		Indicates whether or not the routine can be debugged using the DB2 debugger. <ul style="list-style-type: none"> <li>DISALLOW = Routine is not debuggable</li> <li>ALLOW = Routine is debuggable, and can participate in a client debug session with the DB2 debugger</li> <li>DISABLE = Routine is not debuggable, and this setting cannot be altered without dropping and recreating the routine</li> <li>Blank = Routine type is not currently supported by the DB2 debugger</li> </ul>
TRACE_LEVEL	VARCHAR (1)	Y	Reserved for future use.
DIAGNOSTIC_LEVEL	VARCHAR (1)	Y	Reserved for future use.
CHECKOUT_USERID	VARCHAR (128)	Y	ID of the user who performed a checkout of the object; null if the object is not checked out.
PRECOMPILE_OPTIONS	VARCHAR (1024)	Y	Precompile options specified for the routine.
COMPILE_OPTIONS	VARCHAR (1024)	Y	Compile options specified for the routine.
EXECUTION_CONTROL	CHAR (1)		Execution control mode of a common language runtime (CLR) routine. Possible values are: <ul style="list-style-type: none"> <li>N = Network</li> <li>R = Fileread</li> <li>S = Safe</li> <li>U = Unsafe</li> <li>W = Filewrite</li> <li>Blank = LANGUAGE is not 'CLR'</li> </ul>
CODEPAGE	SMALLINT		Routine code page, which specifies the default code page used for all character parameter types, result types, and local variables within the routine body.
COLLATIONSHEMA	VARCHAR (128)		Schema name of the collation for the routine.
COLLATIONNAME	VARCHAR (128)		Unqualified name of the collation for the routine.
COLLATIONSHEMA_ORDERBY	VARCHAR (128)		Schema name of the collation for ORDER BY clauses in the routine.
COLLATIONNAME_ORDERBY	VARCHAR (128)		Unqualified name of the collation for ORDER BY clauses in the routine.
ENCODING_SCHEME	CHAR (1)		Encoding scheme of the routine, as specified in the PARAMETER CCSID clause. Possible values are: <ul style="list-style-type: none"> <li>A = ASCII</li> <li>U = UNICODE</li> <li>Blank = PARAMETER CCSID clause was not specified</li> </ul>
LAST_REGEN_TIME	TIMESTAMP		Time at which the SQL routine packed descriptor was last regenerated.

## SYSCAT.ROUTINES

Table 123. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
INHERITLOCKREQUEST	CHAR (1)		<ul style="list-style-type: none"> <li>N = This function or method cannot be invoked in the context of an SQL statement that includes a lock-request-clause as part of a specified isolation-clause</li> <li>Y = This function or method inherits the isolation level of the invoking statement; it also inherits the specified lock-request-clause</li> <li>Blank = LANGUAGE is not 'SQL' or ROUTINETYPE is 'P'</li> </ul>
DEFINER <sup>4</sup>	VARCHAR (128)		Authorization ID under which the routine was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

**Note:**

- For SQL procedures created before Version 8.2 and migrated to Version 9, 'E' (instead of 'Q').
- During migration, the SELECTIVITY column will be set to -1 in the packed descriptor and system catalogs for all user-defined routines. For a user-defined predicate, the selectivity in the system catalog will be -1. In this case, the selectivity value used by the optimizer is 0.01.
- For Java routines, the DEBUG\_MODE setting does not indicate whether the Java routine was actually compiled in debug mode, or whether a debug Jar was installed at the server.
- The DEFINER column is included for backwards compatibility. See OWNER.

## SYSCAT.ROUTINESFEDERATED

Each row represents a federated procedure.

Table 124. SYSCAT.ROUTINESFEDERATED Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESHEMA	VARCHAR (128)		Schema name of the routine.
ROUTINENAME	VARCHAR (128)		Unqualified name of the routine.
ROUTINETYPE	CHAR (1)		Type of routine. <ul style="list-style-type: none"> <li>P = Procedure</li> </ul>
OWNER	VARCHAR (128)		Authorization ID under which the routine was created.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>S = The owner is the system</li> <li>U = The owner is an individual user</li> </ul>
SPECIFICNAME	VARCHAR (128)		Name of the routine instance (might be system-generated).
ROUTINEID	INTEGER		Identifier for the routine.
PARAM_COUNT	SMALLINT		Number of routine parameters.
DETERMINISTIC	CHAR (1)		<ul style="list-style-type: none"> <li>N = Results are not deterministic (same parameters might give different results in different routine calls)</li> <li>Y = Results are deterministic</li> </ul>

Table 124. SYSCAT.ROUTINESFEDERATED Catalog View (continued)

Column Name	Data Type	Nullable	Description
EXTERNAL_ACTION	CHAR (1)		<ul style="list-style-type: none"> <li>E = Routine has external side-effects (therefore, the number of invocations is important)</li> <li>N = No side-effects</li> </ul>
SQL_DATA_ACCESS	CHAR (1)		<p>Indicates what type of SQL statements, if any, the database manager should assume is contained in the routine.</p> <ul style="list-style-type: none"> <li>C = Contains SQL (simple expressions with no subqueries only)</li> <li>M = Contains SQL statements that modify data</li> <li>N = Does not contain SQL statements</li> <li>R = Contains read-only SQL statements</li> </ul>
COMMIT_ON_RETURN	CHAR (1)		<p>Indicates whether the transaction is committed on successful return from this procedure.</p> <ul style="list-style-type: none"> <li>N = The unit of work is not committed</li> <li>Y = The unit of work is committed</li> <li>Blank = ROUTINETYPE is not 'P'</li> </ul>
RESULT_SETS	SMALLINT		Estimated maximum number of result sets.
CREATE_TIME	TIMESTAMP		Time at which the routine was created.
ALTER_TIME	TIMESTAMP		Time at which the routine was last altered.
QUALIFIER	VARCHAR (128)		Value of the default schema at the time of object definition. Used to complete any unqualified references.
RESULT_COLS	SMALLINT		For a procedure (ROUTINETYPE = 'P'), contains 0; contains 1 otherwise.
CODEPAGE	SMALLINT		Routine code page, which specifies the default code page used for all character parameter types, result types, and local variables within the routine body.
LAST_REGEN_TIME	TIMESTAMP		Time at which the SQL routine packed descriptor was last regenerated.
REMOTE_PROCEDURE	VARCHAR (128)	Y	Unqualified name of the source procedure for which the federated routine was created.
REMOTE_SCHEMA	VARCHAR (128)	Y	Schema name of the source procedure for which the federated routine was created.
SERVERNAME	VARCHAR (128)	Y	Name of the data source that contains the source procedure for which the federated routine was created.
REMOTE_PACKAGE	VARCHAR (128)	Y	Name of the package to which the source procedure belongs (applies only to wrappers for Oracle data sources).
REMOTE_PROCEDURE_ALTER_TIME	VARCHAR (128)	Y	Reserved for future use.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSCAT.SCHEMAAUTH

---

## SYSCAT.SCHEMAAUTH

Each row represents a user or group that has been granted one or more privileges on a schema.

Table 125. SYSCAT.SCHEMAAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of a privilege.
GRANTORTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• S = Grantor is the system</li><li>• U = Grantor is an individual user</li></ul>
GRANTEE	VARCHAR (128)		Holder of a privilege.
GRANTEETYPE	CHAR (1)		<ul style="list-style-type: none"><li>• G = Grantee is a group</li><li>• R = Grantee is a role</li><li>• U = Grantee is an individual user</li></ul>
SCHEMANAME	VARCHAR (128)		Name of the schema to which this privilege applies.
ALTERINAUTH	CHAR (1)		Privilege to alter or comment on objects in the named schema. <ul style="list-style-type: none"><li>• G = Held and grantable</li><li>• N = Not held</li><li>• Y = Held</li></ul>
CREATEINAUTH	CHAR (1)		Privilege to create objects in the named schema. <ul style="list-style-type: none"><li>• G = Held and grantable</li><li>• N = Not held</li><li>• Y = Held</li></ul>
DROPINAUTH	CHAR (1)		Privilege to drop objects from the named schema. <ul style="list-style-type: none"><li>• G = Held and grantable</li><li>• N = Not held</li><li>• Y = Held</li></ul>

---

## SYSCAT.SCHEMATA

Each row represents a schema.

Table 126. 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, who has the authority to drop the schema and all objects within it. The value for implicitly created schemas is 'SYSIBM'.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• S = The owner is the system</li><li>• U = The owner is an individual user</li></ul>
DEFINER	VARCHAR (128)		Authorization ID under which the schema was created.



Table 126. SYSCAT.SCHEMATA Catalog View (continued)

Column Name	Data Type	Nullable	Description
DEFINERTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• S = The definer is the system</li> <li>• U = The definer is an individual user</li> </ul>
CREATE_TIME	TIMESTAMP		Time at which the schema was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSCAT.SECURITYLABELACCESS

Each row represents a security label that was granted to the database authorization ID.

Table 127. SYSCAT.SECURITYLABELACCESS Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the security label.
GRANTEE	VARCHAR (128)		Holder of the security label.
GRANTEETYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• U = Grantee is an individual user</li> <li>• R = Grantee is a user role</li> <li>• G = Grantee is a group user</li> </ul>
SECLABELID	INTEGER		Identifier for the security label. For the name of the security label, select the SECLABELNAME column for the corresponding SECLABELID value in the SYSCAT.SECURITYLABELS catalog view.
SECPOLICYID	INTEGER		Identifier for the security policy that is associated with the security label. For the name of the security policy, select the SECPOLICYNAME column for the corresponding SECPOLICYID value in the SYSCAT.SECURITYPOLICIES catalog view.
ACCESSTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• B = Both read and write access</li> <li>• R = Read access</li> <li>• W = Write access</li> </ul>
GRANT_TIME	TIMESTAMP		Time at which the security label was granted.

## SYSCAT.SECURITYLABELCOMPONENTELEMENTS

Each row represents an element value for a security label component.

Table 128. SYSCAT.SECURITYLABELCOMPONENTELEMENTS Catalog View

Column Name	Data Type	Nullable	Description
COMPID	INTEGER		Identifier for the security label component.
ELEMENTVALUE	VARCHAR (32)		Element value for the security label component.
ELEMENTVALUEENCODING	CHAR (8) FOR BIT DATA		Encoded form of the element value.

## SYSCAT.SECURITYLABELCOMPONENTELEMENTS

Table 128. SYSCAT.SECURITYLABELCOMPONENTELEMENTS Catalog View (continued)

Column Name	Data Type	Nullable	Description
PARENTELEMENTVALUE	VARCHAR (32)	Y	Name of the parent of an element for tree components; null for set and array components, and for the ROOT node of a tree component.

## SYSCAT.SECURITYLABELCOMPONENTS

Each row represents a security label component.

Table 129. SYSCAT.SECURITYLABELCOMPONENTS Catalog View

Column Name	Data Type	Nullable	Description
COMPNAME	VARCHAR (128)		Name of the security label component.
COMPID	INTEGER		Identifier for the security label component.
COMPTYPE	CHAR (1)		Security label component type. <ul style="list-style-type: none"><li>• A = Array</li><li>• S = Set</li><li>• T = Tree</li></ul>
NUMELEMENTS	INTEGER		Number of elements in the security label component.
CREATE_TIME	TIMESTAMP		Time at which the security label component was created.
REMARKS	VARCHAR (254)		User-provided comments, or null.

## SYSCAT.SECURITYLABELS

Each row represents a security label.

Table 130. SYSCAT.SECURITYLABELS Catalog View

Column Name	Data Type	Nullable	Description
SECLABELNAME	VARCHAR (128)		Name of the security label.
SECLABELID	INTEGER		Identifier for the security label.
SECPOLICYID	INTEGER		Identifier for the security policy to which the security label belongs.
SECLABEL	SYSPROC. DB2SECURITYLABEL		Internal representation of the security label.
CREATE_TIME	TIMESTAMP		Time at which the security label was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSCAT.SECURITYPOLICIES

Each row represents a security policy.

Table 131. SYSCAT.SECURITYPOLICIES Catalog View

Column Name	Data Type	Nullable	Description
SECPOLICYNAME	VARCHAR (128)		Name of the security policy.

Table 131. SYSCAT.SECURITYPOLICIES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SECPOLICYID	INTEGER		Identifier for the security policy.
NUMSECLABELCOMP	INTEGER		Number of security label components in the security policy.
RWSECLABELREL	CHAR (1)		Relationship between the security labels for read and write access granted to the same authorization ID. <ul style="list-style-type: none"> <li>• S = The security label for write access granted to a user is a subset of the security label for read access granted to that same user</li> </ul>
NOTAUTHWRITESECLABEL	CHAR (1)		Action to take when a user is not authorized to write the security label that is specified in the INSERT or UPDATE statement. <ul style="list-style-type: none"> <li>• O = Override</li> <li>• R = Restrict</li> </ul>
CREATE_TIME	TIMESTAMP		Time at which the security policy was created.
USERAUTHS	CHAR (1)		Indicates if authorizations of security labels and exemptions granted to an authorization ID that represents a user will be used or ignored. <ul style="list-style-type: none"> <li>• I = Ignored</li> <li>• U = Used</li> </ul>
GROUPAUTHS	CHAR (1)		Indicates if authorizations of security labels and exemptions granted to an authorization ID that represents a group will be used or ignored. <ul style="list-style-type: none"> <li>• I = Ignored</li> <li>• U = Used</li> </ul>
ROLEAUTHS	CHAR (1)		Indicates if authorizations of security labels and exemptions granted to an authorization ID that represents a role will be used or ignored. <ul style="list-style-type: none"> <li>• I = Ignored</li> <li>• U = Used</li> </ul>
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSCAT. SECURITYPOLICYCOMPONENTRULES

Each row represents the read and write access rules for a security label component of the security policy.

Table 132. SYSCAT.SECURITYPOLICYCOMPONENTRULES Catalog View

Column Name	Data Type	Nullable	Description
SECPOLICYID	INTEGER		Identifier for the security policy.
COMPID	INTEGER		Identifier for the security label component of the security policy.

## SYSCAT.SECURITYPOLICYCOMPONENTRULES

Table 132. SYSCAT.SECURITYPOLICYCOMPONENTRULES Catalog View (continued)

Column Name	Data Type	Nullable	Description
ORDINAL	INTEGER		Position of the security label component as it appears in the security policy, starting with 1.
READACCESSRULENAME	VARCHAR (128)		Name of the read access rule that is associated with the security label component.
READACCESSRULETEXT	VARCHAR (512)		Text of the read access rule that is associated with the security label component.
WRITEACCESSRULENAME	VARCHAR (128)		Name of the write access rule that is associated with the security label component.
WRITEACCESSRULETEXT	VARCHAR (512)		Text of the write access rule that is associated with the security label component.

## SYSCAT.SECURITYPOLICYEXEMPTIONS

Each row represents a security policy exemption that was granted to a database authorization ID.

Table 133. SYSCAT.SECURITYPOLICYEXEMPTIONS Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the exemption.
GRANTEE	VARCHAR (128)		Holder of the exemption.
GRANTEETYPE	CHAR (1)		<ul style="list-style-type: none"><li>• U = Grantee is an individual user</li><li>• R = Grantee is a user role</li><li>• G = Grantee is a group user</li></ul>
SECPOLICYID	INTEGER		Identifier for the security policy for which the exemption was granted. For the name of the security policy, select the SECPOLICYNAME column for the corresponding SECPOLICYID value in the SYSCAT.SECURITYPOLICIES catalog view.
ACCESSRULENAME	VARCHAR (128)		Name of the access rule for which the exemption was granted.
ACCESSTYPE	CHAR (1)		Type of access to which the rule applies. <ul style="list-style-type: none"><li>• R = Read access</li><li>• W = Write access</li></ul>
ORDINAL	INTEGER		Position of the security label component in the security policy to which the rule applies.
ACTIONALLOWED	CHAR (1)		If the rule is DB2LBACWRITEARRAY, then: <ul style="list-style-type: none"><li>• D = Write down</li><li>• U = Write up</li></ul> Blank otherwise.
GRANT_TIME	TIMESTAMP		Time at which the exemption was granted.

## SYSCAT.SEQUENCEAUTH

Each row represents a user or group that has been granted one or more privileges on a sequence.

Table 134. SYSCAT.SEQUENCEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of a privilege.
GRANTORTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>S = Grantor is the system</li> <li>U = Grantor is an individual user</li> </ul>
GRANTEE	VARCHAR (128)		Holder of a privilege.
GRANTEETYPE	CHAR (1)		<ul style="list-style-type: none"> <li>G = Grantee is a group</li> <li>R = Grantee is a role</li> <li>U = Grantee is an individual user</li> </ul>
SEQSCHEMA	VARCHAR (128)		Schema name of the sequence.
SEQNAME	VARCHAR (128)		Unqualified name of the sequence.
ALTERAUTH	CHAR (1)		Privilege to alter the sequence. <ul style="list-style-type: none"> <li>G = Held and grantable</li> <li>N = Not held</li> <li>Y = Held</li> </ul>
USAGEAUTH	CHAR (1)		Privilege to reference the sequence. <ul style="list-style-type: none"> <li>G = Held and grantable</li> <li>N = Not held</li> <li>Y = Held</li> </ul>

## SYSCAT.SEQUENCES

Each row represents a sequence.

Table 135. SYSCAT.SEQUENCES Catalog View

Column Name	Data Type	Nullable	Description
SEQSCHEMA	VARCHAR (128)		Schema name of the sequence.
SEQNAME	VARCHAR (128)		Unqualified name of the sequence.
DEFINER <sup>1</sup>	VARCHAR (128)		Authorization ID under which the sequence was created.
DEFINERTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>S = The definer is the system</li> <li>U = The definer is a individual user</li> </ul>
OWNER	VARCHAR (128)		Authorization ID under which the sequence was created.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>S = The owner is the system</li> <li>U = The owner is an individual user</li> </ul>
SEQID	INTEGER		Identifier for the sequence.
SEQTYPE	CHAR (1)		Type of sequence. <ul style="list-style-type: none"> <li>I = Identity sequence</li> <li>S = Regular sequence</li> </ul>

## SYSCAT.SEQUENCES

Table 135. SYSCAT.SEQUENCES Catalog View (continued)

Column Name	Data Type	Nullable	Description
INCREMENT	DECIMAL (31,0)		Increment value.
START	DECIMAL (31,0)		Start value of the sequence.
MAXVALUE	DECIMAL (31,0)		Maximum value of the sequence.
MINVALUE	DECIMAL (31,0)		Minimum value of the sequence.
NEXTCACHEFIRSTVALUE	DECIMAL (31,0)	Y	The first value available to be assigned in the next cache block. If no caching, the next value available to be assigned.
CYCLE	CHAR (1)		Indicates whether or not the sequence can continue to generate values after reaching its maximum or minimum value. <ul style="list-style-type: none"> <li>• N = Sequence cannot cycle</li> <li>• Y = Sequence can cycle</li> </ul>
CACHE	INTEGER		Number of sequence values to pre-allocate in memory for faster access. 0 indicates that values of the sequence are not to be preallocated. In a partitioned database, this value applies to each database partition.
ORDER	CHAR (1)		Indicates whether or not the sequence numbers must be generated in order of request. <ul style="list-style-type: none"> <li>• N = Sequence numbers are not required to be generated in order of request</li> <li>• Y = Sequence numbers must be generated in order of request</li> </ul>
DATATYPEID	INTEGER		For built-in types, the internal identifier of the built-in type. For distinct types, the internal identifier of the distinct type.
SOURCETYPEID	INTEGER		For a built-in type, this has a value of 0. For a distinct type, this is the internal identifier of the built-in type that is the source type for the distinct type.
CREATE_TIME	TIMESTAMP		Time at which the sequence was created.
ALTER_TIME	TIMESTAMP		Time at which the sequence was last altered.
PRECISION	SMALLINT		Precision of the data type of the sequence. Possible values are: <ul style="list-style-type: none"> <li>• 5 = SMALLINT</li> <li>• 10 = INTEGER</li> <li>• 19 = BIGINT</li> </ul> For DECIMAL, it is the precision of the specified DECIMAL data type.
ORIGIN	CHAR (1)		Origin of the sequence. <ul style="list-style-type: none"> <li>• S = System-generated sequence</li> <li>• U = User-generated sequence</li> </ul>
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

**Note:**

1. The DEFINER column is included for backwards compatibility. See OWNER.

---

## SYSCAT.SERVEROPTIONS

Each row represents a server-specific option value.

Table 136. SYSCAT.SERVEROPTIONS Catalog View

Column Name	Data Type	Nullable	Description
WRAPNAME	VARCHAR (128)	Y	Name of the wrapper.
SERVERNAME	VARCHAR (128)	Y	Uppercase name of the server.
SERVERTYPE	VARCHAR (30)	Y	Type of server.
SERVERVERSION	VARCHAR (18)	Y	Server version.
CREATE_TIME	TIMESTAMP		Time at which the entry was created.
OPTION	VARCHAR (128)		Name of the server option.
SETTING	VARCHAR (2048)		Value of the server option.
SERVEROPTIONKEY	VARCHAR (18)		Uniquely identifies a row.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

---

## SYSCAT.SERVERS

Each row represents a data source.

Table 137. SYSCAT.SERVERS Catalog View

Column Name	Data Type	Nullable	Description
WRAPNAME	VARCHAR (128)		Name of the wrapper.
SERVERNAME	VARCHAR (128)		Uppercase name of the server.
SERVERTYPE	VARCHAR (30)	Y	Type of server.
SERVERVERSION	VARCHAR (18)	Y	Server version.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

---

## SYSCAT.SERVICECLASSES

Each row represents a service class.

Table 138. SYSCAT.SERVICECLASSES Catalog View

Column Name	Data Type	Nullable	Description
SERVICECLASSNAME	VARCHAR (128)		Name of the service class.
PARENTSERVICECLASSNAME	VARCHAR (128)	Y	Service class name of the parent service superclass.
SERVICECLASSID	SMALLINT		Identifier for the service class.
PARENTID	SMALLINT		Identifier for the parent service class for this service class. 0 if this service class is a super service class.
CREATE_TIME	TIMESTAMP		Time when the service class was created.
ALTER_TIME	TIMESTAMP		Time when the service class was last altered.

## SYSCAT.SERVICECLASSES

Table 138. SYSCAT.SERVICECLASSES Catalog View (continued)

Column Name	Data Type	Nullable	Description
ENABLED	CHAR (1)		State of the service class. <ul style="list-style-type: none"><li>• N = Disabled</li><li>• Y = Enabled</li></ul>
AGENTPRIORITY	SMALLINT		Thread priority of the agents in the service class relative to the normal priority of DB2 threads. <ul style="list-style-type: none"><li>• -20 to 20 (Linux and UNIX)</li><li>• -6 to 6 (Windows)</li><li>• -32768 = not set</li></ul>
PREFETCHPRIORITY	CHAR (1)		Prefetch priority of the agents in the service class. <ul style="list-style-type: none"><li>• H = High</li><li>• L = Low</li><li>• M = Medium</li><li>• Blank = not set</li></ul>
INBOUNDCORRELATOR	VARCHAR (128)	Y	For future use.
OUTBOUNDCORRELATOR	VARCHAR (128)	Y	String used to associate the service class with an operating system workload manager service class.
COLLECTAGGACTDATA	CHAR (1)		Specifies what aggregate activity data should be captured for the service class by the applicable event monitor. <ul style="list-style-type: none"><li>• B = Collect base aggregate activity data</li><li>• E = Collect extended aggregate activity data</li><li>• N = None</li></ul>
COLLECTAGGREQDATA	CHAR (1)		Specifies what aggregate activity data should be captured for the service class by the applicable event monitor. <ul style="list-style-type: none"><li>• B = Collect base aggregate request data</li><li>• N = None</li></ul>
COLLECTACTDATA	CHAR (1)		Specifies what activity data should be collected by the applicable event monitor. <ul style="list-style-type: none"><li>• D = Activity data with details</li><li>• N = None</li><li>• V = Activity data with details and values</li><li>• W = Activity data without details</li></ul>
COLLECTACTPARTITION	CHAR (1)		Specifies where activity data is collected. <ul style="list-style-type: none"><li>• C = Database partition of the coordinator of the activity</li><li>• D = All database partitions</li></ul>
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSCAT.STATEMENTS

Each row represents an SQL statement in a package.



Table 139. SYSCAT.STATEMENTS Catalog View

Column Name	Data Type	Nullable	Description
PKGSHEMA	VARCHAR (128)		Schema name of the package.
PKGNAME	VARCHAR (128)		Unqualified name of the package.
STMTNO	INTEGER		Line number of the SQL statement in the source module of the application program.
SECTNO	SMALLINT		Number of the package section containing the SQL statement.
SEQNO	INTEGER		Always 1.
TEXT	CLOB (2M)		Text of the SQL statement.
UNIQUE_ID	CHAR (8) FOR BIT DATA		Identifier for a specific package when multiple packages having the same name exist.
VERSION	VARCHAR (64)	Y	Version identifier for the package.

## SYSCAT.SURROGATEAUTHIDS

Each row represents a user or a group that has been granted SETSESSIONUSER privilege on a user or PUBLIC.

Table 140. SYSCAT.SURROGATEAUTHIDS Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Authorization ID that granted TRUSTEDID the ability to act as a surrogate. When the TRUSTEDID represents a trusted context object, this field represents the authorization ID that created or altered the trusted context object.
TRUSTEDID	VARCHAR (128)		Identifier for the entity that is trusted to act as a surrogate.
TRUSTEDIDTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• C = Trusted context</li> <li>• G = Group</li> <li>• U = User</li> </ul>
SURROGATEAUTHID	VARCHAR (128)		Surrogate authorization ID that can be assumed by TRUSTEDID. 'PUBLIC' indicates that TRUSTEDID can assume any authorization ID.
SURROGATEAUTHIDTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• G = Group</li> <li>• U = User</li> </ul>
AUTHENTICATE	CHAR (1)		<ul style="list-style-type: none"> <li>• N = No authentication is required</li> <li>• Y = Authentication token is required with the authorization ID to authenticate the user before the authorization ID can be assumed</li> <li>• Blank = TRUSTEDIDTYPE is not 'C'</li> </ul>

## SYSCAT.SURROGATEAUTHIDS

Table 140. SYSCAT.SURROGATEAUTHIDS Catalog View (continued)

Column Name	Data Type	Nullable	Description
CONTEXTROLE	VARCHAR (128)	Y	A specific role to be assigned to the assumed authorization ID, which supercedes the default role, if any, that is defined for the trusted context. Null value when TRUSTEDIDTYPE is not 'C'.
GRANT_TIME	TIMESTAMP		Time at which the grant was made.

## SYSCAT.TABAUTH

Each row represents a user or group that has been granted one or more privileges on a table or view.

Table 141. SYSCAT.TABAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• S = Grantor is the system</li><li>• U = Grantor is an individual user</li></ul>
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		<ul style="list-style-type: none"><li>• G = Grantee is a group</li><li>• R = Grantee is a role</li><li>• U = Grantee is an individual user</li></ul>
TABSCHEMA	VARCHAR (128)		Schema name of the table or view.
TABNAME	VARCHAR (128)		Unqualified name of the table or view.
CONTROLAUTH	CHAR (1)		CONTROL privilege. <ul style="list-style-type: none"><li>• N = Not held</li><li>• Y = Held but not grantable</li></ul>
ALTERAUTH	CHAR (1)		Privilege to alter the table; allow a parent table to this table to drop its primary key or unique constraint; allow a table to become a materialized query table that references this table or view in the materialized query; or allow a table that references this table or view in its materialized query to no longer be a materialized query table. <ul style="list-style-type: none"><li>• G = Held and grantable</li><li>• N = Not held</li><li>• Y = Held</li></ul>
DELETEAUTH	CHAR (1)		Privilege to delete rows from a table or updatable view. <ul style="list-style-type: none"><li>• G = Held and grantable</li><li>• N = Not held</li><li>• Y = Held</li></ul>
INDEXAUTH	CHAR (1)		Privilege to create an index on a table. <ul style="list-style-type: none"><li>• G = Held and grantable</li><li>• N = Not held</li><li>• Y = Held</li></ul>

Table 141. SYSCAT.TABAUTH Catalog View (continued)

Column Name	Data Type	Nullable	Description
INSERTAUTH	CHAR (1)		Privilege to insert rows into a table or updatable view, or to run the import utility against a table or view. <ul style="list-style-type: none"> <li>• G = Held and grantable</li> <li>• N = Not held</li> <li>• Y = Held</li> </ul>
REFAUTH	CHAR (1)		Privilege to create and drop a foreign key referencing a table as the parent. <ul style="list-style-type: none"> <li>• G = Held and grantable</li> <li>• N = Not held</li> <li>• Y = Held</li> </ul>
SELECTAUTH	CHAR (1)		Privilege to retrieve rows from a table or view, create views on a table, or to run the export utility against a table or view. <ul style="list-style-type: none"> <li>• G = Held and grantable</li> <li>• N = Not held</li> <li>• Y = Held</li> </ul>
UPDATEAUTH	CHAR (1)		Privilege to run the UPDATE statement against a table or updatable view. <ul style="list-style-type: none"> <li>• G = Held and grantable</li> <li>• N = Not held</li> <li>• Y = Held</li> </ul>

## SYSCAT.TABCONST

Each row represents a table constraint of type CHECK, UNIQUE, PRIMARY KEY, or FOREIGN KEY. For table hierarchies, each constraint is recorded only at the level of the hierarchy where the constraint was created.

Table 142. SYSCAT.TABCONST Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR (128)		Name of the constraint.
TABSCHEMA	VARCHAR (128)		Schema name of the table to which this constraint applies.
TABNAME	VARCHAR (128)		Unqualified name of the table to which this constraint applies.
OWNER	VARCHAR (128)		Authorization ID under which the constraint was created.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• S = The owner is the system</li> <li>• U = The owner is an individual user</li> </ul>
TYPE	CHAR (1)		Indicates the constraint type. <ul style="list-style-type: none"> <li>• F = Foreign key</li> <li>• I = Functional dependency</li> <li>• K = Check</li> <li>• P = Primary key</li> <li>• U = Unique</li> </ul>

## SYSCAT.TABCONST

Table 142. SYSCAT.TABCONST Catalog View (continued)

Column Name	Data Type	Nullable	Description
ENFORCED	CHAR (1)		<ul style="list-style-type: none"><li>• N = Do not enforce constraint</li><li>• Y = Enforce constraint</li></ul>
CHECKEXISTINGDATA	CHAR (1)		<ul style="list-style-type: none"><li>• D = Defer checking any existing data</li><li>• I = Immediately check existing data</li><li>• N = Never check existing data</li></ul>
ENABLEQUERYOPT	CHAR (1)		<ul style="list-style-type: none"><li>• N = Query optimization is disabled</li><li>• Y = Query optimization is enabled</li></ul>
DEFINER <sup>1</sup>	VARCHAR (128)		Authorization ID under which the constraint was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

**Note:**

1. The DEFINER column is included for backwards compatibility. See OWNER.

## SYSCAT.TABDEP

Each row represents a dependency of a view or a materialized query table on some other object. The view or materialized query table depends on the object of type BTYPE of name BNAME, so a change to the object affects the view or materialized query table. Also encodes how privileges on views depend on privileges on underlying tables and views.

Table 143. SYSCAT.TABDEP Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the view or materialized query table.
TABNAME	VARCHAR (128)		Unqualified name of the view or materialized query table.
DTYPE	CHAR (1)		Type of the depending object. <ul style="list-style-type: none"><li>• S = Materialized query table</li><li>• T = Table (staging only)</li><li>• V = View (untyped)</li><li>• W = Typed view</li></ul>
OWNER	VARCHAR (128)		Authorization ID of the creator of the view or materialized query table.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• U = The owner is an individual user</li></ul>

Table 143. SYSCAT.TABDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: <ul style="list-style-type: none"> <li>• A = Alias</li> <li>• F = Routine instance</li> <li>• I = Index, if recording dependency on a base table</li> <li>• N = Nickname</li> <li>• O = Privilege dependency on all subtables or subviews in a table or view hierarchy</li> <li>• R = Structured type</li> <li>• S = Materialized query table</li> <li>• T = Table (untyped)</li> <li>• U = Typed table</li> <li>• V = View (untyped)</li> <li>• W = Typed view</li> <li>• Z = XSR object</li> <li>• v = Global variable</li> </ul>
BSCHEMA	VARCHAR (128)		Schema name of the object on which the view or materialized query table depends.
BNAME	VARCHAR (128)		Unqualified name of the object on which the view or materialized query table depends.
TABAUTH	SMALLINT	Y	If BTYPE is 'N', 'O', 'S', 'T', 'U', 'V', or 'W', encodes the privileges on the underlying table or view on which this view or materialized query table depends; null value otherwise.
DEFINER <sup>1</sup>	VARCHAR (128)		Authorization ID of the creator of the view or materialized query table.

**Note:**

1. The DEFINER column is included for backwards compatibility. See OWNER.

## SYSCAT.TABDETACHEDDEP

Each row represents a detached dependency between a detached dependent table and a detached table.

Table 144. SYSCAT.TABDETACHEDDEP Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the detached table.
TABNAME	VARCHAR (128)		Unqualified name of the detached table.
DEPTABSCHEMA	VARCHAR (128)		Schema name of the detached dependent table.
DEPTABNAME	VARCHAR (128)		Unqualified name of the detached dependent table.

---

**SYSCAT.TABLES**

Each row represents a table, view, alias, or nickname. Each table or view hierarchy has one additional row representing the hierarchy table or hierarchy view that implements the hierarchy. Catalog tables and views are included.

Table 145. SYSCAT.TABLES Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the object.
TABNAME	VARCHAR (128)		Unqualified name of the object.
OWNER	VARCHAR (128)		Authorization ID under which the table, view, alias, or nickname was created.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• S = The owner is the system</li> <li>• U = The owner is an individual user</li> </ul>
TYPE	CHAR (1)		Type of object. <ul style="list-style-type: none"> <li>• A = Alias</li> <li>• G = Global temporary table</li> <li>• H = Hierarchy table</li> <li>• L = Detached table</li> <li>• N = Nickname</li> <li>• S = Materialized query table</li> <li>• T = Table (untyped)</li> <li>• U = Typed table</li> <li>• V = View (untyped)</li> <li>• W = Typed view</li> </ul>
STATUS	CHAR (1)		Status of the object. <ul style="list-style-type: none"> <li>• C = Set integrity pending</li> <li>• N = Normal</li> <li>• X = Inoperative</li> </ul>
BASE_TABSCHEMA	VARCHAR (128)	Y	If TYPE = 'A', contains the schema name of the table, view, alias, or nickname that is referenced by this alias; null value otherwise.
BASE_TABNAME	VARCHAR (128)	Y	If TYPE = 'A', contains the unqualified name of the table, view, alias, or nickname that is referenced by this alias; null value otherwise.
ROWTYPESCHEMA	VARCHAR (128)	Y	Schema name of the row type for this table, if applicable; null value otherwise.
ROWTYPE_NAME	VARCHAR (128)	Y	Unqualified name of the row type for this table, if applicable; null value otherwise.
CREATE_TIME	TIMESTAMP		Time at which the object was created.
ALTER_TIME	TIMESTAMP		Time at which the object was last altered.
INVALIDATE_TIME	TIMESTAMP		Time at which the object was last invalidated.
STATS_TIME	TIMESTAMP	Y	Time at which any change was last made to recorded statistics for this object. Null if statistics are not collected.
COLCOUNT	SMALLINT		Number of columns, including inherited columns (if any).

Table 145. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
TABLEID	SMALLINT		Internal logical object identifier.
TBSPACEID	SMALLINT		Internal logical identifier for the primary table space for this object.
CARD	BIGINT		Total number of rows; -1 if statistics are not collected.
NPAGES	BIGINT		Total number of pages on which the rows of the table exist; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.
FPAGES	BIGINT		Total number of pages; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.
OVERFLOW	BIGINT		Total number of overflow records in the table; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.
TBSPACE	VARCHAR (128)	Y	Name of the primary table space for the table. If no other table space is specified, all parts of the table are stored in this table space. Null for aliases, views, and partitioned tables.
INDEX_TBSPACE	VARCHAR (128)	Y	Name of the table space that holds all indexes created on this table. Null for aliases, views, and partitioned tables, or if the INDEX IN clause was omitted or specified with the same value as the IN clause of the CREATE TABLE statement.
LONG_TBSPACE	VARCHAR (128)	Y	Name of the table space that holds all long data (LONG or LOB column types) for this table. Null for aliases, views, and partitioned tables, or if the LONG IN clause was omitted or specified with the same value as the IN clause of the CREATE TABLE statement.
PARENTS	SMALLINT	Y	Number of parent tables for this object; that is, the number of referential constraints in which this object is a dependent.
CHILDREN	SMALLINT	Y	Number of dependent tables for this object; that is, the number of referential constraints in which this object is a parent.
SELFREFS	SMALLINT	Y	Number of self-referencing referential constraints for this object; that is, the number of referential constraints in which this object is both a parent and a dependent.
KEYCOLUMNS	SMALLINT	Y	Number of columns in the primary key.
KEYINDEXID	SMALLINT	Y	Index identifier for the primary key index; 0 or the null value if there is no primary key.
KEYUNIQUE	SMALLINT		Number of unique key constraints (other than the primary key constraint) defined on this object.

## SYSCAT.TABLES

Table 145. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
CHECKCOUNT	SMALLINT		Number of check constraints defined on this object.
DATA_CAPTURE	CHAR (1)		<ul style="list-style-type: none"> <li>L = Table participates in data replication, including replication of LONG VARCHAR and LONG VARGRAPHIC columns</li> <li>N = Table does not participate in data replication</li> <li>Y = Table participates in data replication, excluding replication of LONG VARCHAR and LONG VARGRAPHIC columns</li> </ul>
CONST_CHECKED	CHAR (32)		<ul style="list-style-type: none"> <li>Byte 1 represents foreign key constraint.</li> <li>Byte 2 represents check constraint.</li> <li>Byte 5 represents materialized query table.</li> <li>Byte 6 represents generated column.</li> <li>Byte 7 represents staging table.</li> <li>Byte 8 represents data partitioning constraint.</li> <li>Other bytes are reserved for future use.</li> </ul> <p>Possible values are:</p> <ul style="list-style-type: none"> <li>F = In byte 5, the materialized query table cannot be refreshed incrementally. In byte 7, the content of the staging table is incomplete and cannot be used for incremental refresh of the associated materialized query table.</li> <li>N = Not checked</li> <li>U = Checked by user</li> <li>W = Was in 'U' state when the table was placed in set integrity pending state</li> <li>Y = Checked by system</li> </ul>
PMAP_ID	SMALLINT	Y	Identifier for the distribution map that is currently in use by this table (null for aliases or views).
PARTITION_MODE	CHAR (1)		<p>Indicates how data is distributed among database partitions in a partitioned database system.</p> <ul style="list-style-type: none"> <li>H = Hashing</li> <li>R = Replicated across database partitions</li> <li>Blank = No database partitioning</li> </ul>
LOG_ATTRIBUTE	CHAR (1)		<ul style="list-style-type: none"> <li>Always 0. This column is no longer used.</li> </ul>
PCTFREE	SMALLINT		Percentage of each page to be reserved for future inserts.
APPEND_MODE	CHAR (1)		<p>Controls how rows are inserted into pages.</p> <ul style="list-style-type: none"> <li>N = New rows are inserted into existing spaces, if available</li> <li>Y = New rows are appended to the end of the data</li> </ul>



Table 145. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
REFRESH	CHAR (1)		Refresh mode. <ul style="list-style-type: none"> <li>• D = Deferred</li> <li>• I = Immediate</li> <li>• O = Once</li> <li>• Blank = Not a materialized query table</li> </ul>
REFRESH_TIME	TIMESTAMP	Y	For REFRESH = 'D' or 'O', time at which the data was last refreshed (REFRESH TABLE statement); null value otherwise.
LOCKSIZE	CHAR (1)		Indicates the preferred lock granularity for tables that are accessed by data manipulation language (DML) statements. Applies to tables only. Possible values are: <ul style="list-style-type: none"> <li>• I = Block insert</li> <li>• R = Row</li> <li>• T = Table</li> <li>• Blank = Not applicable</li> </ul>
VOLATILE	CHAR (1)		<ul style="list-style-type: none"> <li>• C = Cardinality of the table is volatile</li> <li>• Blank = Not applicable</li> </ul>
ROW_FORMAT	CHAR (1)		Not used.
PROPERTY	VARCHAR (32)		Properties for a table. A single blank indicates that the table has no properties. The following is position within string, value, and meaning: <ul style="list-style-type: none"> <li>• 1, Y = User maintained materialized query table</li> <li>• 2, Y = Staging table</li> <li>• 3, Y = Propagate immediate</li> <li>• 11, Y = Nickname that will not be cached</li> </ul>
STATISTICS_PROFILE	CLOB (10M)	Y	RUNSTATS command used to register a statistical profile for the object.
COMPRESSION	CHAR (1)		<ul style="list-style-type: none"> <li>• B = Both value and row compression are activated</li> <li>• N = No compression is activated; a row format that does not support compression is used</li> <li>• R = Row compression is activated; a row format that supports compression might be used</li> <li>• V = Value compression is activated; a row format that supports compression is used</li> <li>• Blank = Not applicable</li> </ul>

## SYSCAT.TABLES

Table 145. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
ACCESS_MODE	CHAR (1)		Access restriction state of the object. These states only apply to objects that are in set integrity pending state or to objects that were processed by a SET INTEGRITY statement. Possible values are: <ul style="list-style-type: none"> <li>• D = No data movement</li> <li>• F = Full access</li> <li>• N = No access</li> <li>• R = Read-only access</li> </ul>
CLUSTERED	CHAR (1)	Y	<ul style="list-style-type: none"> <li>• Y = Table is multidimensionally clustered (even if only by one dimension)</li> <li>• Null value = Table is not multidimensionally clustered</li> </ul>
ACTIVE_BLOCKS	BIGINT		Total number of active blocks in the table, or -1. Applies to multidimensional clustering (MDC) tables only.
DROPRULE	CHAR (1)		<ul style="list-style-type: none"> <li>• N = No rule</li> <li>• R = Restrict rule applies on drop</li> </ul>
MAXFREESPACESEARCH	SMALLINT		Reserved for future use.
AVGCOMPRESSEDROWSIZE	SMALLINT		Average length (in bytes) of compressed rows in this table; -1 if statistics are not collected.
AVGROWCOMPRESSIONRATIO	REAL		For compressed rows in the table, this is the average compression ratio by row; that is, the average uncompressed row length divided by the average compressed row length; -1 if statistics are not collected.
AVGROWSIZE	SMALLINT		Average length (in bytes) of both compressed and uncompressed rows in this table; -1 if statistics are not collected.
PCTROWSCOMPRESSED	REAL		Compressed rows as a percentage of the total number of rows in the table; -1 if statistics are not collected.
LOGINDEXBUILD	VARCHAR (3)	Y	Level of logging that is to be performed during create, recreate, or reorganize index operations on the table. <ul style="list-style-type: none"> <li>• OFF = Index build operations on the table will be logged minimally</li> <li>• ON = Index build operations on the table will be logged completely</li> <li>• Null value = Value of the <i>logindexbuild</i> database configuration parameter will be used to determine whether or not index build operations are to be completely logged</li> </ul>
CODEPAGE	SMALLINT		Code page of the object. This is the default code page used for all character columns, triggers, check constraints, and expression-generated columns.
COLLATIONSCHEMA	VARCHAR (128)		Schema name of the collation for the table.

Table 145. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLLATIONNAME	VARCHAR (128)		Unqualified name of the collation for the table.
COLLATIONSCHEMA_ORDERBY	VARCHAR (128)		Schema name of the collation for ORDER BY clauses in the table.
COLLATIONNAME_ORDERBY	VARCHAR (128)		Unqualified name of the collation for ORDER BY clauses in the table.
ENCODING_SCHEME	CHAR (1)		<ul style="list-style-type: none"> <li>• A = CCSID ASCII was specified</li> <li>• U = CCSID UNICODE was specified</li> <li>• Blank = CCSID clause was not specified</li> </ul>
PCTPAGESSAVED	SMALLINT		Approximate percentage of pages saved in the table as a result of row compression. This value includes overhead bytes for each user data row in the table, but does not include the space that is consumed by dictionary overhead; -1 if statistics are not collected.
LAST_REGEN_TIME	TIMESTAMP	Y	Time at which any views or check constraints on the table were last regenerated.
SECPOLICYID	INTEGER		Identifier for the security policy protecting the table; 0 for non-protected tables.
PROTECTIONGRANULARITY	CHAR (1)		<ul style="list-style-type: none"> <li>• B = Both column- and row-level granularity</li> <li>• C = Column-level granularity</li> <li>• R = Row-level granularity</li> <li>• Blank = Non-protected table</li> </ul>
AUDITPOLICYID	INTEGER	Y	Identifier for the audit policy.
AUDITPOLICYNAME	VARCHAR (128)	Y	Name of the audit policy.
DEFINER <sup>1</sup>	VARCHAR (128)		Authorization ID under which the table, view, alias, or nickname was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

**Note:**

1. The DEFINER column is included for backwards compatibility. See OWNER.

## SYSCAT.TABLESPACES

Each row represents a table space.

Table 146. SYSCAT.TABLESPACES Catalog View

Column Name	Data Type	Nullable	Description
TBSPACE	VARCHAR (128)		Name of the table space.
OWNER	VARCHAR (128)		Authorization ID under which the table space was created.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• S = The owner is the system</li> <li>• U = The owner is an individual user</li> </ul>
CREATE_TIME	TIMESTAMP		Time at which the table space was created.

## SYSCAT.TABLESPACES

Table 146. SYSCAT.TABLESPACES Catalog View (continued)

Column Name	Data Type	Nullable	Description
TBSPACEID	INTEGER		Identifier for the table space.
TBSPACETYPE	CHAR (1)		Type of table space. <ul style="list-style-type: none"> <li>• D = Database-managed space</li> <li>• S = System-managed space</li> </ul>
DATATYPE	CHAR (1)		Type of data that can be stored in this table space. <ul style="list-style-type: none"> <li>• A = All types of permanent data; regular table space</li> <li>• L = All types of permanent data; large table space</li> <li>• T = System temporary tables only</li> <li>• U = Declared temporary tables only</li> </ul>
EXTENTSIZE	INTEGER		Size of each extent, in pages of size PAGESIZE. This many pages are written to one container in the table space before switching to the next container.
PREFETCHSIZE	INTEGER		Number of pages of size PAGESIZE to be read when prefetching is performed; -1 when AUTOMATIC.
OVERHEAD	DOUBLE		Controller overhead and disk seek and latency time, in milliseconds (average for the containers in this table space).
TRANSFERRATE	DOUBLE		Time to read one page of size PAGESIZE into the buffer (average for the containers in this table space).
PAGESIZE	INTEGER		Size (in bytes) of pages in this table space.
DBPGNAME	VARCHAR (128)		Name of the database partition group that is associated with this table space.
BUFFERPOOLID	INTEGER		Identifier for the buffer pool that is used by this table space (1 indicates the default buffer pool).
DROP_RECOVERY	CHAR (1)		Indicates whether or not tables in this table space can be recovered after a drop table operation. <ul style="list-style-type: none"> <li>• N = Tables are not recoverable</li> <li>• Y = Tables are recoverable</li> </ul>
NGNAME <sup>1</sup>	VARCHAR (128)		Name of the database partition group that is associated with this table space.
DEFINER <sup>2</sup>	VARCHAR (128)		Authorization ID under which the table space was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

**Note:**

1. The NGNAME column is included for backwards compatibility. See DBPGNAME.
2. The DEFINER column is included for backwards compatibility. See OWNER.

---

## SYSCAT.TABOPTIONS

Each row represents an option that is associated with a remote table.

Table 147. SYSCAT.TABOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of a table, view, alias, or nickname.
TABNAME	VARCHAR (128)		Unqualified name of a table, view, alias, or nickname.
OPTION	VARCHAR (128)		Name of the table option.
SETTING	CLOB (32K)		Value of the table option.

---

## SYSCAT.TBSPACEAUTH

Each row represents a user or group that has been granted the USE privilege on a particular table space in the database.

Table 148. SYSCAT.TBSPACEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>S = Grantor is the system</li> <li>U = Grantor is an individual user</li> </ul>
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		<ul style="list-style-type: none"> <li>G = Grantee is a group</li> <li>R = Grantee is a role</li> <li>U = Grantee is an individual user</li> </ul>
TBSPACE	VARCHAR (128)		Name of the table space.
USEAUTH	CHAR (1)		Privilege to create tables within the table space. <ul style="list-style-type: none"> <li>G = Held and grantable</li> <li>N = Not held</li> <li>Y = Held</li> </ul>

---

## SYSCAT.THRESHOLDS

Each row represents a threshold.

Table 149. SYSCAT.THRESHOLDS Catalog View

Column Name	Data Type	Nullable	Description
THRESHOLDNAME	VARCHAR (128)		Name of the threshold.
THRESHOLDID	INTEGER		Identifier for the threshold.
ORIGIN	CHAR (1)		Origin of the threshold. <ul style="list-style-type: none"> <li>U = Threshold was created by a user</li> <li>W = Threshold was created through a work action set</li> </ul>

## SYSCAT.THRESHOLDS

Table 149. SYSCAT.THRESHOLDS Catalog View (continued)

Column Name	Data Type	Nullable	Description
THRESHOLDCLASS	CHAR (1)		Classification of the threshold. <ul style="list-style-type: none"> <li>• A = Aggregate threshold</li> <li>• C = Activity threshold</li> </ul>
THRESHOLDPREDICATE	VARCHAR (15)		Type of the threshold. Possible values are: <ul style="list-style-type: none"> <li>• CONCDBC</li> <li>• CONCWCN</li> <li>• CONCWOC</li> <li>• CONNIDLETIME</li> <li>• DBCONN</li> <li>• ESTSQLCOST</li> <li>• ROWSRET</li> <li>• SCCONN</li> <li>• TEMPSPACE</li> <li>• TOTALTIME</li> </ul>
THRESHOLDPREDICATEID	SMALLINT		Identifier for the threshold predicate.
DOMAIN	CHAR (2)		Domain of the threshold. <ul style="list-style-type: none"> <li>• DB = Database</li> <li>• SB = Service subclass</li> <li>• SP = Service superclass</li> <li>• WA = Work action set</li> <li>• WD = Workload definition</li> </ul>
DOMAINID	INTEGER		Identifier for the object with which the threshold is associated. This can be a service class, work action or workload unique ID. If this is a database threshold, this value is 0.
ENFORCEMENT	CHAR (1)		Scope of enforcement for the threshold. <ul style="list-style-type: none"> <li>• D = Database</li> <li>• P = Database partition</li> <li>• W = Workload occurrence</li> </ul>
QUEUEING	CHAR (1)		<ul style="list-style-type: none"> <li>• N = The threshold is not queueing</li> <li>• Y = The threshold is queueing</li> </ul>
MAXVALUE	BIGINT		Upper bound specified by the threshold.
QUEUESIZE	INTEGER		If QUEUEING is 'Y', the size of the queue. -1 otherwise.
COLLECTACTDATA	CHAR (1)		Specifies what activity data should be collected by the applicable event monitor. <ul style="list-style-type: none"> <li>• D = Activity data with details</li> <li>• N = None</li> <li>• V = Activity data with details and values</li> <li>• W = Activity data without details</li> </ul>
COLLECTACTPARTITION	CHAR (1)		Specifies where activity data is collected. <ul style="list-style-type: none"> <li>• C = Database partition of the coordinator of the activity</li> <li>• D = All database partitions</li> </ul>

Table 149. SYSCAT.THRESHOLDS Catalog View (continued)

Column Name	Data Type	Nullable	Description
EXECUTION	CHAR (1)		Indicates whether or not execution continues after the threshold has been exceeded. <ul style="list-style-type: none"> <li>• C = Execution continues</li> <li>• S = Execution stops</li> </ul>
ENABLED	CHAR (1)		<ul style="list-style-type: none"> <li>• N = This threshold is disabled.</li> <li>• Y = This threshold is enabled.</li> </ul>
CREATE_TIME	TIMESTAMP		Time at which the threshold was created.
ALTER_TIME	TIMESTAMP		Time at which the threshold was last altered.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSCAT.TRANSFORMS

Each row represents the functions that handle transformations between a user-defined type and a base SQL type, or the reverse.

Table 150. SYSCAT.TRANSFORMS Catalog View

Column Name	Data Type	Nullable	Description
TYPEID	SMALLINT		Identifier for the data type.
TYPESHEMA	VARCHAR (128)		Schema name of the data type. The schema name for built-in types is 'SYSIBM'.
TYPENAME	VARCHAR (128)		Unqualified name of the data type.
GROUPNAME	VARCHAR (128)		Name of the transform group.
FUNCID	INTEGER		Identifier for the routine.
FUNCHEMA	VARCHAR (128)		Schema name of the routine.
FUNCNAME	VARCHAR (128)		Unqualified name of the routine.
SPECIFICNAME	VARCHAR (128)		Name of the routine instance (might be system-generated).
TRANSFORMTYPE	VARCHAR(8)		<ul style="list-style-type: none"> <li>• 'FROM SQL' = Transform function transforms a structured type from SQL</li> <li>• 'TO SQL' = Transform function transforms a structured type to SQL</li> </ul>
FORMAT	CHAR (1)		Format produced by the FROM SQL transform. <ul style="list-style-type: none"> <li>• S = Structured data type</li> <li>• U = User-defined</li> </ul>
MAXLENGTH	INTEGER	Y	Maximum length (in bytes) of output from the FROM SQL transform; null value for TO SQL transforms.
ORIGIN	CHAR (1)		Source of this group of transforms. <ul style="list-style-type: none"> <li>• O = Original transform group (built-in or system-defined)</li> <li>• R = Redefined transform group (only built-in groups can be redefined)</li> </ul>
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSCAT.TRIGDEP

---

## SYSCAT.TRIGDEP

Each row represents a dependency of a trigger on some other object. The trigger depends on the object of type BTYPE of name BNAME, so a change to the object affects the trigger.

Table 151. SYSCAT.TRIGDEP Catalog View

Column Name	Data Type	Nullable	Description
TRIGSCHEMA	VARCHAR (128)		Schema name of the trigger.
TRIGNAME	VARCHAR (128)		Unqualified name of the trigger.
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: <ul style="list-style-type: none"><li>• A = Alias</li><li>• B = Trigger</li><li>• F = Routine instance</li><li>• H = Hierachy table</li><li>• K = Package</li><li>• L = Detached table</li><li>• O = Privilege dependency on all subtables or subviews in a table or view hierarchy</li><li>• Q = Sequence</li><li>• R = Structured type</li><li>• S = Materialized query table</li><li>• T = Table (not typed)</li><li>• U = Typed table</li><li>• V = View (not typed)</li><li>• W = Typed view</li><li>• X = Index extension</li><li>• Z = XSR object</li><li>• v = Global variable</li></ul>
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.
TABAUTH	SMALLINT	Y	If BTYPE = 'O', 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by a dependent trigger; null value otherwise.

---

## SYSCAT.TRIGGERS

Each row represents a trigger. For table hierarchies, each trigger is recorded only at the level of the hierarchy where the trigger was created.

Table 152. SYSCAT.TRIGGERS Catalog View

Column Name	Data Type	Nullable	Description
TRIGSCHEMA	VARCHAR (128)		Schema name of the trigger.
TRIGNAME	VARCHAR (128)		Unqualified name of the trigger.



Table 152. SYSCAT.TRIGGERS Catalog View (continued)

Column Name	Data Type	Nullable	Description
OWNER	VARCHAR (128)		Authorization ID under which the trigger was created.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• S = The owner is the system</li> <li>• U = The owner is an individual user</li> </ul>
TABSHEMA	VARCHAR (128)		Schema name of the table or view to which this trigger applies.
TABNAME	VARCHAR (128)		Unqualified name of the table or view to which this trigger applies.
TRIGTIME	CHAR (1)		<p>Time at which triggered actions are applied to the base table, relative to the event that fired the trigger.</p> <ul style="list-style-type: none"> <li>• A = Trigger is applied after the event</li> <li>• B = Trigger is applied before the event</li> <li>• I = Trigger is applied instead of the event</li> </ul>
TRIGEVENT	CHAR (1)		<p>Event that fires the trigger.</p> <ul style="list-style-type: none"> <li>• D = Delete operation</li> <li>• I = Insert operation</li> <li>• U = Update operation</li> </ul>
GRANULARITY	CHAR (1)		<p>Trigger is executed once per:</p> <ul style="list-style-type: none"> <li>• R = Row</li> <li>• S = Statement</li> </ul>
VALID	CHAR (1)		<ul style="list-style-type: none"> <li>• N = Trigger is invalid</li> <li>• Y = Trigger is valid</li> <li>• X = Trigger is inoperative and must be recreated</li> </ul>
CREATE_TIME	TIMESTAMP		Time at which the trigger was defined. Used in resolving functions and types.
QUALIFIER	VARCHAR (128)		Value of the default schema at the time of object definition. Used to complete any unqualified references.
FUNC_PATH	CLOB (2K)		SQL path at the time the trigger was defined. Used in resolving functions and types.
TEXT	CLOB (2M)		Full text of the CREATE TRIGGER statement, exactly as typed.
LAST_REGEN_TIME	TIMESTAMP		Time at which the packed descriptor for the trigger was last regenerated.
COLLATIONSCHEMA	VARCHAR (128)		Schema name of the collation for the trigger.
COLLATIONNAME	VARCHAR (128)		Unqualified name of the collation for the trigger.
COLLATIONSCHEMA_ORDERBY	VARCHAR (128)		Schema name of the collation for ORDER BY clauses in the trigger.
COLLATIONNAME_ORDERBY	VARCHAR (128)		Unqualified name of the collation for ORDER BY clauses in the trigger.
DEFINER <sup>1</sup>	VARCHAR (128)		Authorization ID under which the trigger was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSCAT.TRIGGERS

Table 152. SYSCAT.TRIGGERS Catalog View (continued)

Column Name	Data Type	Nullable	Description
-------------	-----------	----------	-------------

**Note:**

1. The DEFINER column is included for backwards compatibility. See OWNER.

## SYSCAT.TYPEMAPPINGS

Each row represents a data type mapping between a locally-defined data type and a data source data type. There are two mapping types (mapping directions):

- Forward type mappings map a data source data type to a locally-defined data type.
- Reverse type mappings map a locally-defined data type to a data source data type.

Table 153. SYSCAT.TYPEMAPPINGS Catalog View

Column Name	Data Type	Nullable	Description
TYPE_MAPPING	VARCHAR (18)		Name of the type mapping (might be system-generated).
MAPPINGDIRECTION	CHAR (1)		Indicates whether this type mapping is a forward or a reverse type mapping. <ul style="list-style-type: none"><li>• F = Forward type mapping</li><li>• R = Reverse type mapping</li></ul>
TYPESHEMA	VARCHAR (128)	Y	Schema name of the local type in a data type mapping; null for built-in types.
TYPENAME	VARCHAR (128)		Unqualified name of the local type in a data type mapping.
TYPEID	SMALLINT		Identifier for the data type.
SOURCETYPEID	SMALLINT		Identifier for the source type.
OWNER	VARCHAR (128)		Authorization ID under which this type mapping was created. 'SYSIBM' indicates a built-in type mapping.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• S = The owner is the system</li><li>• U = The owner is an individual user</li></ul>
LENGTH	INTEGER	Y	Maximum length or precision of the local data type in this mapping. If null, the system determines the maximum length or precision. For character types, represents the maximum number of bytes.
SCALE	SMALLINT	Y	Maximum number of digits in the fractional part of a local decimal value in this mapping. If null, the system determines the maximum number.
LOWER_LEN	INTEGER	Y	Minimum length or precision of the local data type in this mapping. If null, the system determines the minimum length or precision. For character types, represents the minimum number of bytes.

Table 153. SYSCAT.TYEMAPPINGS Catalog View (continued)

Column Name	Data Type	Nullable	Description
UPPER_LEN	INTEGER	Y	Maximum length or precision of the local data type in this mapping. If null, the system determines the maximum length or precision. For character types, represents the maximum number of bytes.
LOWER_SCALE	SMALLINT	Y	Minimum number of digits in the fractional part of a local decimal value in this mapping. If null, the system determines the minimum number.
UPPER_SCALE	SMALLINT	Y	Maximum number of digits in the fractional part of a local decimal value in this mapping. If null, the system determines the maximum number.
S_OPR_P	CHAR (2)	Y	Relationship between the scale and precision of a local decimal value in this mapping. Basic comparison operators (=, <, >, <=, >=, <>) can be used. A null value indicates that no specific relationship is required.
BIT_DATA	CHAR (1)	Y	Indicates whether or not this character type is for bit data. Possible values are: <ul style="list-style-type: none"> <li>• N = This type is not for bit data</li> <li>• Y = This type is for bit data</li> <li>• Null value = This is not a character data type, or the system determines the bit data attribute</li> </ul>
WRAPNAME	VARCHAR (128)	Y	Data access protocol (wrapper) to which this mapping applies.
SERVERNAME	VARCHAR (128)	Y	Uppercase name of the server.
SERVERTYPE	VARCHAR (30)	Y	Type of server.
SERVERVERSION	VARCHAR (18)	Y	Server version.
REMOTE_TYPESHEMA	VARCHAR (128)	Y	Schema name of the data source data type.
REMOTE_TYPENAME	VARCHAR (128)		Unqualified name of the data source data type.
REMOTE_META_TYPE	CHAR (1)	Y	Indicates whether this remote type is a system built-in type or a distinct type. <ul style="list-style-type: none"> <li>• S = System built-in type</li> <li>• T = Distinct type</li> </ul>
REMOTE_LOWER_LEN	INTEGER	Y	Minimum length or precision of the remote data type in this mapping, or the null value. For character types, represents the minimum number of characters (not bytes). For binary types, represents the minimum number of bytes. A value of -1 indicates that the default length or precision is used, or that the remote type does not have a length or precision.

## SYSCAT.TYEMAPPINGS

Table 153. SYSCAT.TYEMAPPINGS Catalog View (continued)

Column Name	Data Type	Nullable	Description
REMOTE_UPPER_LEN	INTEGER	Y	Maximum length or precision of the remote data type in this mapping, or the null value. For character types, represents the maximum number of characters (not bytes). For binary types, represents the maximum number of bytes. A value of -1 indicates that the default length or precision is used, or that the remote type does not have a length or precision.
REMOTE_LOWER_SCALE	SMALLINT	Y	Minimum number of digits in the fractional part of a remote decimal value in this mapping, or the null value.
REMOTE_UPPER_SCALE	SMALLINT	Y	Maximum number of digits in the fractional part of a remote decimal value in this mapping, or the null value.
REMOTE_S_OPR_P	CHAR (2)	Y	Relationship between the scale and precision of a remote decimal value in this mapping. Basic comparison operators (=, <, >, <=, >=, <>) can be used. A null value indicates that no specific relationship is required.
REMOTE_BIT_DATA	CHAR (1)	Y	Indicates whether or not this remote character type is for bit data. Possible values are: <ul style="list-style-type: none"><li>• N = This type is not for bit data</li><li>• Y = This type is for bit data</li><li>• Null value = This is not a character data type, or the system determines the bit data attribute</li></ul>
USER_DEFINED	CHAR (1)		Indicates whether or not the mapping is user-defined. The value is always 'Y'; that is, the mapping is always user-defined.
CREATE_TIME	TIMESTAMP		Time at which this mapping was created.
DEFINER <sup>1</sup>	VARCHAR (128)		Authorization ID under which this type mapping was created. 'SYSIBM' indicates a built-in type mapping.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

**Note:**

1. The DEFINER column is included for backwards compatibility. See OWNER.

## SYSCAT.USEROPTIONS

Each row represents a server-specific user option value.

Table 154. SYSCAT.USEROPTIONS Catalog View

Column Name	Data Type	Nullable	Description
AUTHID	VARCHAR (128)		Local authorization ID, in uppercase characters.
AUTHIDTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• U = Grantee is an individual user</li></ul>

Table 154. SYSCAT.USEROPTIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
SERVERNAME	VARCHAR (128)		Name of the server on which the user is defined.
OPTION	VARCHAR (128)		Name of the user option.
SETTING	VARCHAR (2048)		Value of the user option.

## SYSCAT.VARIABLEAUTH

Each row represents a user, group or role that has been granted one or more privileges by a specific grantor on a global variable in the database.

Table 155. SYSCAT.VARIABLEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• S = Grantor is the system</li> <li>• U = Grantor is an individual user</li> </ul>
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		<ul style="list-style-type: none"> <li>• G = Grantee is a group</li> <li>• R = Grantee is a role</li> <li>• U = Grantee is an individual user</li> </ul>
VARSHEMA	VARCHAR (128)		Schema name of the global variable.
VARNAME	VARCHAR (128)		Unqualified name of the global variable.
VARID	INTEGER		Identifier for the global variable.
READAUTH	CHAR (1)		Privilege to read the global variable. <ul style="list-style-type: none"> <li>• G = Held and grantable</li> <li>• N = Not held</li> <li>• Y = Held</li> </ul>
WRITEAUTH	CHAR (1)		Privilege to write the global variable. <ul style="list-style-type: none"> <li>• G = Held and grantable</li> <li>• N = Not held</li> <li>• Y = Held</li> </ul>

## SYSCAT.VARIABLEDEP

Each row represents a dependency of a global variable on some other object. The global variable depends on the object of type BTYPE of name BNAME, so a change to the object affects the global variable.

Table 156. SYSCAT.VARIABLEDEP Catalog View

Column Name	Data Type	Nullable	Description
VARSHEMA	VARCHAR (128)		Schema name of the global variable that has dependencies on another object.
VARNAME	VARCHAR (128)		Unqualified name of the global variable that has dependencies on another object.

## SYSCAT.VARIABLEDEP

Table 156. SYSCAT.VARIABLEDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: <ul style="list-style-type: none"><li>• A = Alias</li><li>• F = Routine instance</li><li>• H = Hierarchy table</li><li>• N = Nickname</li><li>• O = Privilege dependency on all subtables or subviews in a table or view hierarchy</li><li>• S = Materialized query table</li><li>• T = Table (not typed)</li><li>• U = Typed table</li><li>• V = View (not typed)</li><li>• W = Typed view</li><li>• v = Global variable</li></ul>
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.
TABAUTH	SMALLINT	Y	If BTYPE = 'O', 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by the dependent global variable; null value otherwise.

## SYSCAT.VARIABLES

Each row represents a global variable.

Table 157. SYSCAT.VARIABLES Catalog View

Column Name	Data Type	Nullable	Description
VARSHEMA	VARCHAR (128)		Schema name of the global variable.
VARNAME	VARCHAR (128)		Unqualified name of the global variable.
VARID	INTEGER		Identifier for the global variable.
OWNER	VARCHAR (128)		Authorization ID of the owner of the global variable.
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• U = The owner is an individual user</li></ul>
CREATE_TIME	TIMESTAMP		Time at which the global variable was created.
LAST_REGEN_TIME	TIMESTAMP		Time at which the default expression was last regenerated.
TYPESHEMA	VARCHAR (128)		Schema name of the data type. The schema name for built-in types is 'SYSIBM'.
TYPENAME	VARCHAR (128)		Unqualified name of the data type.
LENGTH	INTEGER		Maximum length of the global variable.
SCALE	SMALLINT		Scale of the global variable.

Table 157. SYSCAT.VARIABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
CODEPAGE	SMALLINT		Code page of the global variable.
COLLATIONSCHEMA	VARCHAR (128)		Schema name of the collation for the variable.
COLLATIONNAME	VARCHAR (128)		Unqualified name of the collation for the variable.
COLLATIONSCHEMA_ORDERBY	VARCHAR (128)		Schema name of the collation for ORDER BY clauses in the variable.
COLLATIONNAME_ORDERBY	VARCHAR (128)		Unqualified name of the collation for ORDER BY clauses in the variable.
SCOPE	CHAR (1)		Scope of the global variable. • S = Session
DEFAULT	CLOB (64K)	Y	Expression used to calculate the initial value of the global variable when first referenced.
QUALIFIER	VARCHAR (128)	Y	Value of the default schema at the time the variable was defined.
FUNC_PATH	CLOB (2K)	Y	SQL path at the time the variable was defined.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSCAT.VIEWS

Each row represents a view.

Table 158. SYSCAT.VIEWS Catalog View

Column Name	Data Type	Nullable	Description
VIEWSCHEMA	VARCHAR (128)		Schema name of the view.
VIEWNAME	VARCHAR (128)		Unqualified name of the view.
OWNER	VARCHAR (128)		Authorization ID under which the view was created.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
SEQNO	SMALLINT		Always 1.
VIEWCHECK	CHAR (1)		Type of view checking. • C = Cascaded check option • L = Local check option • N = No check option
READONLY	CHAR (1)		• N = View can be updated by users with appropriate authorization • Y = View is read-only because of its definition
VALID	CHAR (1)		• X = View or materialized query table definition is inoperative and must be recreated • Y = View or materialized query table definition is valid

## SYSCAT.VIEWS

Table 158. SYSCAT.VIEWS Catalog View (continued)

Column Name	Data Type	Nullable	Description
QUALIFIER	VARCHAR (128)		Value of the default schema at the time of object definition. Used to complete any unqualified references.
FUNC_PATH	CLOB (2K)		SQL path in effect when the view was defined. When the view is referenced in data manipulation language (DML) statements, this path must be used to resolve function calls in the view. 'SYSIBM' for pre-Version 2 views.
TEXT	CLOB (2M)		Full text of the CREATE VIEW statement, exactly as typed.
DEFINER <sup>1</sup>	VARCHAR (128)		Authorization ID under which the view was created.

**Note:**

1. The DEFINER column is included for backwards compatibility. See OWNER.

## SYSCAT.WORKACTIONS

Each row represents a work action that is defined for a work action set.

Table 159. SYSCAT.WORKACTIONS Catalog View

Column Name	Data Type	Nullable	Description
ACTIONNAME	VARCHAR (128)		Name of the work action.
ACTIONID	INTEGER		Identifier for the work action.
ACTIONSETNAME	VARCHAR (128)	Y	Name of the work action set.
ACTIONSETID	INTEGER		Identifier of the work action set to which this work action belongs. This column refers to the ACTIONSETID column in the SYSCAT.WORKACTIONSETS view.
WORKCLASSNAME	VARCHAR (128)	Y	Name of the work class.
WORKCLASSID	INTEGER		Identifier of the work class. This column refers to the WORKCLASSID column in the SYSCAT.WORKCLASSES view.
CREATE_TIME	TIMESTAMP		Time at which the work action was created.
ALTER_TIME	TIMESTAMP		Time at which the work action was last altered.
ENABLED	CHAR (1)		<ul style="list-style-type: none"><li>• N = This work action is disabled.</li><li>• Y = This work action is enabled.</li></ul>



Table 159. SYSCAT.WORKACTIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
ACTIONTYPE	CHAR (1)		<p>The action type that will be performed on each DB2 activity that matches the work class attributes specified in the work class under the matching scope. For this column description, OBJECTTYPE refers to column OBJECTTYPE in SYSCAT.WORKACTIONSETS.</p> <ul style="list-style-type: none"> <li>• B - Collect basic aggregate activity data. This action type can only be specified if the OBJECTTYPE is 'b' (service class).</li> <li>• C - Allow the execution of any DB2 activity that falls under the work class with which this work action is associated to run and increment the counter for the work class.</li> <li>• D - Collect activity data with details at the database partition of the coordinator of the activity.</li> <li>• E - Collect extended aggregate activity data. This action type can only be specified if the OBJECTTYPE is 'b' (service class).</li> <li>• M - Map to a service subclass. This action type can only be specified if the OBJECTTYPE is 'b' (service class).</li> <li>• P - Prevent the execution of any DB2 activity that falls under the work class with which this work action is associated.</li> <li>• T - The action will be in the form of a threshold. This action type can only be specified if the OBJECTTYPE is blank (database).</li> <li>• U - Map all activities that have a nesting level of zero and all activities nested under this activity to a service subclass. This action type can only be specified if the OBJECTTYPE is 'b' (service class).</li> </ul>
ACTIONTYPE (cont'd)			<ul style="list-style-type: none"> <li>• V - Collect activity data with details and values at the database partition of the coordinator of the activity.</li> <li>• W - Collect activity data without details at the database partition of the coordinator of the activity.</li> <li>• X - Collect activity data with details at the database partition of the coordinator of the activity and collect activity data at all database partitions.</li> <li>• Y - Collect activity data with details and values at the database partition of the coordinator of the activity and collect activity data at all database partitions.</li> <li>• Z - Collect activity data without details at all database partitions.</li> </ul>

## SYSCAT.WORKACTIONS

Table 159. SYSCAT.WORKACTIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
REFOBJECTID	INTEGER	Y	If ACTIONTYPE is 'M' (map) or 'N' (map nested), this value is set to the ID of the service subclass to which the DB2 activity is mapped. If ACTIONTYPE is 'T' (threshold), this value is set to the ID of the threshold to be used. For all other actions, this value is NULL.
REFOBJECTTYPE	VARCHAR (30)		If the ACTIONTYPE is 'M' or 'N', this value is set to 'SERVICE CLASS'; if the ACTIONTYPE is 'T', this value is 'THRESHOLD'; null value otherwise.

## SYSCAT.WORKACTIONSETS

Each row represents a work action set.

Table 160. SYSCAT.WORKACTIONSETS Catalog View

Column Name	Data Type	Nullable	Description
ACTIONSETNAME	VARCHAR (128)		Name of the work action set.
ACTIONSETID	INTEGER		Identifier for the work action set.
WORKCLASSETNAME	VARCHAR (128)	Y	Name of the work class set.
WORKCLASSETID	INTEGER		The identifier of the work class set that is to be mapped to the object specified by the OBJECTID. This column refers to WORKCLASSETID in the SYSCAT.WORKCLASSETS view.
CREATE_TIME	TIMESTAMP		Time at which the work action set was created.
ALTER_TIME	TIMESTAMP		Time at which the work action set was last altered.
ENABLED	CHAR (1)		<ul style="list-style-type: none"><li>• N = This work action set is disabled.</li><li>• Y = This work action set is enabled.</li></ul>
OBJECTTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• b = Service superclass</li><li>• Blank = Database</li></ul>
OBJECTNAME	VARCHAR (128)	Y	Name of the service class.
OBJECTID	INTEGER		The identifier of the object to which the work class set (specified by the WORKCLASSETID) is mapped. If the OBJECTTYPE is blank, the OBJECTID is -1. If the OBJECTTYPE is 'b', the OBJECTID is the ID of the service superclass.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSCAT.WORKCLASSES

Each row represents a work class defined for a work class set.

Table 161. SYSCAT.WORKCLASSES Catalog View

Column Name	Data Type	Nullable	Description
WORKCLASSNAME	VARCHAR (128)		Name of the work class.
WORKCLASSETNAME	VARCHAR (128)	Y	Name of the work class set.
WORKCLASSID	INTEGER		Identifier for the work class.
WORKCLASSETID	INTEGER		Identifier for the work class set to which this work class belongs. This column refers to the WORKCLASSETID column in the SYSCAT.WORKCLASSETS view.
CREATE_TIME	TIMESTAMP		Time at which the work class was created.
ALTER_TIME	TIMESTAMP		Time at which the work class was last altered.
WORKTYPE	SMALLINT		The type of DB2 activity. <ul style="list-style-type: none"> <li>• 1 = ALL</li> <li>• 2 = READ</li> <li>• 3 = WRITE</li> <li>• 4 = CALL</li> <li>• 5 = DML</li> <li>• 6 = DDL</li> <li>• 7 = LOAD</li> </ul>
RANGEUNITS	CHAR (1)		The units to use for the bottom and top range. <ul style="list-style-type: none"> <li>• C = Cardinality</li> <li>• T = Timerons</li> <li>• Blank = Not applicable</li> </ul>
FROMVALUE	DOUBLE	Y	The low value of the range in the units specified by the RANGEUNITS. Null value when RANGEUNITS is blank.
TOVALUE	DOUBLE	Y	The high value of the range in the units specified by the RANGEUNITS. Null value when RANGEUNITS is blank. -1 value is used to indicate no upper bound.
ROUTINESHEMA	VARCHAR (128)	Y	Schema name of the procedures that are called from the CALL statement. Null value when WORKTYPE is not 4 (CALL) or 1 (ALL).
EVALUATIONORDER	SMALLINT		Uniquely identifies the evaluation order used for choosing a work class within a work class set.

## SYSCAT.WORKCLASSETS

Each row represents a work class set.

Table 162. SYSCAT.WORKCLASSETS Catalog View

Column Name	Data Type	Nullable	Description
WORKCLASSETNAME	VARCHAR (128)		Name of the work class set.
WORKCLASSETID	INTEGER		Identifier for the work class set.

## SYSCAT.WORKCLASSETS

Table 162. SYSCAT.WORKCLASSETS Catalog View (continued)

Column Name	Data Type	Nullable	Description
CREATE_TIME	TIMESTAMP		Time at which the work class set was created.
ALTER_TIME	TIMESTAMP		Time at which the work class set was last altered.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSCAT.WORKLOADAUTH

Each row represents a user, group, or role that has been granted USAGE privilege on a workload.

Table 163. SYSCAT.WORKLOADAUTH Catalog View

Column Name	Data Type	Nullable	Description
WORKLOADID	INTEGER		Identifier for the workload.
WORKLOADNAME	VARCHAR (128)		Name of the workload.
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• U = Grantee is an individual user</li></ul>
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		<ul style="list-style-type: none"><li>• G = Grantee is a group</li><li>• R = Grantee is a role</li><li>• U = Grantee is an individual user</li></ul>
USAGEAUTH	CHAR (1)		Indicates whether grantee holds USAGE privilege on the workload. <ul style="list-style-type: none"><li>• N = Not held</li><li>• Y = Held</li></ul>

## SYSCAT.WORKLOADCONNATTR

Each row represents a connection attribute in the definition of a workload.

Table 164. SYSCAT.WORKLOADCONNATTR Catalog View

Column Name	Data Type	Nullable	Description
WORKLOADID	INTEGER		Identifier for the workload.
WORKLOADNAME	VARCHAR (128)		Name of the workload.
CONNATTRTYPE	VARCHAR (30)		Type of the connection attribute. <ul style="list-style-type: none"><li>• 1 = APPLNAME</li><li>• 2 = SYSTEM_USER</li><li>• 3 = SESSION_USER</li><li>• 4 = SESSION_USER GROUP</li><li>• 5 = SESSION_USER ROLE</li><li>• 6 = CURRENT CLIENT_USERID</li><li>• 7 = CURRENT CLIENT_APPLNAME</li><li>• 8 = CURRENT CLIENT_WRKSTNNAME</li><li>• 9 = CURRENT CLIENT_ACCTNG</li></ul>

Table 164. SYSCAT.WORKLOADCONNATTR Catalog View (continued)

Column Name	Data Type	Nullable	Description
CONNATTRVALUE	VARCHAR (1000)		Value of the connection attribute.

## SYSCAT.WORKLOADS

Each row represents a workload.

Table 165. SYSCAT.WORKLOADS Catalog View

Column Name	Data Type	Nullable	Description
WORKLOADID	INTEGER		Identifier for the workload.
WORKLOADNAME	VARCHAR (128)		Name of the workload.
EVALUATIONORDER	SMALLINT		Evaluation order used for choosing a workload.
CREATE_TIME	TIMESTAMP		Time at which the workload was created.
ALTER_TIME	TIMESTAMP		Time at which the workload was last altered.
ENABLED	CHAR (1)		<ul style="list-style-type: none"> <li>N = This workload is disabled.</li> <li>Y = This workload is enabled.</li> </ul>
ALLOWACCESS	CHAR (1)		<ul style="list-style-type: none"> <li>N = A UOW associated with this workload will be rejected.</li> <li>Y = A unit of work (UOW) associated with this workload can access the database.</li> </ul>
SERVICECLASSNAME	VARCHAR (128)		Name of the service subclass to which a unit of work (associated with this workload) is assigned.
PARENTSERVICECLASSNAME	VARCHAR (128)	Y	Name of the service superclass to which a unit of work (associated with this workload) is assigned.
COLLECTAGGACTDATA	CHAR (1)		<p>Specifies what aggregate activity data should be captured for the workload by the applicable event monitor.</p> <ul style="list-style-type: none"> <li>N = None</li> </ul>
COLLECTACTDATA	CHAR (1)		<p>Specifies what activity data should be collected by the applicable event monitor.</p> <ul style="list-style-type: none"> <li>D = Activity data with details</li> <li>N = None</li> <li>V = Activity data with details and valuesApplies when the COLLECT column is set to 'C'</li> <li>W = Activity data without details</li> </ul>
COLLECTACTPARTITION	CHAR (1)		<p>Specifies where activity data is collected.</p> <ul style="list-style-type: none"> <li>C = Database partition of the coordinator of the activity</li> <li>D = All database partitions</li> </ul>
EXTERNALNAME	VARCHAR (128)	Y	Reserved for future use.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSCAT.WRAPOPTIONS

---

## SYSCAT.WRAPOPTIONS

Each row represents a wrapper-specific option.

Table 166. SYSCAT.WRAPOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
WRAPNAME	VARCHAR (128)		Name of the wrapper.
OPTION	VARCHAR (128)		Name of the wrapper option.
SETTING	VARCHAR (2048)		Value of the wrapper option.

---

## SYSCAT.WRAPPERS

Each row represents a registered wrapper.

Table 167. SYSCAT.WRAPPERS Catalog View

Column Name	Data Type	Nullable	Description
WRAPNAME	VARCHAR (128)		Name of the wrapper.
WRAPTYPE	CHAR (1)		Type of wrapper. <ul style="list-style-type: none"><li>• N = Non-relational</li><li>• R = Relational</li></ul>
WRAPVERSION	INTEGER		Version of the wrapper.
LIBRARY	VARCHAR (255)		Name of the file that contains the code used to communicate with the data sources that are associated with this wrapper.
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

---

## SYSCAT.XDBMAPGRAPHS

Each row represents a schema graph for an XDB map (XSR object).

Table 168. SYSCAT.XDBMAPGRAPHS Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Unique generated identifier for an XSR object.
OBJECTSCHEMA	VARCHAR (128)		Schema name of the XSR object.
OBJECTNAME	VARCHAR (128)		Unqualified name of the XSR object.
SCHEMAGRAPHID	INTEGER		Schema graph identifier, which is unique within an XDB map identifier.
NAMESPACE	VARCHAR (1001)	Y	String identifier for the namespace URI of the root element.
ROOTELEMENT	VARCHAR (1001)	Y	String identifier for the element name of the root element.

---

## SYSCAT.XDBMAPSHREDTREES

Each row represents one shred tree for a given schema graph identifier.

Table 169. SYSCAT.XDBMAPSHREDTREES Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Unique generated identifier for an XSR object.
OBJECTSCHEMA	VARCHAR (128)		Schema name of the XSR object.
OBJECTNAME	VARCHAR (128)		Unqualified name of the XSR object.
SCHEMAGRAPHID	INTEGER		Schema graph identifier, which is unique within an XDB map identifier.
SHREDTREEID	INTEGER		Shred tree identifier, which is unique within an XDB map identifier.
MAPPINGDESCRIPTION	CLOB (1M)	Y	Diagnostic mapping information.

## SYSCAT.XSROBJECTAUTH

Each row represents a user or group that has been granted the USAGE privilege on a particular XSR object.

Table 170. SYSCAT.XSROBJECTAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		<ul style="list-style-type: none"> <li>S = Grantor is the system</li> <li>U = Grantor is an individual user</li> </ul>
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		<ul style="list-style-type: none"> <li>G = Grantee is a group</li> <li>R = Grantee is a role</li> <li>U = Grantee is an individual user</li> </ul>
OBJECTID	BIGINT		Identifier for the XSR object.
USAGEAUTH	CHAR (1)		Privilege to use the XSR object and its components. <ul style="list-style-type: none"> <li>N = Not held</li> <li>Y = Held</li> </ul>

## SYSCAT.XSROBJECTCOMPONENTS

Each row represents an XSR object component.

Table 171. SYSCAT.XSROBJECTCOMPONENTS Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Unique generated identifier for an XSR object.
OBJECTSCHEMA	VARCHAR (128)		Schema name of the XSR object.
OBJECTNAME	VARCHAR (128)		Unqualified name of the XSR object.
COMPONENTID	BIGINT		Unique generated identifier for an XSR object component.
TARGETNAMESPACE	VARCHAR (1001)	Y	String identifier for the target namespace.
SCHEMALOCATION	VARCHAR (1001)	Y	String identifier for the schema location.

## SYSCAT.XSROBJECTCOMPONENTS

Table 171. SYSCAT.XSROBJECTCOMPONENTS Catalog View (continued)

Column Name	Data Type	Nullable	Description
COMPONENT	BLOB (30M)		External representation of the component.
CREATE_TIME	TIMESTAMP		Time at which the XSR object component was registered.
STATUS	CHAR (1)		Registration status. <ul style="list-style-type: none"><li>• C = Complete</li><li>• I = Incomplete</li></ul>

## SYSCAT.XSROBJECTDEP

Each row represents a dependency of an XSR object on some other object. The XSR object depends on the object of type BTYPE of name BNAME, so a change to the object affects the XSR object.

Table 172. SYSCAT.XSROBJECTDEP Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Unique generated identifier for an XSR object.
OBJECTSCHEMA	VARCHAR (128)		Schema name of the XSR object.
OBJECTNAME	VARCHAR (128)		Unqualified name of the XSR object.
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: <ul style="list-style-type: none"><li>• A = Alias</li><li>• B = Trigger</li><li>• F = Routine instance</li><li>• H = Hierachy table</li><li>• K = Package</li><li>• L = Detached table</li><li>• O = Privilege dependency on all subtables or subviews in a table or view hierarchy</li><li>• Q = Sequence</li><li>• R = Structured type</li><li>• S = Materialized query table</li><li>• T = Table (not typed)</li><li>• U = Typed table</li><li>• V = View (not typed)</li><li>• W = Typed view</li><li>• X = Index extension</li><li>• Z = XSR object</li><li>• v = Global variable</li></ul>
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.



Table 172. SYSCAT.XSROBJECTDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
TABAUTH	SMALLINT	Y	If BTYPE = 'O', 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by a dependent trigger; null value otherwise.

## SYSCAT.XSROBJECTHIERARCHIES

Each row represents the hierarchical relationship between an XSR object and its components.

Table 173. SYSCAT.XSROBJECTHIERARCHIES Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Identifier for an XSR object.
COMPONENTID	BIGINT		Identifier for an XSR component.
HTYPE	CHAR (1)		Hierarchy type. <ul style="list-style-type: none"> <li>• D = Document</li> <li>• N = Top-level namespace</li> <li>• P = Primary document</li> </ul>
TARGETNAMESPACE	VARCHAR (1001)	Y	String identifier for the component's target namespace.
SCHEMALOCATION	VARCHAR (1001)	Y	String identifier for the component's schema location.

## SYSCAT.XSROBJECTS

Each row represents an XML schema repository object.

Table 174. SYSCAT.XSROBJECTS Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Unique generated identifier for an XSR object.
OBJECTSCHEMA	VARCHAR (128)		Schema name of the XSR object.
OBJECTNAME	VARCHAR (128)		Unqualified name of the XSR object.
TARGETNAMESPACE	VARCHAR (1001)	Y	String identifier for the target namespace, or public identifier.
SCHEMALOCATION	VARCHAR (1001)	Y	String identifier for the schema location, or system identifier.
OBJECTINFO	XML	Y	Metadata document.
OBJECTTYPE	CHAR (1)		XSR object type. <ul style="list-style-type: none"> <li>• D = DTD</li> <li>• E = External entity</li> <li>• S = XML schema</li> </ul>
OWNER	VARCHAR (128)		Authorization ID under which the XSR object was registered.

## SYSCAT.XSROBJECTS

Table 174. SYSCAT.XSROBJECTS Catalog View (continued)

Column Name	Data Type	Nullable	Description
OWNERTYPE	CHAR (1)		<ul style="list-style-type: none"><li>• S = The owner is the system</li><li>• U = The owner is an individual user</li></ul>
CREATE_TIME	TIMESTAMP		Time at which the object was registered.
ALTER_TIME	TIMESTAMP		Time at which the object was last updated (replaced).
STATUS	CHAR (1)		Registration status. <ul style="list-style-type: none"><li>• C = Complete</li><li>• I = Incomplete</li><li>• R = Replace</li><li>• T = Temporary</li></ul>
DECOMPOSITION	CHAR (1)		Indicates whether or not decomposition (shredding) is enabled on this XSR object. <ul style="list-style-type: none"><li>• N = Not enabled</li><li>• X = Inoperative</li><li>• Y = Enabled</li></ul>
REMARKS	VARCHAR (254)	Y	User-provided comments, or null.

## SYSIBM.SYSDUMMY1

Contains one row. This view is available for applications that require compatibility with DB2 Universal Database for z/OS.

Table 175. SYSIBM.SYSDUMMY1 Catalog View

Column Name	Data Type	Nullable	Description
IBMREQD	CHAR(1)		'Y'

## SYSSTAT.COLDIST

Each row represents the *n*th most frequent value of some column, or the *n*th quantile (cumulative distribution) value of the column. Applies to columns of real tables only (not views). No statistics are recorded for inherited columns of typed tables.

Table 176. SYSSTAT.COLDIST Catalog View

Column Name	Data Type	Nullable	Updat-able	Description
TABSCHEMA	VARCHAR (128)			Schema name of the table to which the statistics apply.
TABNAME	VARCHAR (128)			Unqualified name of the table to which the statistics apply.
COLNAME	VARCHAR (128)			Name of the column to which the statistics apply.
TYPE	CHAR (1)			<ul style="list-style-type: none"><li>• F = Frequency value</li><li>• Q = Quantile value</li></ul>

Table 176. SYSSTAT.COLDIST Catalog View (continued)

Column Name	Data Type	Nullable	Updat-able	Description
SEQNO	SMALLINT			If TYPE = 'F', <i>n</i> in this column identifies the <i>n</i> th most frequent value. If TYPE = 'Q', <i>n</i> in this column identifies the <i>n</i> th quantile value.
COLVALUE <sup>1</sup>	VARCHAR (254)	Y	Y	Data value as a character literal or a null value.
VALCOUNT	BIGINT		Y	If TYPE = 'F', VALCOUNT is the number of occurrences of COLVALUE in the column. If TYPE = 'Q', VALCOUNT is the number of rows whose value is less than or equal to COLVALUE.
DISTCOUNT <sup>2</sup>	BIGINT	Y	Y	If TYPE = 'Q', this column records the number of distinct values that are less than or equal to COLVALUE (null if unavailable).

**Note:**

1. In the catalog view, the value of COLVALUE is always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column's table, and will therefore use actual column values when applied during query optimization.
2. DISTCOUNT is collected only for columns that are the first key column in an index.

## SYSSTAT.COLGROUPDIST

Each row represents the value of the column in a column group that makes up the *n*th most frequent value of the column group or the *n*th quantile value of the column group.

Table 177. SYSSTAT.COLGROUPDIST Catalog View

Column Name	Data Type	Nullable	Updat-able	Description
COLGROUPID	INTEGER			Identifier for the column group.
TYPE	CHAR (1)			<ul style="list-style-type: none"> <li>• F = Frequency value</li> <li>• Q = Quantile value</li> </ul>
ORDINAL	SMALLINT			Ordinal number of the column in the column group.
SEQNO	SMALLINT			If TYPE = 'F', <i>n</i> in this column identifies the <i>n</i> th most frequent value. If TYPE = 'Q', <i>n</i> in this column identifies the <i>n</i> th quantile value.
COLVALUE	VARCHAR (254)		Y	Data value as a character literal or a null value.

## SYSSTAT.COLGROUPDISTCOUNTS

Each row represents the distribution statistics that apply to the *n*th most frequent value of a column group or the *n*th quantile of a column group.

## SYSSTAT.COLGROUPDISTCOUNTS

Table 178. SYSSTAT.COLGROUPDISTCOUNTS Catalog View

Column Name	Data Type	Nullable	Updat-able	Description
COLGROUPID	INTEGER			Identifier for the column group.
TYPE	CHAR (1)			<ul style="list-style-type: none"><li>• F = Frequency value</li><li>• Q = Quantile value</li></ul>
SEQNO	SMALLINT			Sequence number <i>n</i> representing the <i>n</i> th TYPE value.
VALCOUNT	BIGINT		Y	If TYPE = 'F', VALCOUNT is the number of occurrences of COLVALUE for the column group with this SEQNO. If TYPE = 'Q', VALCOUNT is the number of rows whose value is less than or equal to COLVALUE for the column group with this SEQNO.
DISTCOUNT	BIGINT		Y	If TYPE = 'Q', this column records the number of distinct values that are less than or equal to COLVALUE for the column group with this SEQNO (null if unavailable).

## SYSSTAT.COLGROUPS

Each row represents a column group and statistics that apply to the entire column group.

Table 179. SYSSTAT.COLGROUPS Catalog View

Column Name	Data Type	Nullable	Updat-able	Description
COLGROUPSCHEMA	VARCHAR (128)			Schema name of the column group.
COLGROUPNAME	VARCHAR (128)			Unqualified name of the column group.
COLGROUPID	INTEGER			Identifier for the column group.
COLGROUPCARD	BIGINT		Y	Cardinality of the column group.
NUMFREQ_VALUES	SMALLINT			Number of frequent values collected for the column group.
NUMQUANTILES	SMALLINT			Number of quantiles collected for the column group.

## SYSSTAT.COLUMNS

Each row represents a column defined for a table, view, or nickname.

Table 180. SYSSTAT.COLUMNS Catalog View

Column Name	Data Type	Nullable	Updat-able	Description
TABSCHEMA	VARCHAR (128)			Schema name of the table, view, or nickname that contains the column.
TABNAME	VARCHAR (128)			Unqualified name of the table, view, or nickname that contains the column.

Table 180. SYSSTAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Updat-able	Description
COLNAME	VARCHAR (128)			Name of the column.
COLCARD	BIGINT		Y	Number of distinct values in the column; -1 if statistics are not collected; -2 for inherited columns and columns of hierarchy tables.
HIGH2KEY <sup>1</sup>	VARCHAR (254)	Y	Y	Second-highest data value. Representation of numeric data changed to character literals. Empty if statistics are not collected. Empty for inherited columns and columns of hierarchy tables.
LOW2KEY <sup>1</sup>	VARCHAR (254)	Y	Y	Second-lowest data value. Representation of numeric data changed to character literals. Empty if statistics are not collected. Empty for inherited columns and columns of hierarchy tables.
AVGCOLLEN	INTEGER		Y	Average space (in bytes) required for the column; -1 if a long field or LOB, or statistics have not been collected; -2 for inherited columns and columns of hierarchy tables.
NUMNULLS	BIGINT		Y	Number of null values in the column; -1 if statistics are not collected.
SUB_COUNT	SMALLINT		Y	Average number of sub-elements in the column. Applicable to character string columns only.
SUB_DELIM_LENGTH	SMALLINT		Y	Average length of the delimiters that separate each sub-element in the column. Applicable to character string columns only.
AVGCOLLENCHAR	INTEGER		Y	Average number of characters (based on the collation in effect for the column) required for the column; -1 if a long field or LOB, or statistics have not been collected; -2 for inherited columns and columns of hierarchy tables.

**Note:**

1. In the catalog view, the values of HIGH2KEY and LOW2KEY are always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column's table, and will therefore use actual column values when applied during query optimization.

---

## SYSSTAT.INDEXES

Each row represents an index. Indexes on typed tables are represented by two rows: one for the "logical index" on the typed table, and one for the "H-index" on the hierarchy table.

## SYSSTAT.INDEXES

Table 181. SYSSTAT.INDEXES Catalog View

Column Name	Data Type	Nullable	Updat-able	Description
INDSCHEMA	VARCHAR (128)			Schema name of the index.
INDNAME	VARCHAR (128)			Unqualified name of the index.
TABSCHEMA	VARCHAR (128)			Schema name of the table or nickname on which the index is defined.
TABNAME	VARCHAR (128)			Unqualified name of the table or nickname on which the index is defined.
COLNAMES	VARCHAR (640)			This column is no longer used and will be removed in the next release.
NLEAF	BIGINT		Y	Number of leaf pages; -1 if statistics are not collected.
NLEVELS	SMALLINT		Y	Number of index levels; -1 if statistics are not collected.
FIRSTKEYCARD	BIGINT		Y	Number of distinct first-key values; -1 if statistics are not collected.
FIRST2KEYCARD	BIGINT		Y	Number of distinct keys using the first two columns of the index; -1 if statistics are not collected, or if not applicable.
FIRST3KEYCARD	BIGINT		Y	Number of distinct keys using the first three columns of the index; -1 if statistics are not collected, or if not applicable.
FIRST4KEYCARD	BIGINT		Y	Number of distinct keys using the first four columns of the index; -1 if statistics are not collected, or if not applicable.
FULLKEYCARD	BIGINT		Y	Number of distinct full-key values; -1 if statistics are not collected.
CLUSTERRATIO <sup>4</sup>	SMALLINT		Y	Degree of data clustering with the index; -1 if statistics are not collected or if detailed index statistics are collected (in which case, CLUSTERFACTOR will be used instead).
CLUSTERFACTOR <sup>4</sup>	DOUBLE		Y	Finer measurement of the degree of clustering; -1 if statistics are not collected or if the index is defined on a nickname.
SEQUENTIAL_PAGES	BIGINT		Y	Number of leaf pages located on disk in index key order with few or no large gaps between them; -1 if statistics are not collected.
DENSITY	INTEGER		Y	Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percent (integer between 0 and 100); -1 if statistics are not collected.

Table 181. SYSSTAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Updat-able	Description
PAGE_FETCH_PAIRS <sup>4</sup>	VARCHAR (520)		Y	A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the table with this index using that hypothetical buffer. Zero-length string if no data is available.
NUMRIDS <sup>4</sup>	BIGINT		Y	Total number of row identifiers (RIDs) or block identifiers (BIDs) in the index; -1 if not known.
NUMRIDS_DELETED <sup>4</sup>	BIGINT		Y	Total number of row identifiers (or block identifiers) in the index that are marked deleted, excluding those identifiers on leaf pages on which all the identifiers are marked deleted.
NUM_EMPTY_LEAFS	BIGINT		Y	Total number of index leaf pages that have all of their row identifiers (or block identifiers) marked deleted.
AVERAGE_RANDOM_FETCH_PAGES <sup>1,2,4</sup>	DOUBLE		Y	Average number of random table pages between sequential page accesses when fetching using the index; -1 if not known.
AVERAGE_RANDOM_PAGES <sup>2</sup>	DOUBLE		Y	Average number of random table pages between sequential page accesses; -1 if not known.
AVERAGE_SEQUENCE_GAP <sup>2</sup>	DOUBLE		Y	Gap between index page sequences. Detected through a scan of index leaf pages, each gap represents the average number of index pages that must be randomly fetched between sequences of index pages; -1 if not known.
AVERAGE_SEQUENCE_FETCH_GAP <sup>1,2,4</sup>	DOUBLE		Y	Gap between table page sequences when fetching using the index. Detected through a scan of index leaf pages, each gap represents the average number of table pages that must be randomly fetched between sequences of table pages; -1 if not known.
AVERAGE_SEQUENCE_PAGES <sup>2</sup>	DOUBLE		Y	Average number of index pages that are accessible in sequence (that is, the number of index pages that the prefetchers would detect as being in sequence); -1 if not known.
AVERAGE_SEQUENCE_FETCH_PAGES <sup>1,2,4</sup>	DOUBLE		Y	Average number of table pages that are accessible in sequence (that is, the number of table pages that the prefetchers would detect as being in sequence) when fetching using the index; -1 if not known.

## SYSSTAT.INDEXES

Table 181. SYSSTAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Updat-able	Description
AVGPARTITION_ CLUSTERRATIO <sup>3,4</sup>	SMALLINT		Y	Degree of data clustering within a single data partition. -1 if the table is not partitioned, if statistics are not collected, or if detailed statistics are collected (in which case AVGPARTITION_ CLUSTERFACTOR will be used instead).
AVGPARTITION_ CLUSTERFACTOR <sup>3,4</sup>	DOUBLE		Y	Finer measurement of the degree of clustering within a single data partition. -1 if the table is not partitioned, if statistics are not collected, or if the index is defined on a nickname.
AVGPARTITION_PAGE_ FETCH_PAIRS <sup>3,4</sup>	VARCHAR (520)		Y	A list of paired integers in character form. Each pair represents a potential buffer pool size and the corresponding page fetches required to access a single data partition from the table. Zero-length string if no data is available, or if the table is not partitioned.
DATAPARTITION_ CLUSTERFACTOR	DOUBLE		Y	A statistic measuring the "clustering" of the index keys with regard to data partitions. It is a number between 0 and 1, with 1 representing perfect clustering and 0 representing no clustering.
INDCARD	BIGINT		Y	Cardinality of the index. This might be different from the cardinality of the table for indexes that do not have a one-to-one relationship between the table rows and the index entries.

### Note:

1. When using DMS table spaces, this statistic cannot be computed.
2. Prefetch statistics are not gathered during a LOAD...STATISTICS YES, or a CREATE INDEX...COLLECT STATISTICS operation, or when the database configuration parameter *seqdetect* is turned off.
3. AVGPARTITION\_CLUSTERRATIO, AVGPARTITION\_CLUSTERFACTOR, and AVGPARTITION\_PAGE\_FETCH\_PAIRS measure the degree of clustering within a single data partition (local clustering). CLUSTERRATIO, CLUSTERFACTOR, and PAGE\_FETCH\_PAIRS measure the degree of clustering in the entire table (global clustering). Global clustering and local clustering values can diverge significantly if the table partitioning key is not a prefix of the index key, or when the table partitioning key and the index key are logically independent of each other.
4. This statistic cannot be updated if the index type is 'XPTH' (an XML path index).
5. Because logical indexes on an XML column do not have statistics, the SYSSTAT.INDEXES catalog view excludes rows whose index type is 'XVIL'.

## SYSSTAT.ROUTINES

Each row represents a user-defined routine (scalar function, table function, sourced function, method, or procedure). Does not include built-in functions.



Table 182. SYSSTAT.ROUTINES Catalog View

Column Name	Data Type	Nullable	Updat-able	Description
ROUTINESHEMA	VARCHAR (128)			Schema name of the routine.
ROUTINENAME	VARCHAR (128)			Unqualified name of the routine.
ROUTINETYPE	CHAR (1)			Type of routine. <ul style="list-style-type: none"> <li>• F = Function</li> <li>• M = Method</li> <li>• P = Procedure</li> </ul>
SPECIFICNAME	VARCHAR (128)			Name of the routine instance (might be system-generated).
IOS_PER_INVOC	DOUBLE		Y	Estimated number of inputs/outputs (I/Os) per invocation; 0 is the default; -1 if not known.
INSTS_PER_INVOC	DOUBLE		Y	Estimated number of instructions per invocation; 450 is the default; -1 if not known.
IOS_PER_ARGBYTE	DOUBLE		Y	Estimated number of I/Os per input argument byte; 0 is the default; -1 if not known.
INSTS_PER_ARGBYTE	DOUBLE		Y	Estimated number of instructions per input argument byte; 0 is the default; -1 if not known.
PERCENT_ARGBYTES	SMALLINT		Y	Estimated average percent of input argument bytes that the routine will actually read; 100 is the default; -1 if not known.
INITIAL_IOS	DOUBLE		Y	Estimated number of I/Os performed the first time that the routine is invoked; 0 is the default; -1 if not known.
INITIAL_INSTS	DOUBLE		Y	Estimated number of instructions executed the first time the routine is invoked; 0 is the default; -1 if not known.
CARDINALITY	BIGINT		Y	Predicted cardinality of a table function; -1 if not known, or if the routine is not a table function.
SELECTIVITY	DOUBLE		Y	For user-defined predicates; -1 if there are no user-defined predicates.

## SYSSTAT.TABLES

Each row represents a table, view, alias, or nickname. Each table or view hierarchy has one additional row representing the hierarchy table or hierarchy view that implements the hierarchy. Catalog tables and views are included.

Table 183. SYSSTAT.TABLES Catalog View

Column Name	Data Type	Nullable	Updat-able	Description
TABSCHEMA	VARCHAR (128)			Schema name of the object.
TABNAME	VARCHAR (128)			Unqualified name of the object.

## SYSSTAT.TABLES

Table 183. SYSSTAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Updat-able	Description
CARD	BIGINT		Y	Total number of rows; -1 if statistics are not collected.
NPAGES	BIGINT		Y	Total number of pages on which the rows of the table exist; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.
FPAGES	BIGINT		Y	Total number of pages; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.
OVERFLOW	BIGINT		Y	Total number of overflow records in the table; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.
CLUSTERED	CHAR (1)	Y		<ul style="list-style-type: none"> <li>• Y = Table is multidimensionally clustered (even if only by one dimension)</li> <li>• Null value = Table is not multidimensionally clustered</li> </ul>
ACTIVE_BLOCKS	BIGINT		Y	Total number of active blocks in the table, or -1. Applies to multidimensional clustering (MDC) tables only.
AVGCOMPRESSEDROWSIZE	SMALLINT		Y	Average length (in bytes) of compressed rows in this table; -1 if statistics are not collected.
AVGROWCOMPRESSIONRATIO	REAL		Y	For compressed rows in the table, this is the average compression ratio by row; that is, the average uncompressed row length divided by the average compressed row length; -1 if statistics are not collected.
AVGROWSIZE	SMALLINT			Average length (in bytes) of both compressed and uncompressed rows in this table; -1 if statistics are not collected.
PCTROWSCOMPRESSED	REAL		Y	Compressed rows as a percentage of the total number of rows in the table; -1 if statistics are not collected.
PCTPAGESSAVED	SMALLINT		Y	Approximate percentage of pages saved in the table as a result of row compression. This value includes overhead bytes for each user data row in the table, but does not include the space that is consumed by dictionary overhead; -1 if statistics are not collected.

---

## Appendix E. Federated systems

---

### Valid server types in SQL statements

Server types indicate the kind of data source that the server definition represents.

Server types vary by vendor, purpose, and operating system. Supported values depend on the data source.

For most data sources, you must specify a valid server type in the CREATE SERVER statement.

*Table 184. Data sources and server types*

Data source	Server type
BioRS	A server type is not required in the CREATE SERVER statement.
Blast	BLASTN, BLASTP, BLASTX, TBLASTN, TBLASTX
Entrez	NUCLEOTIDE, OMIM, PUBMED
Excel	A server type is not required in the CREATE SERVER statement.
HMMER	PFAM, SEARCH
IBM DB2 Universal Database for Linux, UNIX, and Windows	DB2/UDB
IBM DB2 Universal Database for System i and AS/400®	DB2/ISERIES
IBM DB2 Universal Database for z/OS	DB2/ZOS
IBM DB2 for VM	DB2/VM
Informix	INFORMIX
Microsoft SQL Server	MSSQLSERVER (Required for data sources supported by the DataDirect Connect ODBC 4.2 (or later) driver or the Microsoft SQL Server ODBC 3.0 (or later) driver.)
ODBC	ODBC (Required for ODBC data sources that are supported by the ODBC 3.x driver.)
OLE DB	A server type is not required in the CREATE SERVER statement.
Oracle	ORACLE (Required for Oracle data sources supported by Oracle NET8 client software.)
Sybase (CTLIB)	SYBASE
Table-structured files	A server type is not required in the CREATE SERVER statement.
Teradata	TERADATA
Web services	A server type is not required in the CREATE SERVER statement.
WebSphere Business Integration	WBI

## Valid server types in SQL statements

Table 184. Data sources and server types (continued)

Data source	Server type
XML	A server type is not required in the CREATE SERVER statement.

## Function mapping options for federated systems

The federated server provides default mappings between DB2 functions and data source functions. For most data sources, the default function mappings are in the wrappers. To use a data source function that the federated server does not recognize or to change the default mapping, you create a function mapping.

When you create a function mapping, you specify the name of the data source function and must enable the mapped function. Then when you use the mapped function, the query optimizer compares the cost of running the function at the data source with the cost of running the function at the federated server.

Table 185. Options for function mappings

Name	Description
DISABLE	Enable or disable a default function mapping. Valid values are Y and N. The default is N.
REMOTE_NAME	The name of the data source function. The default is the local name.

## Default forward data type mappings

The two kinds of mappings between data source data types and federated database data types are forward type mappings and reverse type mappings. In a forward type mapping, the mapping is from a remote type to a comparable local type.

You can override a default type mapping, or create a new type mapping with the CREATE TYPE MAPPING statement.

These mappings are valid with all the supported versions, unless otherwise noted.

For all default forward data types mapping from a data source to the federated database, the federated schema is SYSIBM.

The following tables show the default forward mappings between federated database data types and data source data types.

### DB2 Database for Linux, UNIX, and Windows data sources

The following table lists the forward default data type mappings for DB2 Database for Linux, UNIX, and Windows data sources.

Table 186. DB2 Database for Linux, UNIX, and Windows forward default data type mappings (Not all columns shown)

REMOTE_ TYPE NAME	REMOTE_ LOWER_ _LEN	REMOTE_ UPPER_ LEN	REMOTE_ LOWER_ SCALE	REMOTE_ UPPER_ SCALE	REMOTE_ BIT_ DATA	REMOTE_ DATA_ OPERATORS	FEDERATED_ TYPE NAME	FEDERATED_ LENGTH	FEDERATED_ SCALE	FEDERATED_ BIT_ DATA
BIGINT	-	-	-	-	-	-	BIGINT	-	0	-

## DB2 Database for Linux, UNIX, and Windows data sources

Table 186. DB2 Database for Linux, UNIX, and Windows forward default data type mappings (Not all columns shown) (continued)

REMOTE_ TYPE NAME	REMOTE_ LOWER_ _LEN	REMOTE_ UPPER_ LEN	REMOTE_ LOWER_ SCALE	REMOTE_ UPPER_ SCALE	REMOTE_ BIT_ DATA	REMOTE_ DATA_ OPERATORS	FEDERATED_ TYPE NAME	FEDERATED_ LENGTH	FEDERATED_ SCALE	FEDERATED_ BIT_ DATA
BLOB	-	-	-	-	-	-	BLOB	-	-	-
CHAR	-	-	-	-	-	-	CHAR	-	0	N
CHAR	-	-	-	-	Y	-	CHAR	-	0	Y
CLOB	-	-	-	-	-	-	CLOB	-	-	-
DATE	-	-	-	-	-	-	DATE	-	0	-
DBCLOB	-	-	-	-	-	-	DBCLOB	-	-	-
DECIMAL	-	-	-	-	-	-	DECIMAL	-	-	-
DOUBLE	-	-	-	-	-	-	DOUBLE	-	-	-
FLOAT	-	-	-	-	-	-	DOUBLE	-	-	-
GRAPHIC	-	-	-	-	-	-	GRAPHIC	-	0	N
INTEGER	-	-	-	-	-	-	INTEGER	-	0	-
LONGVAR	-	-	-	-	N	-	CLOB	-	-	-
LONGVAR	-	-	-	-	Y	-	BLOB	-	-	-
LONGVARG	-	-	-	-	-	-	DBCLOB	-	-	-
REAL	-	-	-	-	-	-	REAL	-	-	-
SMALLINT	-	-	-	-	-	-	SMALLINT	-	0	-
TIME	-	-	-	-	-	-	TIME	-	0	-
TIMESTAMP	-	-	-	-	-	-	TIMESTAMP	-	0	-
TIMESTMP	-	-	-	-	-	-	TIMESTAMP	-	0	-
VARCHAR	-	-	-	-	-	-	VARCHAR	-	0	N
VARCHAR	-	-	-	-	Y	-	VARCHAR	-	0	Y
VARGRAPH	-	-	-	-	-	-	VARGRAPHIC	-	0	N
VARGRAPHIC	-	-	-	-	-	-	VARGRAPHIC	-	0	N

## DB2 for System i data sources

The following table lists the forward default data type mappings for DB2 for System i data sources.

Table 187. DB2 for System i forward default data type mappings (Not all columns shown)

Remote Typename	Remote Lower Len	Remote Upper Len	Remote Lower Scale	Remote Upper Scale	Remote Bit Data	Remote Data Operators	Federated Typename	Federated Length	Federated Scale	Federated Bit Data
BLOB	-	-	-	-	-	-	BLOB	-	-	-
CHAR	1	254	-	-	-	-	CHAR	-	0	N
CHAR	255	32672	-	-	-	-	VARCHAR	-	0	N
CHAR	1	254	-	-	Y	-	CHAR	-	0	Y
CHAR	255	32672	-	-	Y	-	VARCHAR	-	0	Y
CLOB	-	-	-	-	-	-	CLOB	-	-	-
DATE	-	-	-	-	-	-	DATE	-	0	-
DBCLOB	-	-	-	-	-	-	DBCLOB	-	-	-
DECIMAL	-	-	-	-	-	-	DECIMAL	-	-	-
FLOAT	4	-	-	-	-	-	REAL	-	-	-
FLOAT	8	-	-	-	-	-	DOUBLE	-	-	-

## DB2 for System i data sources

Table 187. DB2 for System i forward default data type mappings (Not all columns shown) (continued)

Remote Typename	Remote Lower Len	Remote Upper Len	Remote Lower Scale	Remote Upper Scale	Remote Bit Data	Remote Data Operators	Federated Typename	Federated Length	Federated Scale	Federated Bit Data
GRAPHIC	1	127	-	-	-	-	GRAPHIC	-	0	N
GRAPHIC	128	16336	-	-	-	-	VARGRAPHIC	-	0	N
INTEGER	-	-	-	-	-	-	INTEGER	-	0	-
NUMERIC	-	-	-	-	-	-	DECIMAL	-	-	-
SMALLINT	-	-	-	-	-	-	SMALLINT	-	0	-
TIME	-	-	-	-	-	-	TIME	-	0	-
TIMESTAMP	-	-	-	-	-	-	TIMESTAMP	-	0	-
TIMESTMP	-	-	-	-	-	-	TIMESTAMP	-	0	-
VARCHAR	1	32672	-	-	-	-	VARCHAR	-	0	N
VARCHAR	1	32672	-	-	Y	-	VARCHAR	-	0	Y
VARG	1	16336	-	-	-	-	VARGRAPHIC	-	0	N
VARGRAPHIC	1	16336	-	-	-	-	VARGRAPHIC	-	0	N

## DB2 for VM and VSE data sources

The following table lists the forward default data type mappings for DB2 for VM and VSE data sources.

Table 188. DB2 Server for VM and VSE forward default data type mappings (Not all columns shown)

Remote Typename	Remote Lower Len	Remote Upper Len	Remote Lower Scale	Remote Upper Scale	Remote Bit Data	Remote Data Operators	Federated Typename	Federated Length	Federated Scale	Federated Bit Data
BLOB	-	-	-	-	-	-	BLOB	-	-	-
CHAR	1	254	-	-	-	-	CHAR	-	0	N
CHAR	1	254	-	-	Y	-	CHAR	-	0	Y
CLOB	-	-	-	-	-	-	CLOB	-	-	-
DATE	-	-	-	-	-	-	DATE	-	0	-
DBAHW	-	-	-	-	-	-	SMALLINT	-	0	-
DBAINT	-	-	-	-	-	-	INTEGER	-	0	-
DBCLOB	-	-	-	-	-	-	DBCLOB	-	-	-
DECIMAL	-	-	-	-	-	-	DECIMAL	-	-	-
FLOAT	4	-	-	-	-	-	REAL	-	-	-
FLOAT	8	-	-	-	-	-	DOUBLE	-	-	-
GRAPHIC	1	127	-	-	-	-	GRAPHIC	-	0	N
INTEGER	-	-	-	-	-	-	INTEGER	-	-	-
SMALLINT	-	-	-	-	-	-	SMALLINT	-	-	-
TIME	-	-	-	-	-	-	TIME	-	0	-
TIMESTAMP	-	-	-	-	-	-	TIMESTAMP	-	0	-
TIMESTMP	-	-	-	-	-	-	TIMESTAMP	-	0	-
VARCHAR	1	32672	-	-	-	-	VARCHAR	-	0	N
VARCHAR	1	32672	-	-	Y	-	VARCHAR	-	0	Y
VARGRAPHIC	1	16336	-	-	-	-	VARGRAPHIC	-	0	N
VARGRAPH	1	16336	-	-	-	-	VARGRAPHIC	-	0	N

## DB2 for z/OS data sources

The following table lists the forward default data type mappings for DB2 for z/OS data sources.

Table 189. DB2 for z/OS forward default data type mappings (Not all columns shown)

Remote Typename	Remote Lower Len	Remote Upper Len	Remote Lower Scale	Remote Upper Scale	Remote Bit Data	Remote Data Operators	Federated Typename	Federated Length	Federated Scale	Federated Bit Data
BLOB	-	-	-	-	-	-	BLOB	-	-	-
CHAR	1	254	-	-	-	-	CHAR	-	0	N
CHAR	255	32672	-	-	-	-	VARCHAR	-	0	N
CHAR	1	254	-	-	Y	-	CHAR	-	0	Y
CHAR	255	32672	-	-	Y	-	VARCHAR	-	0	Y
CLOB	-	-	-	-	-	-	CLOB	-	-	-
DATE	-	-	-	-	-	-	DATE	-	0	-
DBCLOB	-	-	-	-	-	-	DBCLOB	-	-	-
DECIMAL	-	-	-	-	-	-	DECIMAL	-	-	-
FLOAT	4	-	-	-	-	-	REAL	-	-	-
FLOAT	8	-	-	-	-	-	DOUBLE	-	-	-
GRAPHIC	1	127	-	-	-	-	GRAPHIC	-	0	N
INTEGER	-	-	-	-	-	-	INTEGER	-	0	-
ROWID	-	-	-	-	Y	-	VARCHAR	40	-	Y
SMALLINT	-	-	-	-	-	-	SMALLINT	-	0	-
TIME	-	-	-	-	-	-	TIME	-	0	-
TIMESTAMP	-	-	-	-	-	-	TIMESTAMP	-	0	-
TIMESTMP	-	-	-	-	-	-	TIMESTAMP	-	0	-
VARCHAR	1	32672	-	-	-	-	VARCHAR	-	0	N
VARCHAR	1	32672	-	-	Y	-	VARCHAR	-	0	Y
VARG	1	16336	-	-	-	-	VARGGRAPHIC	-	0	N
VARGGRAPHIC	1	16336	-	-	-	-	VARGGRAPHIC	-	0	N

## Informix data sources

The following table lists the forward default data type mappings for Informix data sources.

Table 190. Informix forward default data type mappings (Not all columns shown)

Remote Typename	Remote Lower Len	Remote Upper Len	Remote Lower Scale	Remote Upper Scale	Remote Bit Data	Remote Data Operators	Federated Typename	Federated Length	Federated Scale	Federated Bit Data
BLOB	-	-	-	-	-	-	BLOB	2147483647	-	-
BOOLEAN	-	-	-	-	-	-	CHARACTER	1	-	-
BYTE	-	-	-	-	-	-	BLOB	2147483647	-	-
CHAR	1	254	-	-	-	-	CHARACTER	-	-	-
CHAR	255	32672	-	-	-	-	VARCHAR	-	-	-
CLOB	-	-	-	-	-	-	CLOB	2147483647	-	-
DATE	-	-	-	-	-	-	DATE	4	-	-
DATETIME	0	4	0	4	-	-	DATE	4	-	-
DATETIME	6	10	6	10	-	-	TIME	3	-	-
DATETIME	0	4	6	15	-	-	TIMESTAMP	10	-	-

## Informix data sources

Table 190. Informix forward default data type mappings (Not all columns shown) (continued)

Remote Typename	Remote Lower Len	Remote Upper Len	Remote Lower Scale	Remote Upper Scale	Remote Bit Data	Remote Data Operators	Federated Typename	Federated Length	Federated Scale	Federated Bit Data
DATETIME	6	10	11	15	-	-	TIMESTAMP	10	-	-
DECIMAL	1	31	0	31	-	-	DECIMAL	-	-	-
DECIMAL	32	130	-	-	-	-	DOUBLE	8	-	-
DECIMAL	1	32	255	255	-	-	DOUBLE	-	-	-
FLOAT	-	-	-	-	-	-	DOUBLE	8	-	-
INTEGER	-	-	-	-	-	-	INTEGER	4	-	-
INTERVAL	-	-	-	-	-	-	VARCHAR	25	-	-
INT8	-	-	-	-	-	-	BIGINT	19	0	-
LVARCHAR	1	32672	-	-	-	-	VARCHAR	-	-	-
MONEY	1	31	0	31	-	-	DECIMAL	-	-	-
MONEY	32	32	-	-	-	-	DOUBLE	8	-	-
NCHAR	1	254	-	-	-	-	CHARACTER	-	-	-
NCHAR	255	32672	-	-	-	-	VARCHAR	-	-	-
NVARCHAR	1	32672	-	-	-	-	VARCHAR	-	-	-
REAL	-	-	-	-	-	-	REAL	4	-	-
SERIAL	-	-	-	-	-	-	INTEGER	4	-	-
SERIAL8	-	-	-	-	-	-	BIGINT	-	-	-
SMALLFLOAT	-	-	-	-	-	-	REAL	4	-	-
SMALLINT	-	-	-	-	-	-	SMALLINT	2	-	-
TEXT	-	-	-	-	-	-	CLOB	2147483647	-	-
VARCHAR	1	32672	-	-	-	-	VARCHAR	-	-	-

### Notes:

- For the Informix DATETIME data type, the DB2 UNIX and Windows federated server uses the Informix high-level qualifier as the REMOTE\_LENGTH and the Informix low-level qualifier as the REMOTE\_SCALE.

The Informix qualifiers are the "TU\_" constants defined in the Informix Client SDK `datatime.h` file. The constants are:

0 = YEAR	8 = MINUTE	13 = FRACTION(3)
2 = MONTH	10 = SECOND	14 = FRACTION(4)
4 = DAY	11 = FRACTION(1)	15 = FRACTION(5)
6 = HOUR	12 = FRACTION(2)	

## Microsoft SQL Server data sources

The following table lists the forward default data type mappings for Microsoft SQL Server data sources.

Table 191. Microsoft SQL Server forward default data type mappings

Remote Typename	Remote Lower Len	Remote Upper Len	Remote Lower Scale	Remote Upper Scale	Remote Bit Data	Remote Data Operators	Federated Typename	Federated Length	Federated Scale	Federated Bit Data
bigint <sup>2</sup>	-	-	-	-	-	-	BIGINT	-	-	-
binary	1	254	-	-	-	-	CHARACTER	-	-	Y
binary	255	8000	-	-	-	-	VARCHAR	-	-	Y
bit	-	-	-	-	-	-	SMALLINT	2	-	-
char	1	254	-	-	-	-	CHAR	-	-	N
char	255	8000	-	-	-	-	VARCHAR	-	-	N



Table 191. Microsoft SQL Server forward default data type mappings (continued)

Remote Typename	Remote Lower Len	Remote Upper Len	Remote Lower Scale	Remote Upper Scale	Remote Bit Data	Remote Data Operators	Federated Typename	Federated Length	Federated Scale	Federated Bit Data
datetime	-	-	-	-	-	-	TIMESTAMP	10	-	-
decimal	1	31	0	31	-	-	DECIMAL	-	-	-
decimal	32	38	0	38	-	-	DOUBLE	-	-	-
float	-	8	-	-	-	-	DOUBLE	8	-	-
float	-	4	-	-	-	-	REAL	4	-	-
image	-	-	-	-	-	-	BLOB	2147483647	-	Y
int	-	-	-	-	-	-	INTEGER	4	-	-
money	-	-	-	-	-	-	DECIMAL	19	4	-
nchar	1	127	-	-	-	-	CHAR	-	-	N
nchar	128	4000	-	-	-	-	VARCHAR	-	-	N
numeric	1	31	0	31	-	-	DECIMAL	-	-	-
numeric	32	38	0	38	-	-	DOUBLE	8	-	-
ntext	-	-	-	-	-	-	CLOB	2147483647	-	Y
nvarchar	1	4000	-	-	-	-	VARCHAR	-	-	N
real	-	-	-	-	-	-	REAL	4	-	-
smallint	-	-	-	-	-	-	SMALLINT	2	-	-
smalldatetime	-	-	-	-	-	-	TIMESTAMP	10	-	-
smallmoney	-	-	-	-	-	-	DECIMAL	10	4	-
SQL_BIGINT	-	-	-	-	-	-	BIGINT	-	-	-
SQL_BINARY	1	254	-	-	-	-	CHARACTER	-	-	Y
SQL_BINARY	255	8000	-	-	-	-	VARCHAR	-	-	Y
SQL_BIT	-	-	-	-	-	-	SMALLINT	2	-	-
SQL_CHAR	1	254	-	-	-	-	CHAR	-	-	N
SQL_CHAR	255	8000	-	-	-	-	VARCHAR	-	-	N
SQL_DATE	-	-	-	-	-	-	DATE	4	-	-
SQL_DECIMAL	1	31	0	31	-	-	DECIMAL	-	-	-
SQL_DECIMAL	32	38	0	38	-	-	DOUBLE	8	-	-
SQL_DOUBLE	-	-	-	-	-	-	DOUBLE	8	-	-
SQL_FLOAT	-	-	-	-	-	-	DOUBLE	8	-	-
SQL_GUID	-	-	-	-	-	-	VARCHAR	-	-	Y
SQL_INTEGER	-	-	-	-	-	-	INTEGER	4	-	-
SQL_LONGVARCHAR	-	-	-	-	-	-	CLOB	2147483647	-	N
SQL_LONGVARBINARY	-	-	-	-	-	-	BLOB	-	-	Y
SQL_NUMERIC	1	31	0	31	-	-	DECIMAL	-	-	-
SQL_NUMERIC	32	38	0	38	-	-	DOUBLE	8	-	-
SQL_REAL	-	-	-	-	-	-	REAL	8	-	-
SQL_SMALLINT	-	-	-	-	-	-	SMALLINT	2	-	-
SQL_TIME	-	-	-	-	-	-	TIME	3	-	-
SQL_TIMESTAMP	-	-	-	-	-	-	TIMESTAMP	10	-	-
SQL_TINYINT	-	-	-	-	-	-	SMALLINT	2	-	-
SQL_VARBINARY	1	8000	-	-	-	-	VARCHAR	-	-	Y
SQL_VARCHAR	1	8000	-	-	-	-	VARCHAR	-	-	N

## Microsoft SQL Server data sources

Table 191. Microsoft SQL Server forward default data type mappings (continued)

Remote Typename	Remote Lower Len	Remote Upper Len	Remote Lower Scale	Remote Upper Scale	Remote Bit Data	Remote Data Operators	Federated Typename	Federated Length	Federated Scale	Federated Bit Data
SQL_WCHAR	1	254	-	-	-	-	CHARACTER	-	-	N
SQL_WCHAR	255	8800	-	-	-	-	VARCHAR	-	-	N
SQL_WLONGVARCHAR		1073741823	-	-	-	-	CLOB	2147483647	-	N
SQL_WVARCHAR	1	16336	-	-	-	-	VARCHAR	-	-	N
text	-	-	-	-	-	-	CLOB	-	-	N
timestamp	-	-	-	-	-	-	VARCHAR	8	-	Y
tinyint	-	-	-	-	-	-	SMALLINT	2	-	-
uniqueidentifier	1	4000	-	-	Y	-	VARCHAR	16	-	Y
varbinary	1	8000	-	-	-	-	VARCHAR	-	-	Y
varchar	1	8000	-	-	-	-	VARCHAR	-	-	N

### Note:

1. This type mapping is valid only with Microsoft SQL Server Version 2000.

## ODBC data sources

The following table lists the forward default data type mappings for ODBC data sources.

Table 192. ODBC forward default data type mappings (Not all columns shown)

Remote Typename	Remote Lower Len	Remote Upper Len	Remote Lower Scale	Remote Upper Scale	Remote Bit Data	Remote Data Operators	Federated Typename	Federated Length	Federated Scale	Federated Bit Data
SQL_BIGINT	-	-	-	-	-	-	BIGINT	8	-	-
SQL_BINARY	1	254	-	-	-	-	CHARACTER	-	-	Y
SQL_BINARY	255	32672	-	-	-	-	VARCHAR	-	-	Y
SQL_BIT	-	-	-	-	-	-	SMALLINT	2	-	-
SQL_CHAR	1	254	-	-	-	-	CHAR	-	-	N
SQL_CHAR	255	32672	-	-	-	-	VARCHAR	-	-	N
SQL_DECIMAL	1	31	0	31	-	-	DECIMAL	-	-	-
SQL_DECIMAL	32	38	0	38	-	-	DOUBLE	8	-	-
SQL_DOUBLE	-	-	-	-	-	-	DOUBLE	8	-	-
SQL_FLOAT	-	8	-	-	-	-	FLOAT	8	-	-
SQL_FLOAT	-	4	-	-	-	-	FLOAT	4	-	-
SQL_INTEGER	-	-	-	-	-	-	INTEGER	4	-	-
SQL_LONGVARCHAR	-	-	-	-	-	-	CLOB	2147483647	-	N
SQL_LONGVARBINARY	-	-	-	-	-	-	BLOB	2147483647	-	Y
SQL_NUMERIC	1	31	0	31	-	-	DECIMAL	-	-	-
SQL_NUMERIC	32	32	0	31	-	-	DOUBLE	8	-	-
SQL_REAL	-	-	-	-	-	-	REAL	4	-	-
SQL_SMALLINT	-	-	-	-	-	-	SMALLINT	2	-	-
SQL_TYPE_DATE	-	-	-	-	-	-	DATE	4	-	-
SQL_TYPE_TIME	-	-	-	-	-	-	TIME	3	-	-

Table 192. ODBC forward default data type mappings (Not all columns shown) (continued)

Remote Typename	Remote Lower Len	Remote Upper Len	Remote Lower Scale	Remote Upper Scale	Remote Bit Data	Remote Data Operators	Federated Typename	Federated Length	Federated Scale	Federated Bit Data
SQL_TYPE_TIMESTAMP	-	-	-	-	-	-	TIMESTAMP	10	-	-
SQL_TINYINT	-	-	-	-	-	-	SMALLINT	2	-	-
SQL_VARBINARY	1	32672	-	-	-	-	VARCHAR	-	-	Y
SQL_VARCHAR	1	32672	-	-	-	-	VARCHAR	-	-	N
SQL_WCHAR	1	127	-	-	-	-	CHAR	-	-	N
SQL_WCHAR	128	16336	-	-	-	-	VARCHAR	-	-	N
SQL_WVARCHAR	1	16336	-	-	-	-	VARCHAR	-	-	N
SQL_WLONGVARCHAR	-	1073741823	-	-	-	-	CLOB	2147483647	-	N

## Oracle NET8 data sources

The following table lists the forward default data type mappings for Oracle NET8 data sources.

Table 193. Oracle NET8 forward default data type mappings

Remote Typename	Remote Lower Len	Remote Upper Len	Remote Lower Scale	Remote Upper Scale	Remote Bit Data	Remote Data Operators	Federated Typename	Federated Length	Federated Scale	Federated Bit Data
BLOB	0	0	0	0	-	\0	BLOB	2147483647	0	Y
CHAR	1	254	0	0	-	\0	CHAR	0	0	N
CHAR	255	2000	0	0	-	\0	VARCHAR	0	0	N
CLOB	0	0	0	0	-	\0	CLOB	2147483647	0	N
DATE	0	0	0	0	-	\0	TIMESTAMP	0	0	N
FLOAT	1	126	0	0	-	\0	DOUBLE	0	0	N
LONG	0	0	0	0	-	\0	CLOB	2147483647	0	N
LONG RAW	0	0	0	0	-	\0	BLOB	2147483647	0	Y
NUMBER	10	18	0	0	-	\0	BIGINT	0	0	N
NUMBER	1	38	-84	127	-	\0	DOUBLE	0	0	N
NUMBER	1	31	0	31	-	>=	DECIMAL	0	0	N
NUMBER	1	4	0	0	-	\0	SMALLINT	0	0	N
NUMBER	5	9	0	0	-	\0	INTEGER	0	0	N
NUMBER	-	10	0	0	-	\0	DECIMAL	0	0	N
RAW	1	2000	0	0	-	\0	VARCHAR	0	0	Y
ROWID	0	0	0	NULL	-	\0	CHAR	18	0	N
TIMESTAMP <sup>1</sup>	-	-	-	-	-	-	TIMESTAMP	10	-	-
VARCHAR2	1	4000	0	0	-	\0	VARCHAR	0	0	N

**Note:**

1. This type mapping is valid only for Oracle 9i (or later) client and server configurations.

## Sybase data sources

The following table lists the forward default data type mappings for Sybase data sources.

## Sybase data sources

Table 194. Sybase CTLIB forward default data type mappings

Remote Typename	Remote Lower Len	Remote Upper Len	Remote Lower Scale	Remote Upper Scale	Remote Bit Data	Remote Data Operators	Federated Typename	Federated Length	Federated Scale	Federated Bit Data
binary	1	254	-	-	-	-	CHAR	-	-	Y
binary	255	32672	-	-	-	-	VARCHAR	-	-	Y
bit	-	-	-	-	-	-	SMALLINT	-	-	-
char	1	254	-	-	-	-	CHAR	-	-	N
char	255	32672	-	-	-	-	VARCHAR	-	-	N
char null (see varchar)										
datetime	-	-	-	-	-	-	TIMESTAMP	-	-	-
datetimen	-	-	-	-	-	-	TIMESTAMP	-	-	-
decimal	1	31	0	31	-	-	DECIMAL	-	-	-
decimal	32	38	0	38	-	-	DOUBLE	-	-	-
decimaln	1	31	0	31	-	-	DECIMAL	-	-	-
decimaln	32	38	0	38	-	-	DOUBLE	-	-	-
float	-	4	-	-	-	-	REAL	-	-	-
float	-	8	-	-	-	-	DOUBLE	-	-	-
floatn	-	4	-	-	-	-	REAL	-	-	-
floatn	-	8	-	-	-	-	DOUBLE	-	-	-
image	-	-	-	-	-	-	BLOB	-	-	-
int	-	-	-	-	-	-	INTEGER	-	-	-
intn	-	-	-	-	-	-	INTEGER	-	-	-
money	-	-	-	-	-	-	DECIMAL	19	4	-
moneyn	-	-	-	-	-	-	DECIMAL	19	4	-
nchar	1	254	-	-	-	-	CHAR	-	-	N
nchar	255	32672	-	-	-	-	VARCHAR	-	-	N
nchar null (see nvarchar)										
numeric	1	31	0	31	-	-	DECIMAL	-	-	-
numeric	32	38	0	38	-	-	DOUBLE	-	-	-
numericn	1	31	0	31	-	-	DECIMAL	-	-	-
numericn	32	38	0	38	-	-	DOUBLE	-	-	-
nvarchar	1	32672	-	-	-	-	VARCHAR	-	-	N
real	-	-	-	-	-	-	REAL	-	-	-
smalldatetime	-	-	-	-	-	-	TIMESTAMP	-	-	-
smallint	-	-	-	-	-	-	SMALLINT	-	-	-
smallmoney	-	-	-	-	-	-	DECIMAL	10	4	-
sysname	-	-	-	-	-	-	VARCHAR	30	-	N
text	-	-	-	-	-	-	CLOB	-	-	-
timestamp	-	-	-	-	-	-	VARCHAR	8	-	Y
tinyint	-	-	-	-	-	-	SMALLINT	-	-	-
unichar <sup>1</sup>	1	254	-	-	-	-	CHAR	-	-	N
unichar <sup>1</sup>	255	32672	-	-	-	-	VARCHAR	-	-	N
unichar null (see univarchar)										
univarchar <sup>1</sup>	1	32672	-	-	-	-	VARCHAR	-	-	N

Table 194. Sybase CTLIB forward default data type mappings (continued)

Remote Typename	Remote Lower Len	Remote Upper Len	Remote Lower Scale	Remote Upper Scale	Remote Bit Data	Remote Data Operators	Federated Typename	Federated Length	Federated Scale	Federated Bit Data
varbinary	1	32672	-	-	-	-	VARCHAR	-	-	Y
varchar	1	32672	-	-	-	-	VARCHAR	-	-	N

**Note:**

1. Valid for non-Unicode federated databases.

## Teradata data sources

The following table lists the forward default data type mappings for Teradata data sources.

Table 195. Teradata forward default data type mappings (Not all columns shown)

Remote Typename	Remote Lower Len	Remote Upper Len	Remote Lower Scale	Remote Upper Scale	Remote Bit Data	Remote Data Operators	Federated Typename	Federated Length	Federated Scale	Federated Bit Data
BYTE	1	254	-	-	-	-	CHAR	-	-	Y
BYTE	255	32672	-	-	-	-	VARCHAR	-	-	Y
BYTE	32673	64000	-	-	-	-	BLOB	-	-	-
BYTEINT	-	-	-	-	-	-	SMALLINT	-	-	-
CHAR	1	254	-	-	-	-	CHARACTER	-	-	-
CHAR	255	32672	-	-	-	-	VARCHAR	-	-	-
CHAR	32673	64000	-	-	-	-	CLOB	-	-	-
DATE	-	-	-	-	-	-	DATE	-	-	-
DECIMAL	1	18	0	18	-	-	DECIMAL	-	-	-
DOUBLE PRECISION	-	-	-	-	-	-	DOUBLE	-	-	-
FLOAT	-	-	-	-	-	-	DOUBLE	-	-	-
GRAPHIC	1	127	-	-	-	-	GRAPHIC	-	-	-
GRAPHIC	128	16336	-	-	-	-	VARGRAPHIC	-	-	-
GRAPHIC	16337	32000	-	-	-	-	DBCLOB	-	-	-
INTEGER	-	-	-	-	-	-	INTEGER	-	-	-
INTERVAL	-	-	-	-	-	-	CHAR	-	-	-
NUMERIC	1	18	0	18	-	-	DECIMAL	-	-	-
REAL	-	-	-	-	-	-	DOUBLE	-	-	-
SMALLINT	-	-	-	-	-	-	SMALLINT	-	-	-
TIME	0	21	0	21	-	-	TIME	-	-	-
TIMESTAMP	-	-	-	-	-	-	TIMESTAMP	-	-	-
VARBYTE	1	32762	-	-	-	-	VARCHAR	-	-	Y
VARBYTE	32763	64000	-	-	-	-	BLOB	-	-	-
VARCHAR	1	32672	-	-	-	-	VARCHAR	-	-	-
VARCHAR	32673	64000	-	-	-	-	CLOB	-	-	-
VARGRAPHIC	1	16336	-	-	-	-	VARGRAPHIC	-	-	-
VARGRAPHIC	16337	32000	-	-	-	-	DBCLOB	-	-	-

## Default reverse data type mappings

For most data sources, the default type mappings are in the wrappers.

## Default reverse data type mappings

The two kinds of mappings between data source data types and federated database data types are forward type mappings and reverse type mappings. In a forward type mapping, the mapping is from a remote type to a comparable local type. The other type of mapping is a reverse type mapping, which is used with transparent DDL to create or modify remote tables.

The default type mappings for DB2 family data sources are in the DRDA wrapper. The default type mappings for Informix are in the INFORMIX wrapper, and so forth.

When you define a remote table or view to the federated database, the definition includes a reverse type mapping. The mapping is from a local federated database data type for each column, and the corresponding remote data type. For example, there is a default reverse type mapping in which the local type REAL points to the Informix type SMALLFLOAT.

Federated databases do not support mappings for LONG VARCHAR, LONG VARGRAPHIC, and user-defined types.

When you use the CREATE TABLE statement to create a remote table, you specify the local data types you want to include in the remote table. These default reverse type mappings will assign corresponding remote types to these columns. For example, suppose that you use the CREATE TABLE statement to define an Informix table with a column C2. You specify BIGINT as the data type for C2 in the statement. The default reverse type mapping of BIGINT depends on which version of Informix you are creating the table on. The mapping for C2 in the Informix table will be to DECIMAL in Informix Version 8 and to INT8 in Informix Version 9.

You can override a default reverse type mapping, or create a new reverse type mapping with the CREATE TYPE MAPPING statement.

The following tables show the default reverse mappings between federated database local data types and remote data source data types.

These mappings are valid with all the supported versions, unless otherwise noted.

## DB2 Database for Linux, UNIX, and Windows data sources

The following table lists the reverse default data type mappings for DB2 Database for Linux, UNIX, and Windows data sources.

*Table 196. DB2 Database for Linux, UNIX, and Windows reverse default data type mappings (Not all columns shown)*

Federated Typename	Federated Lower Len	Federated Upper Len	Federated Lower Scale	Federated Upper Scale	Federated Bit Data	Federated Data Operators	Remote Typename	Remote Length	Remote Scale	Federated Bit Data
BIGINT	-	8	-	-	-	-	BIGINT	-	-	-
BLOB	-	-	-	-	-	-	BLOB	-	-	-
CHARACTER	-	-	-	-	-	-	CHAR	-	-	N
CHARACTER	-	-	-	-	Y	-	CHAR	-	-	Y
CLOB	-	-	-	-	-	-	CLOB	-	-	-
DATE	-	4	-	-	-	-	DATE	-	-	-
DBCLOB	-	-	-	-	-	-	DBCLOB	-	-	-
DECIMAL	-	-	-	-	-	-	DECIMAL	-	-	-
DOUBLE	-	8	-	-	-	-	DOUBLE	-	-	-

## DB2 Database for Linux, UNIX, and Windows data sources

Table 196. DB2 Database for Linux, UNIX, and Windows reverse default data type mappings (Not all columns shown) (continued)

Federated Typename	Federated Lower Len	Federated Upper Len	Federated Lower Scale	Federated Upper Scale	Federated Bit Data	Federated Data Operators	Remote Typename	Remote Length	Remote Scale	Federated Bit Data
FLOAT	-	8	-	-	-	-	DOUBLE	-	-	-
GRAPHIC	-	-	-	-	-	-	GRAPHIC	-	-	N
INTEGER	-	4	-	-	-	-	INTEGER	-	-	-
REAL	-	-	-	-	-	-	REAL	-	-	-
SMALLINT	-	2	-	-	-	-	SMALLINT	-	-	-
TIME	-	3	-	-	-	-	TIME	-	-	-
TIMESTAMP	-	10	-	-	-	-	TIMESTAMP	-	-	-
VARCHAR	-	-	-	-	-	-	VARCHAR	-	-	N
VARCHAR	-	-	-	-	Y	-	VARCHAR	-	-	Y
VARGRAPH	-	-	-	-	-	-	VARGRAPHIC	-	-	N
VARGRAPHIC	-	-	-	-	-	-	VARGRAPHIC	-	-	-

## DB2 for System i data sources

The following table lists the reverse default data type mappings for DB2 for System i data sources.

Table 197. DB2 for System i reverse default data type mappings (Not all columns shown)

Federated Typename	Federated Lower Len	Federated Upper Len	Federated Lower Scale	Federated Upper Scale	Federated Bit Data	Federated Data Operations	Remote Typename	Remote Length	Remote Scale	Remote Bit Data
BLOB	-	-	-	-	-	-	BLOB	-	-	-
CHARACTER	-	-	-	-	-	-	CHARACTER	-	-	N
CHARACTER	-	-	-	-	Y	-	CHARACTER	-	-	Y
CLOB	-	-	-	-	-	-	CLOB	-	-	-
DATE	-	4	-	-	-	-	DATE	-	-	-
DBCLOB	-	-	-	-	-	-	DBCLOB	-	-	-
DECIMAL	-	-	-	-	-	-	NUMERIC	-	-	-
DECIMAL	-	-	-	-	-	-	DECIMAL	-	-	-
DOUBLE	-	8	-	-	-	-	FLOAT	-	-	-
GRAPHIC	-	-	-	-	-	-	GRAPHIC	-	-	N
INTEGER	-	4	-	-	-	-	INTEGER	-	-	-
REAL	-	4	-	-	-	-	FLOAT	-	-	-
SMALLINT	-	2	-	-	-	-	SMALLINT	-	-	-
TIME	-	3	-	-	-	-	TIME	-	-	-
TIMESTAMP	-	10	-	-	-	-	TIMESTAMP	-	-	-
VARCHAR	-	-	-	-	-	-	VARCHAR	-	-	N
VARCHAR	-	-	-	-	Y	-	VARCHAR	-	-	Y
VARGRAPHIC	-	-	-	-	-	-	VARG	-	-	N

## DB2 for VM and VSE data sources

The following table lists the reverse default data type mappings for DB2 for VM and VSE data sources.

## DB2 for VM and VSE data sources

Table 198. DB2 for VM and VSE reverse default data type mappings (Not all columns shown)

Federated Typename	Federated Lower Len	Federated Upper Len	Federated Lower Scale	Federated Upper Scale	Federated Bit Data	Federated Data Operators	Remote Typename	Remote Length	Remote Scale	Remote Bit Data
BLOB	-	-	-	-	-	-	BLOB	-	-	-
CHARACTER	-	-	-	-	-	-	CHAR	-	-	-
CHARACTER	-	-	-	-	Y	-	CHAR	-	-	Y
CLOB	-	-	-	-	-	-	CLOB	-	-	-
DATE	-	4	-	-	-	-	DATE	-	-	-
DBCLOB	-	-	-	-	-	-	DBCLOB	-	-	-
DECIMAL	-	-	-	-	-	-	DECIMAL	-	-	-
DOUBLE	-	8	-	-	-	-	FLOAT	-	-	-
GRAPHIC	-	-	-	-	-	-	GRAPHIC	-	-	N
INTEGER	-	4	-	-	-	-	INTEGER	-	-	-
REAL	-	4	-	-	-	-	REAL	-	-	-
SMALLINT	-	2	-	-	-	-	SMALLINT	-	-	-
TIME	-	3	-	-	-	-	TIME	-	-	-
TIMESTAMP	-	10	-	-	-	-	TIMESTAMP	-	-	-
VARCHAR	-	-	-	-	-	-	VARCHAR	-	-	-
VARCHAR	-	-	-	-	Y	-	VARCHAR	-	-	Y
VARGRAPH	-	-	-	-	-	-	VARGRAPH	-	-	N

## DB2 for z/OS data sources

The following table lists the reverse default data type mappings for DB2 for z/OS data sources.

Table 199. DB2 for z/OS reverse default data type mappings (Not all columns shown)

Federated Typename	Federated Lower Len	Federated Upper Len	Federated Lower Scale	Federated Upper Scale	Federated Bit Data	Federated Data Operators	Remote Typename	Remote Length	Remote Scale	Remote Bit Data
BLOB	-	-	-	-	-	-	BLOB	-	-	-
CHARACTER	-	-	-	-	-	-	CHAR	-	-	N
CHARACTER	-	-	-	-	Y	-	CHAR	-	-	Y
CLOB	-	-	-	-	-	-	CLOB	-	-	-
DATE	-	4	-	-	-	-	DATE	-	-	-
DBCLOB	-	-	-	-	-	-	DBCLOB	-	-	-
DECIMAL	-	-	-	-	-	-	DECIMAL	-	-	-
DOUBLE	-	8	-	-	-	-	DOUBLE	-	-	-
FLOAT	-	8	-	-	-	-	DOUBLE	-	-	-
GRAPHIC	-	-	-	-	-	-	GRAPHIC	-	-	N
INTEGER	-	4	-	-	-	-	INTEGER	-	-	-
REAL	-	4	-	-	-	-	REAL	-	-	-
SMALLINT	-	2	-	-	-	-	SMALLINT	-	-	-
TIME	-	3	-	-	-	-	TIME	-	-	-
TIMESTAMP	-	10	-	-	-	-	TIMESTAMP	-	-	-
VARCHAR	-	-	-	-	-	-	VARCHAR	-	-	N
VARCHAR	-	-	-	-	Y	-	VARCHAR	-	-	Y
VARGRAPHIC	-	-	-	-	-	-	VARGRAPHIC	-	-	N



## Informix data sources

The following table lists the reverse default data type mappings for Informix data sources.

Table 200. Informix reverse default data type mappings

Federated Typename	Federated Lower Len	Federated Upper Len	Federated Lower Scale	Federated Upper Scale	Federated Bit Data	Federated Data Operators	Remote Typename	Remote Length	Remote Scale	Remote Bit Data
BIGINT <sup>1</sup>	-	-	-	-	-	-	DECIMAL	19	-	-
BIGINT <sup>2</sup>	-	-	-	-	-	-	INT8	-	-	-
BLOB	1	2147483647	-	-	-	-	BYTE	-	-	-
CHARACTER	-	-	-	-	N	-	CHAR	-	-	-
CHARACTER	-	-	-	-	Y	-	BYTE	-	-	-
CLOB	1	2147483647	-	-	-	-	TEXT	-	-	-
DATE	-	4	-	-	-	-	DATE	-	-	-
DECIMAL	-	-	-	-	-	-	DECIMAL	-	-	-
DOUBLE	-	8	-	-	-	-	FLOAT	-	-	-
INTEGER	-	4	-	-	-	-	INTEGER	-	-	-
REAL	-	4	-	-	-	-	SMALLFLOAT	-	-	-
SMALLINT	-	2	-	-	-	-	SMALLINT	-	-	-
TIME	-	3	-	-	-	-	DATETIME	6	10	-
TIMESTAMP	-	10	-	-	-	-	DATETIME	0	15	-
VARCHAR	1	254	-	-	N	-	VARCHAR	-	-	-
VARCHAR <sup>1</sup>	255	32672	-	-	N	-	TEXT	-	-	-
VARCHAR	-	-	-	-	Y	-	BYTE	-	-	-
VARCHAR <sup>2</sup>	255	2048	-	-	N	-	LVARCHAR	-	-	-
VARCHAR <sup>2</sup>	2049	32672	-	-	N	-	TEXT	-	-	-

**Note:**

1. This type mapping is valid only with Informix server Version 8 (or lower).
2. This type mapping is valid only with Informix server Version 9 (or higher).

For the Informix DATETIME data type, the federated server uses the Informix high-level qualifier as the REMOTE\_LENGTH and the Informix low-level qualifier as the REMOTE\_SCALE.

The Informix qualifiers are the "TU\_" constants defined in the Informix Client SDK `datatime.h` file. The constants are:

0 = YEAR	8 = MINUTE	13 = FRACTION(3)
2 = MONTH	10 = SECOND	14 = FRACTION(4)
4 = DAY	11 = FRACTION(1)	15 = FRACTION(5)
6 = HOUR	12 = FRACTION(2)	

## Microsoft SQL Server data sources

The following table lists the reverse default data type mappings for Microsoft SQL Server data sources.

Table 201. Microsoft SQL Server reverse default data type mappings (Not all columns shown)

Federated Typename	Federated Lower Len	Federated Upper Len	Federated Lower Scale	Federated Upper Scale	Federated Bit Data	Federated Data Operators	Remote Typename	Remote Length	Remote Scale	Remote Bit Data
BIGINT <sup>1</sup>	-	-	-	-	-	-	bigint	-	-	-
BLOB	-	-	-	-	-	-	image	-	-	-

## Microsoft SQL Server data sources

Table 201. Microsoft SQL Server reverse default data type mappings (Not all columns shown) (continued)

Federated Typename	Federated Lower Len	Federated Upper Len	Federated Lower Scale	Federated Upper Scale	Federated Bit Data	Federated Data Operators	Remote Typename	Remote Length	Remote Scale	Remote Bit Data
CHARACTER	-	-	-	-	Y	-	binary	-	-	-
CHARACTER	-	-	-	-	N	-	char	-	-	-
CLOB	-	-	-	-	-	-	text	-	-	-
DATE	-	4	-	-	-	-	datetime	-	-	-
DECIMAL	-	-	-	-	-	-	decimal	-	-	-
DOUBLE	-	8	-	-	-	-	float	-	-	-
INTEGER	-	-	-	-	-	-	int	-	-	-
SMALLINT	-	-	-	-	-	-	smallint	-	-	-
REAL	-	4	-	-	-	-	real	-	-	-
TIME	-	3	-	-	-	-	datetime	-	-	-
TIMESTAMP	-	10	-	-	-	-	datetime	-	-	-
VARCHAR	1	8000	-	-	N	-	varchar	-	-	-
VARCHAR	8001	32672	-	-	N	-	text	-	-	-
VARCHAR	1	8000	-	-	Y	-	varbinary	-	-	-
VARCHAR	8001	32672	-	-	Y	-	image	-	-	-

**Note:**

1. This type mapping is valid only with Microsoft SQL Server Version 2000.

## Oracle NET8 data sources

The following table lists the reverse default data type mappings for Oracle NET8 data sources.

Table 202. Oracle NET8 reverse default data type mappings

Federated Typename	Federated Lower Len	Federated Upper Len	Federated Lower Scale	Federated Upper Scale	Federated Bit Data	Federated Data Operators	Remote Typename	Remote Length	Remote Scale	Remote Bit Data
BIGINT	0	8	0	0	N	\0	NUMBER	19	0	N
BLOB	0	2147483647	0	0	Y	\0	BLOB	0	0	Y
CHARACTER	1	254	0	0	N	\0	CHAR	0	0	N
CHARACTER	1	254	0	0	Y	\0	RAW	0	0	Y
CLOB	0	2147483647	0	0	N	\0	CLOB	0	0	N
DATE	0	4	0	0	N	\0	DATE	0	0	N
DECIMAL	0	0	0	0	N	\0	NUMBER	0	0	N
DOUBLE	0	8	0	0	N	\0	FLOAT	126	0	N
FLOAT	0	8	0	0	N	\0	FLOAT	126	0	N
INTEGER	0	4	0	0	N	\0	NUMBER	10	0	N
REAL	0	4	0	0	N	\0	FLOAT	63	0	N
SMALLINT	0	2	0	0	N	\0	NUMBER	5	0	N
TIME	0	3	0	0	N	\0	DATE	0	0	N
TIMESTAMP <sup>1</sup>	0	10	0	0	N	\0	DATE	0	0	N
TIMESTAMP <sup>2</sup>	0	10	0	0	N	\0	TIMESTAMP	6	0	N
VARCHAR	1	4000	0	0	N	\0	VARCHAR2	0	0	N
VARCHAR	1	2000	0	0	Y	\0	RAW	0	0	Y

Table 202. Oracle NET8 reverse default data type mappings (continued)

Federated Typename	Federated Lower Len	Federated Upper Len	Federated Lower Scale	Federated Upper Scale	Federated Bit Data	Federated Data Operators	Remote Typename	Remote Length	Remote Scale	Remote Bit Data
-----------------------	---------------------------	---------------------------	-----------------------------	-----------------------------	-----------------------	--------------------------------	--------------------	------------------	-----------------	--------------------

**Note:**

1. This type mapping is valid only with Oracle Version 8.
2. This type mapping is valid only with Oracle Version 9 and Version 10.

## Sybase data sources

The following table lists the reverse default data type mappings for Sybase data sources.

Table 203. Sybase CTLIB default reverse data type mappings

Federated Typename	Federated Lower Len	Federated Upper Len	Federated Lower Scale	Federated Upper Scale	Federated Bit Data	Federated Data Operators	Remote Typename	Remote Length	Remote Scale	Remote Bit Data
BIGINT	-	-	-	-	-	-	decimal	19	0	-
BLOB	-	-	-	-	-	-	image	-	-	-
CHARACTER	-	-	-	-	N	-	char	-	-	-
CHARACTER	-	-	-	-	Y	-	binary	-	-	-
CLOB	-	-	-	-	-	-	text	-	-	-
DATE	-	-	-	-	-	-	datetime	-	-	-
DECIMAL	-	-	-	-	-	-	decimal	-	-	-
DOUBLE	-	-	-	-	-	-	float	-	-	-
INTEGER	-	-	-	-	-	-	integer	-	-	-
REAL	-	-	-	-	-	-	real	-	-	-
SMALLINT	-	-	-	-	-	-	smallint	-	-	-
TIME	-	-	-	-	-	-	datetime	-	-	-
TIMESTAMP	-	-	-	-	-	-	datetime	-	-	-
VARCHAR <sup>1</sup>	1	255	-	-	N	-	varchar	-	-	-
VARCHAR <sup>1</sup>	256	32672	-	-	N	-	text	-	-	-
VARCHAR <sup>2</sup>	1	16384	-	-	N	-	varchar	-	-	-
VARCHAR <sup>2</sup>	16385	32672	-	-	N	-	text	-	-	-
VARCHAR <sup>1</sup>	1	255	-	-	Y	-	varbinary	-	-	-
VARCHAR <sup>1</sup>	256	32672	-	-	Y	-	image	-	-	-
VARCHAR <sup>2</sup>	1	16384	-	-	Y	-	varbinary	-	-	-
VARCHAR <sup>2</sup>	16385	32672	-	-	Y	-	image	-	-	-

**Note:**

1. This type mapping is valid only for CTLIB with Sybase server version 12.0 (or earlier).
2. This type mapping is valid only for CTLIB with Sybase server version 12.5 (or later).

## Teradata data sources

The following table lists the reverse default data type mappings for Teradata data sources.

## Teradata data sources

Table 204. Teradata reverse default data type mappings (Not all columns shown)

Federated Typename	Federated Lower Len	Federated Upper Len	Federated Lower Scale	Federated Upper Scale	Federated Bit Data	Federated Data Operators	Remote Typename	Remote Length	Remote Scale	Remote Bit Data
BLOB <sup>1</sup>	1	64000	-	-	-	-	VARBYTE	-	-	-
CHARACTER	-	-	-	-	-	-	CHARACTER	-	-	-
CHARACTER	-	-	-	-	Y	-	BYTE	-	-	-
CLOB <sup>2</sup>	1	64000	-	-	-	-	VARCHAR	-	-	-
DATE	-	-	-	-	-	-	DATE	-	-	-
DBCLOB <sup>3</sup>	1	64000	-	-	-	-	VARGRAPHIC	-	-	-
DECIMAL	1	18	0	18	-	-	DECIMAL	-	-	-
DECIMAL	19	31	0	31	-	-	FLOAT	8	-	-
DOUBLE	-	-	-	-	-	-	FLOAT	-	-	-
GRAPHIC	-	-	-	-	-	-	GRAPHIC	-	-	-
INTEGER	-	-	-	-	-	-	INTEGER	-	-	-
REAL	-	-	-	-	-	-	FLOAT	8	-	-
SMALLINT	-	-	-	-	-	-	SMALLINT	-	-	-
TIME	-	-	-	-	-	-	TIME	15	-	-
TIMESTAMP	-	-	-	-	-	-	TIMESTAMP	26	-	-
VARCHAR	-	-	-	-	-	-	VARCHAR	-	-	-
VARCHAR	-	-	-	-	Y	-	VARBYTE	-	-	-
VARGRAPHIC	-	-	-	-	-	-	VARGRAPHIC	-	-	-

**Note:**

1. The Teradata VARBYTE data type can contain only the specified length (1 to 64000) of a BLOB data type.
2. The Teradata VARCHAR data type can contain only the specified length (1 to 64000) of a CLOB data type.
3. The Teradata VARGRAPHIC data type can contain only the specified length (1 to 32000) of a DBCLOB data type.

---

## Appendix F. The SAMPLE database

Sample database can be used for various purposes like testing your applications, trying different features of DB2 and so on. Most of the sample application programs under DB2PATH/sql/lib/samples use sample database for demonstrating various features of DB2 that makes it easy to understand the technology.

Once the sample database is created, you will notice :

- Organizational schema for non-XML data and
- Purchase order schema for XML data are created.

The data and database objects under these schemas are created using real time environment on a small scale.

Following is a description of each of the tables in the SAMPLE database. Initial data values for each table are given; a dash (-) indicates a NULL value.

### ACT table

Name:	ACTNO	ACTKWD	ACTDESC
Type:	SMALLINT	CHAR(6)	VARCHAR(20)
Values:	10	MANAGE	MANAGE/ADVISE
	20	ECOST	ESTIMATE COST
	30	DEFINE	DEFINE SPECS
	40	LEADPR	LEAD PROGRAM/DESIGN
	50	SPECS	WRITE SPECS
	60	LOGIC	DESCRIBE LOGIC
	70	CODE	CODE PROGRAMS
	80	TEST	TEST PROGRAMS
	90	ADMQS	ADM QUERY SYSTEM
	100	TEACH	TEACH CLASSES
	110	COURSE	DEVELOP COURSES
	120	STAFF	PERS AND STAFFING
	130	OPERAT	OPER COMPUTER SYS
	140	MAINT	MAINT SOFTWARE SYS
	150	ADMSYS	ADM OPERATING SYS
	160	ADMDB	ADM DATA BASES
	170	ADMDC	ADM DATA COMM
	180	DOC	DOCUMENT

### ADEFUSER table

Name:	WORKDEPT	NO_OF_EMPLOYEES
Type:	CHAR(3)	INTEGER
Values:	A00	5

## The SAMPLE database

Name:	WORKDEPT	NO_OF_EMPLOYEES
	B01	1
	C01	4
	D11	11
	D21	7
	E01	1
	E11	7
	E21	6

### 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
Values:	042:BF	4	12:10:00	14:00:00
	553:MJA	1	10:30:00	11:00:00
	543:CWM	3	09:10:00	10:30:00
	778:RES	2	12:10:00	14:00:00
	044:HD	3	17:12:30	18:00:00

### DEPT table

Name:	DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
Type:	CHAR(3)	VARCHAR(36)	CHAR(6)	CHAR(3)	CHAR(16)
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	
	F22	BRANCH OFFICE F2		E01	
	G22	BRANCH OFFICE G2		E01	
	H22	BRANCH OFFICE H2		E01	
	I22	BRANCH OFFICE I2		E01	
	J22	BRANCH OFFICE J2		E01	

**DEPARTMENT table**

<b>Name:</b>	<b>DEPTNO</b>	<b>DEPTNAME</b>	<b>MGRNO</b>	<b>ADMRDEPT</b>	<b>LOCATION</b>
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	
	F22	BRANCH OFFICE F2		E01	
	G22	BRANCH OFFICE G2		E01	
	H22	BRANCH OFFICE H2		E01	
	I22	BRANCH OFFICE I2		E01	
	J22	BRANCH OFFICE J2		E01	

**EMPLOYEE and EMP tables**

These two tables have identical content.

<b>Names:</b>	<b>EMPNO</b>	<b>FIRSTNME</b>	<b>MIDINIT</b>	<b>LASTNAME</b>	<b>WORKDEPT</b>	<b>PHONENO</b>	<b>HIREDATE</b>
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

+

<b>JOB</b>	<b>EDLEVEL</b>	<b>SEX</b>	<b>BIRTHDATE</b>	<b>SALARY</b>	<b>BONUS</b>	<b>COMM</b>
CHAR(8)	SMALLINT NOT NULL	CHAR(1)	DATE	DECIMAL(9,2)	DECIMAL(9,2)	DECIMAL(9,2)
Job	Number of years of formal education	Sex (M male, F female)	Date of birth	Yearly salary	Yearly bonus	Yearly commission

The following table contains the values in the EMPLOYEE table.

# The SAMPLE database

EMPNO	FIRSTNAME	MID INIT	LASTNAME	WORK DEPT	PHONE NO	HIREDATE	JOB	ED LEVEL	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	DECIMAL (9,2)	DECIMAL (9,2)	DECIMAL (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	LUCCHESI	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
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
	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



Name:	EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
	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
	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
	000140	bitmap	db200140.bmp
	000140	gif	db200140.gif
	000150	bitmap	db200150.bmp
	000150	gif	db200150.gif
	000190	bitmap	db200190.bmp
	000190	gif	db200190.gif

## The SAMPLE database

### EMPPROJACT table

Name:	EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
Type:	CHAR(6)	CHAR(6)	SMALLINT	DEC(5,2)	DATE	DATE
Values:	000070	AD3110	10	1.00	01/01/1982	02/01/1983
	000230	AD3111	60	1.00	01/01/1982	03/15/1982
	000230	AD3111	60	0.50	03/15/1982	04/15/1982
	000230	AD3111	70	0.50	03/15/1982	10/15/1982
	000230	AD3111	80	0.50	04/15/1982	10/15/1982
	000230	AD3111	180	0.50	10/15/1982	01/01/1983
	000240	AD3111	70	1.00	02/15/1982	09/15/1982
	000240	AD3111	80	1.00	09/15/1982	01/01/1983
	000250	AD3112	60	1.00	01/01/1982	02/01/1982
	000250	AD3112	60	0.50	02/01/1982	03/15/1982
	000250	AD3112	60	1.00	01/01/1983	02/01/1983
	000250	AD3112	70	0.50	02/01/1982	03/15/1982
	000250	AD3112	70	1.00	03/15/1982	08/15/1982
	000250	AD3112	70	0.25	08/15/1982	10/15/1982
	000250	AD3112	80	0.25	08/15/1982	10/15/1982
	000250	AD3112	80	0.50	10/15/1982	12/01/1982
	000250	AD3112	180	0.50	08/15/1982	01/01/1983
	000260	AD3113	70	0.50	06/15/1982	07/01/1982
	000260	AD3113	70	1.00	07/01/1982	02/01/1983
	000260	AD3113	80	1.00	01/01/1982	03/01/1982
	000260	AD3113	80	0.50	03/01/1982	04/15/1982
	000260	AD3113	180	0.50	03/01/1982	04/15/1982
	000260	AD3113	180	1.00	04/15/1982	06/01/1982
	000260	AD3113	180	1.00	06/01/1982	07/01/1982
	000270	AD3113	60	0.50	03/01/1982	04/01/1982
	000270	AD3113	60	1.00	04/01/1982	09/01/1982
	000270	AD3113	60	0.25	09/01/1982	10/15/1982
	000270	AD3113	70	0.75	09/01/1982	10/15/1982
	000270	AD3113	70	1.00	10/15/1982	02/01/1983
	000270	AD3113	80	1.00	01/01/1982	03/01/1982
	000270	AD3113	80	0.50	03/01/1982	04/01/1982
	000030	IF1000	10	0.50	06/01/1982	01/01/1983
	000130	IF1000	90	1.00	10/01/1982	01/01/1983
	000130	IF1000	100	0.50	10/01/1982	01/01/1983
	000140	IF1000	90	0.50	10/01/1982	01/01/1983
	000030	IF2000	10	0.50	01/01/1982	01/01/1983
	000140	IF2000	100	1.00	01/01/1982	03/01/1982
	000140	IF2000	100	0.50	03/01/1982	07/01/1982

Name:	EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
	000140	IF2000	110	0.50	03/01/1982	07/01/1982
	000140	IF2000	110	0.50	10/01/1982	01/01/1983
	000010	MA2100	10	0.50	01/01/1982	11/01/1982
	000110	MA2100	20	1.00	01/01/1982	03/01/1983
	000010	MA2110	10	1.00	01/01/1982	02/01/1983
	000200	MA2111	50	1.00	01/01/1982	06/15/1982
	000200	MA2111	60	1.00	06/15/1982	02/01/1983
	000220	MA2111	40	1.00	01/01/1982	02/01/1983
	000150	MA2112	60	1.00	01/01/1982	07/15/1982
	000150	MA2112	180	1.00	07/15/1982	02/01/1983
	000170	MA2112	60	1.00	01/01/1982	06/01/1983
	000170	MA2112	70	1.00	06/01/1982	02/01/1983
	000190	MA2112	70	1.00	01/01/1982	10/01/1982
	000190	MA2112	80	1.00	10/01/1982	10/01/1982
	000160	MA2113	60	1.00	07/15/1982	02/01/1983
	000170	MA2113	80	1.00	01/01/1982	02/01/1983
	000180	MA2113	70	1.00	04/01/1982	06/15/1982
	000210	MA2113	80	0.50	10/01/1982	02/01/1983
	000210	MA2113	180	0.50	10/01/1982	02/01/1983
	000050	OP1000	10	0.25	01/01/1982	02/01/1983
	000090	OP1010	10	1.00	01/01/1982	02/01/1983
	000280	OP1010	130	1.00	01/01/1982	02/01/1983
	000290	OP1010	130	1.00	01/01/1982	02/01/1983
	000300	OP1010	130	1.00	01/01/1982	02/01/1983
	000310	OP1010	130	1.00	01/01/1982	02/01/1983
	000050	OP1010	10	0.75	01/01/1982	02/01/1983
	000100	OP1010	10	1.00	01/01/1982	02/01/1983
	000320	OP2011	140	0.75	01/01/1982	02/01/1983
	000320	OP2011	150	0.25	01/01/1982	02/01/1983
	000330	OP2012	140	0.25	01/01/1982	02/01/1983
	000330	OP2012	160	0.75	01/01/1982	02/01/1983
	000340	OP2013	140	0.50	01/01/1982	02/01/1983
	000340	OP2013	170	0.50	01/01/1982	02/01/1983
	000020	PL2100	30	1.00	01/01/1982	09/15/1982

**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

## The SAMPLE database

Name:	EMPNO	RESUME_FORMAT	RESUME
	000130	html	db200130.htm
	000140	ascii	db200140.asc
	000140	html	db200140.htm
	000150	ascii	db200150.asc
	000150	html	db200150.htm
	000190	ascii	db200190.asc
	000190	html	db200190.htm

### 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
	1988-12-25-17.12.30.000000	BADAMSON	FWD: Fantastic year! 4th Quarter Bonus.	To: JWALKER Cc: QUINTANA, NICHOLLS Jim, Looks like our hard work has paid off. I have some good beer in the fridge if you want to come over to celebrate a bit. Delores and Heather, are you interested as well? Bruce <Forwarding from ISTERN> Subject: FWD: Fantastic year! 4th Quarter Bonus. To: Dept_D11 Congratulations on a job well done. Enjoy this year's bonus. Irv <Forwarding from CHAAS> Subject: Fantastic year! 4th Quarter Bonus. To: All_Managers Our 4th quarter results are in. We pulled together as a team and exceeded our plan! I am pleased to announce a bonus this year of 18%. Enjoy the holidays. Christine Haas

Name:	RECEIVED	SOURCE	SUBJECT	NOTE_TEXT
	1988-12-23-08.53.58.000000	ISTERN	FWD: Fantastic year! 4th Quarter Bonus.	To: Dept_D11 Congratulations on a job well done. Enjoy this year's bonus. Irv <Forwarding from CHAAS> Subject: Fantastic year! 4th Quarter Bonus. To: All_Managers Our 4th quarter results are in. We pulled together as a team and exceeded our plan! I am pleased to announce a bonus this year of 18%. Enjoy the holidays. Christine Haas
	1988-12-22-14.07.21.136421	CHAAS	Fantastic year! 4th Quarter Bonus.	To: All_Managers Our 4th quarter results are in. We pulled together as a team and exceeded our plan! I am pleased to announce a bonus this year of 18%. Enjoy the holidays. Christine Haas

### 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

### PROJ table

Name:	PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
Type:	CHAR(6)	VARCHAR(36)	CHAR(3)	CHAR(6)	DEC(5,2)	DATE	DATE	CHAR(6)
Values:	AD3100	ADMIN SERVICES	D01	000010	6.50	01/01/1982	02/01/1983	

## The SAMPLE database

Name:	PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
	AD3110	GENERAL ADMIN SYSTEMS	D21	000070	6.00	01/01/1982	02/01/1983	AD3100
	AD3111	PAYROLL PROGRAMMING	D21	000230	2.00	01/01/1982	02/01/1983	AD3100
	AD3112	PERSONNEL PROGRAMMING	D21	000250	1.00	01/01/1982	02/01/1983	AD3100
	AD3113	ACCOUNT PROGRAMMING	D21	000270	2.00	01/01/1982	02/01/1983	AD3100
	IF1000	QUERY SERVICES	C01	000030	2.00	01/01/1982	02/01/1983	
	IF2000	USER EDUCATION	C01	000030	1.00	01/01/1982	02/01/1983	
	MA2100	WELD LINE AUTOMATION	D01	000010	12.00	01/01/1982	02/01/1983	
	MA2110	W L PROGRAMMING	D11	000060	9.00	01/01/1982	02/01/1983	MA2100
	MA2111	W L PROGRAM DESIGN	D11	000220	2.00	01/01/1982	12/01/1982	MA2100
	MA2112	W L ROBOT DESIGN	D11	000150	3.00	01/01/1982	12/01/1982	MA2100
	MA2113	W L PROD CONT PROGS	D11	000160	3.00	02/15/1982	12/01/1982	MA2100
	OP1000	OPERATION SUPPORT	E01	000050	6.00	01/01/1982	02/01/1983	
	OP1010	OPERATION	E11	000090	5.00	01/01/1982	02/01/1983	OP1000
	OP2000	GEN SYSTEMS SERVICES	E01	000050	5.00	01/01/1982	02/01/1983	
	OP2010	SYSTEMS SUPPORT	E21	000100	4.00	01/01/1982	02/01/1983	OP2010
	OP2011	SCP SYSTEMS SUPPORT	E21	000320	1.00	01/01/1982	02/01/1983	OP2010
	OP2012	APPLICATIONS SUPPORT	E21	000330	1.00	01/01/1982	02/01/1983	OP2010
	OP2013	DB/DC SUPPORT	E21	000340	1.00	01/01/1982	02/01/1983	OP2010
	PL2100	WELD LINE PLANNING	B01	000020	1.00	01/01/1982	09/15/1982	MA2100

### PROJACT table

Name:	PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
Type:	CHAR(6)	SMALLINT	DEC(5,2)	DATE	DATE
Values:	AD3100	10		01/01/1982	
	AD3110	10		01/01/1982	
	AD3111	60		01/01/1982	
	AD3111	60		03/15/1982	
	AD3111	70		03/15/1982	
	AD3111	80		04/15/1982	
	AD3111	180		10/15/1982	
	AD3111	70		02/15/1982	
	AD3111	80		09/15/1982	
	AD3112	60		01/01/1982	
	AD3112	60		02/01/1982	

Name:	PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
	AD3112	60		01/01/1983	
	AD3112	70		02/01/1982	
	AD3112	70		03/15/1982	
	AD3112	70		08/15/1982	
	AD3112	80		08/15/1982	
	AD3112	80		10/15/1982	
	AD3112	180		08/15/1982	
	AD3113	70		06/15/1982	
	AD3113	70		07/01/1982	
	AD3113	80		01/01/1982	
	AD3113	80		03/01/1982	
	AD3113	180		03/01/1982	
	AD3113	180		04/15/1982	
	AD3113	180		06/01/1982	
	AD3113	60		03/01/1982	
	AD3113	60		04/01/1982	
	AD3113	60		09/01/1982	
	AD3113	70		09/01/1982	
	AD3113	70		10/15/1982	
	IF1000	10		06/01/1982	
	IF1000	90		10/01/1982	
	IF1000	100		10/01/1982	
	IF2000	10		01/01/1982	
	IF2000	100		01/01/1982	
	IF2000	100		03/01/1982	
	IF2000	110		03/01/1982	
	IF2000	110		10/01/1982	
	MA2100	10		01/01/1982	
	MA2100	20		01/01/1982	
	MA2110	10		01/01/1982	
	MA2111	50		01/01/1982	
	MA2111	60		06/15/1982	
	MA2111	40		01/01/1982	
	MA2112	60		01/01/1982	
	MA2112	180		07/15/1982	
	MA2112	70		06/01/1982	
	MA2112	70		01/01/1982	
	MA2112	80		10/01/1982	
	MA2113	60		07/15/1982	
	MA2113	80		01/01/1982	
	MA2113	70		04/01/1982	

## The SAMPLE database

Name:	PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
	MA2113	80		10/01/1982	
	MA2113	180		10/01/1982	
	OP1000	10		01/01/1982	
	OP1010	10		01/01/1982	
	OP1010	130		01/01/1982	
	OP2010	10		01/01/1982	
	OP2011	140		01/01/1982	
	OP2011	150		01/01/1982	
	OP2012	140		01/01/1982	
	OP2012	160		01/01/1982	
	OP2013	140		01/01/1982	
	OP2013	170		01/01/1982	
	PL2100	30		01/01/1982	

## 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
	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)	INTEGER
Desc:	Date of sales	Employee's last name	Region of sales	Number of sales



Name:	SALES_DATE	SALES_PERSON	REGION	SALES
Values:	12/31/2005	LUCCHESI	Ontario-South	1
	12/31/2005	LEE	Ontario-South	3
	12/31/2005	LEE	Quebec	1
	12/31/2005	LEE	Manitoba	2
	12/31/2005	GOUNOT	Quebec	1
	03/29/2006	LUCCHESI	Ontario-South	3
	03/29/2006	LUCCHESI	Quebec	1
	03/29/2006	LEE	Ontario-South	2
	03/29/1996	LEE	Ontario-North	2
	03/29/2006	LEE	Quebec	3
	03/29/2006	LEE	Manitoba	5
	03/29/2006	GOUNOT	Ontario-South	3
	03/29/2006	GOUNOT	Quebec	1
	03/29/2006	GOUNOT	Manitoba	7
	03/30/2006	LUCCHESI	Ontario-South	1
	03/30/2006	LUCCHESI	Quebec	2
	03/30/2006	LUCCHESI	Manitoba	1
	03/30/2006	LEE	Ontario-South	7
	03/30/2006	LEE	Ontario-North	3
	03/30/2006	LEE	Quebec	7
	03/30/2006	LEE	Manitoba	4
	03/30/2006	GOUNOT	Ontario-South	2
	03/30/2006	GOUNOT	Quebec	18
	03/30/2006	GOUNOT	Manitoba	1
	03/31/2006	LUCCHESI	Manitoba	1
	03/31/2006	LEE	Ontario-South	14
	03/31/2006	LEE	Ontario-North	3
	03/31/2006	LEE	Quebec	7
	03/31/2006	LEE	Manitoba	3
	03/31/2006	GOUNOT	Ontario-South	2
	03/31/2006	GOUNOT	Quebec	1
	04/01/2006	LUCCHESI	Ontario-South	3
	04/01/2006	LUCCHESI	Manitoba	1
	04/01/2006	LEE	Ontario-South	8
	04/01/2006	LEE	Ontario-North	-
	04/01/2006	LEE	Quebec	8
	04/01/2006	LEE	Manitoba	9
	04/01/2006	GOUNOT	Ontario-South	3
	04/01/2006	GOUNOT	Ontario-North	1
	04/01/2006	GOUNOT	Quebec	3
	04/01/2006	GOUNOT	Manitoba	7

### STAFF table

Name:	ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
Type:	SMALLINT NOT NULL	VARCHAR(9)	SMALLINT	CHAR(5)	SMALLINT	DECIMAL(7,2)	DECIMAL(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
	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

### PRODUCT table

Name:	PID	NAME	PRICE	PROMOPRICE	PROMOSTART	PROMOEND	DESCRIPTION
Type:	VARCHAR(10) NOT NULL	VARCHAR(128)	DECIMAL(30,2)	DECIMAL(30,2)	DATE	DATE	XML
Values:	100-100-01	Snow Shovel, Basic 22 inch	9.99	7.25	11/19/2004	12/19/2004	p1.xml
	100-101-01	Snow Shovel, Deluxe 24 inch	19.99	15.99	12/18/2005	02/28/2006	p2.xml
	100-103-01	Snow Shovel, Super Deluxe 26 inch	49.99	39.99	12/22/2005	02/22/2006	p3.xml
	100-201-01	Ice Scraper, Windshield 4 inch	3.99	--	--	--	p4.xml

Here is the XML schema definition file for the XML column in the above table:  
product.xsd

### PURCHASEORDER table

Name:	POID	STATUS	CUSTID	ORDERDATE	PORDER	COMMENTS
Type:	BIGINT NOT NULL	VARCHAR(10) NOT NULL	BIGINT	DATE	XML	VARCHAR(1000)
Values:	5000	Unshipped	1002	02/18/2006	po1.xml	THIS IS A NEW PURCHASE ORDER
	5001	Shipped	1003	02/03/2005	po2.xml	THIS IS A NEW PURCHASE ORDER
	5002	Shipped	1001	02/29/2004	po3.xml	THIS IS A NEW PURCHASE ORDER

Name:	POID	STATUS	CUSTID	ORDERDATE	PORDER	COMMENTS
	5003	Shipped	1002	02/28/2005	po4.xml	THIS IS A NEW PURCHASE ORDER
	5004	Shipped	1005	11/18/2005	po5.xml	THIS IS A NEW PURCHASE ORDER
	5006	Shipped	1002	03/01/2006	po6.xml	THIS IS A NEW PURCHASE ORDER

Here is the XML schema definition file for the XML column in the above table:  
porder.xsd

### CUSTOMER table

Name:	CID	INFO
Type:	BIGINT NOT NULL	XML
Values:	1000	c1.xml
	1001	c2.xml
	1002	c3.xml
	1003	c4.xml
	1004	c5.xml
	1005	c6.xml

Here is the XML schema definition file for the XML column in the above table:  
customer.xsd

### CATALOG table

Name:	NAME	CATALOG
Type:	VARCHAR(128) NOT NULL	XML
Values:	Spring catalog	cat1.xml cat1.xml
	Summer catalog	cat2.xml cat2.xml
	Fall catalog	cat3.xml cat3.xml
	Winter catalog	cat4.xml cat4.xml

Here is the XML schema definition file for the XML column in the above table:  
catalog.xsd

### INVENTORY table

Name:	PID	QUANTITY	LOCATION
Type:	VARCHAR(10) NOT NULL	INTEGER	VARCHAR(128)
Values:	100-100-01	5	--
	100-101-01	25	Store
	100-103-01	55	Store
	100-201-01	99	Warehouse

## The SAMPLE database

### PRODUCTSUPPLIER table

Name:	PID	SID
Type:	VARCHAR(10) NOT NULL	VARCHAR(10) NOT NULL
Values:	100-100-01	123-456-78
	100-101-01	123-456-78
	100-103-01	555-789-00
	100-201-01	989-897-23

### SUPPLIERS table

Name:	SID	ADDR
Type:	VARCHAR(10) NOT NULL	XML
Values:	123-456-78	s1.xml s1.xml
	555-789-00	s2.xml s2.xml
	989-897-23	s3.xml s3.xml
	111-898-45	s4.xml s4.xml

Here is the XML schema definition file for the XML column in the above table:  
supplier.xsd

### 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



Figure 15. Dolores M. Quintana

#### Quintana resume

The following text is found in the file db200130.asc.

**Resume: Dolores 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

Following is the contents of the file db200130.htm.

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 3//EN">
<HTML><HEAD>
<TITLE>Resume: Delores M. Quintana
```

## The SAMPLE database

```
<!-- Begin Header Records ===== -->
<!-- DB200130 SCRIPT A converted by B2H R4.1 (346) (CMS) by MJA at -->
<!-- RCHVMW2 on 16 Aug 2000 at 11:35:23 -->
<META HTTP-EQUIV="updated" CONTENT="Wed, 16 Aug 2000 11:33:57">
<META HTTP-EQUIV="review" CONTENT="Thu, 16 Aug 2001 11:33:57">
<META HTTP-EQUIV="expires" CONTENT="Fri, 16 Aug 2002 11:33:57"><BODY>
<!-- End Header Records ===== -->
<H1>Resume: Delores M. Quintana<HR>
<H2>Table of Contents
Resume: Delores M. Quintana

Personal Information

Department Information

Education

Work History

Interests

<HR><P>
<P>
<H3>Resume: Delores M. Quintana<P>
<H3>Personal Information<DL COMPACT>
<DT>Address:
<DD>1150 Eglinton Ave

Mellonville, Idaho 83725
<DT>Phone:
<DD>(208) 875-9933
<DT>Birthdate:
<DD>September 15, 1925
<DT>Sex:
<DD>Female
<DT>Marital Status:
<DD>Married
<DT>Height:
<DD>5'2"
<DT>Weight:
<DD>120 lbs.<P>
<H3>Department Information<DL COMPACT>
<DT>Employee Number:
<DD>000130
<DT>Dept Number:
<DD>C01
<DT>Manager:
<DD>Sally Kwan
<DT>Position:
<DD>Analyst
<DT>Phone:
<DD>(208) 385-4578
<DT>Hire Date:
<DD>1971-07-28<P>
<H3>Education<DL>
<P><DT>1965
<DD>Math and English, B.A.

Adelphi University
<P><DT>1960
<DD>Dental Technician

Florida Institute of Technology<P>
<H3>Work History<DL>
<P><DT>10/91 - present
<DD>Advisory Systems Analyst

Producing documentation tools for engineering department.
<P><DT>12/85 - 9/91
<DD>Technical Writer

Writer, text programmer, and planner.
<P><DT>1/79 - 11/85
```

```
<DD>COBOL Payroll Programmer

Writing payroll programs for a diesel fuel company.<P>
<H3>Interests<UL COMPACT>
Cooking
Reading
Sewing
Remodeling
```

### Nicholls photo



Figure 16. Heather A. Nicholls

### Nicholls resume

The following text is found in the file db200140.asc.

#### 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"

**Weight:**

130 lbs.

##### Department Information

**Employee Number:**

000140

**Dept Number:**

C01

**Manager:**

Sally Kwan

## The SAMPLE database

**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

Following is the content of the file db200140.htm.

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 3//EN">
<HTML><HEAD>
<TITLE>Resume: Heather A. Nicholls
<!-- Begin Header Records ===== -->
<!-- DB200140 SCRIPT A converted by B2H R4.1 (346) (CMS) by MJA at -->
<!-- RCHVMW2 on 16 Aug 2000 at 11:35:21 -->
<META HTTP-EQUIV="updated" CONTENT="Wed, 16 Aug 2000 11:34:17">
<META HTTP-EQUIV="review" CONTENT="Thu, 16 Aug 2001 11:34:17">
<META HTTP-EQUIV="expires" CONTENT="Fri, 16 Aug 2002 11:34:17"><BODY>
<!-- End Header Records ===== -->
<H1>Resume: Heather A. Nicholls<HR>
<H2>Table of Contents
Resume: Heather A. Nicholls

Personal Information

Department Information

Education

Work History

Interests

<HR><P>
<P>
<H3>Resume: Heather A. Nicholls<P>
<H3>Personal Information<DL COMPACT>
<DT>Address:
<DD>844 Don Mills Ave

Mellonville, Idaho 83734
<DT>Phone:
<DD>(208) 610-2310
```



<DT>Birthdate:  
 <DD>January 19, 1946  
 <DT>Sex:  
 <DD>Female  
 <DT>Marital Status:  
 <DD>Single  
 <DT>Height:  
 <DD>5'8"  
 <DT>Weight:  
 <DD>130 lbs.<P>  
 <H3><A NAME="Header\_3">Department Information<DL COMPACT>  
 <DT>Employee Number:  
 <DD>000140  
 <DT>Dept Number:  
 <DD>C01  
 <DT>Manager:  
 <DD>Sally Kwan  
 <DT>Position:  
 <DD>Analyst  
 <DT>Phone:  
 <DD>(208) 385-1793  
 <DT>Hire Date:  
 <DD>1976-12-15<P>  
 <H3><A NAME="Header\_4">Education<DL>  
 <P><DT>1972  
 <DD>Computer Engineering, Ph.D.  
 <BR>  
 University of Washington  
 <P><DT>1969  
 <DD>Music and Physics, B.A.  
 <BR>  
 Vassar College<P>  
 <H3><A NAME="Header\_5">Work History<DL>  
 <P><DT>2/83 - present  
 <DD>Architect, OCR Development  
 <BR>  
 Designing the architecture of OCR products.  
 <P><DT>12/76 - 1/83  
 <DD>Text Programmer  
 <BR>  
 Optical CHARacter recognition (OCR) programming in PL/I.  
 <P><DT>9/72 - 11/76  
 <DD>Punch Card Quality Analyst  
 <BR>  
 Checking punch cards met quality specifications.<P>  
 <H3><A NAME="Header\_6">Interests<UL COMPACT>  
 <LI>Model railroading  
 <LI>Interior decorating  
 <LI>Embroidery  
 <LI>Knitting<A NAME="Bot\_Of\_Page">

## Adamson photo

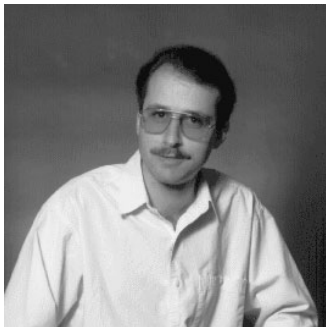


Figure 17. Bruce Adamson

## Adamson resume

The following text is found in the file db200150.asc.

### 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****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

Following is the content of the file db200150.htm.

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 3//EN">
<HTML><HEAD>
<TITLE>Resume: Bruce Adamson
<!-- Begin Header Records ===== -->
<!-- DB200150 SCRIPT A converted by B2H R4.1 (346) (CMS) by MJA at -->
<!-- RCHVMW2 on 16 Aug 2000 at 11:35:21 -->
<META HTTP-EQUIV="updated" CONTENT="Wed, 16 Aug 2000 11:34:39">
<META HTTP-EQUIV="review" CONTENT="Thu, 16 Aug 2001 11:34:39">
<META HTTP-EQUIV="expires" CONTENT="Fri, 16 Aug 2002 11:34:39"><BODY>
<!-- End Header Records ===== -->
<H1>Resume: Bruce Adamson<HR>
<H2>Table of Contents
Resume: Bruce Adamson

Personal Information

Department Information

Education

Work History

Interests

<HR><P>
<P>
<H3>Resume: Bruce Adamson<P>
<H3>Personal Information<DL COMPACT>
<DT>Address:
<DD>3600 Steeles Ave

Mellonville, Idaho 83757
<DT>Phone:
<DD>(208) 725-4489
<DT>Birthdate:
<DD>May 17, 1947
<DT>Sex:
<DD>Male
<DT>Marital Status:
<DD>Married
<DT>Height:
<DD>6'0"
```

## The SAMPLE database

```
<DT>Weight:
<DD>175 lbs.<P>
<H3>Department Information<DL COMPACT>
<DT>Employee Number:
<DD>000150
<DT>Dept Number:
<DD>D11
<DT>Manager:
<DD>Irving Stern
<DT>Position:
<DD>Designer
<DT>Phone:
<DD>(208) 385-4510
<DT>Hire Date:
<DD>1972-02-12<P>
<H3>Education<DL>
<P><DT>1971
<DD>Environmental Engineering, M.Sc.

Johns Hopkins University
<P><DT>1968
<DD>American History, B.A.

Northwestern University<P>
<H3>Work History<DL>
<P><DT>8/79 - present
<DD>Neural Network Design

Developing neural networks for machine intelligence products.
<P><DT>2/72 - 7/79
<DD>Robot Vision Development

Developing rule-based systems to emulate sight.
<P><DT>9/71 - 1/72
<DD>Numerical Integration Specialist

Helping bank systems communicate with each other.<P>
<H3>Interests<UL COMPACT>
Racing motorcycles
Building loudspeakers
Assembling personal computers
Sketching
```

### Walker photo

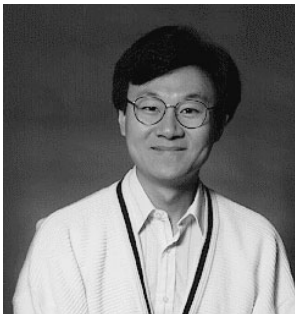


Figure 18. James H. Walker

### Walker resume

The following text is found in the file db200190.asc.

#### 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.

**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

## The SAMPLE database

- Dancing

Following is the content of the file db200190.htm.

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 3//EN">
<HTML><HEAD>
<TITLE>Resume: James H. Walker
<!-- Begin Header Records ===== -->
<!-- DB200190 SCRIPT A converted by B2H R4.1 (346) (CMS) by MJA at -->
<!-- RCHVMW2 on 16 Aug 2000 at 11:35:20 -->
<META HTTP-EQUIV="updated" CONTENT="Wed, 16 Aug 2000 11:34:59">
<META HTTP-EQUIV="review" CONTENT="Thu, 16 Aug 2001 11:34:59">
<META HTTP-EQUIV="expires" CONTENT="Fri, 16 Aug 2002 11:34:59"><BODY>
<!-- End Header Records ===== -->
<H1>Resume: James H. Walker<HR>
<H2>Table of Contents
Resume: James H. Walker

Personal Information

Department Information

Education

Work History

Interests

<HR><P>
<P>
<H3>Resume: James H. Walker<P>
<H3>Personal Information<DL COMPACT>
<DT>Address:
<DD>3500 Steeles Ave

Mellonville, Idaho 83757
<DT>Phone:
<DD>(208) 725-7325
<DT>Birthdate:
<DD>June 25, 1952
<DT>Sex:
<DD>Male
<DT>Marital Status:
<DD>Single
<DT>Height:
<DD>5'11"
<DT>Weight:
<DD>166 lbs.<P>
<H3>Department Information<DL COMPACT>
<DT>Employee Number:
<DD>000190
<DT>Dept Number:
<DD>D11
<DT>Manager:
<DD>Irving Stern
<DT>Position:
<DD>Designer
<DT>Phone:
<DD>(208) 385-2986
<DT>Hire Date:
<DD>1974-07-26<P>
<H3>Education<DL>
<P><DT>1974
<DD>Computer Studies, B.Sc.

University of Massachusetts
<P><DT>1972
<DD>Linguistic Anthropology, B.A.

University of Toronto<P>
<H3>Work History<DL>
<P><DT>6/87 - present
<DD>Microcode Design
```

<BR>  
Optimizing algorithms for mathematical functions.  
<P><DT>4/77 - 5/87  
<DD>Printer Technical Support  
<BR>  
Installing and supporting laser printers.  
<P><DT>9/74 - 3/77  
<DD>Maintenance Programming  
<BR>  
Patching assembly language compiler for mainframes.<P>  
<H3><A NAME="Header\_6">Interests<UL COMPACT>  
<LI>Wine tasting  
<LI>Skiing  
<LI>Swimming  
<LI>Dancing<A NAME="Bot\_Of\_Page">

## The SAMPLE database



---

## Appendix G. Reserved schema names and reserved words

There are restrictions on the use of certain names that are required by the database manager. In some cases, names are reserved, and cannot be used by application programs. In other cases, certain names are not recommended for use by application programs, although their use is not prevented by the database manager.

The reserved schema names are:

- SYSCAT
- SYSFUN
- SYSIBM
- SYSIBMADM
- SYSPROC
- SYSSTAT

It is strongly recommended that schema names never begin with the 'SYS' prefix, because 'SYS', by convention, is used to indicate an area that is reserved by the system. No aliases, global variables, triggers, user-defined functions, or user-defined types can be placed into a schema whose name starts with 'SYS' (SQLSTATE 42939).

The DB2QP schema and the SYSTOOLS schema are set aside for use by DB2 tools. It is recommended that users not explicitly define objects in these schemas, although their use is not prevented by the database manager.

It is also recommended that SESSION not be used as a schema name. Because declared temporary tables must be qualified by SESSION, it is possible to have an application declare a temporary table with a name that is identical to that of a persistent table, complicating the application logic. To avoid this possibility, do not use the schema SESSION except when dealing with declared temporary tables.

There are no specifically reserved words in DB2 Version 9. Keywords can be used as ordinary identifiers, except in a context where they could also be interpreted as SQL keywords. In such cases, the word must be specified as a delimited identifier. For example, COUNT cannot be used as a column name in a SELECT statement, unless it is delimited.

ISO/ANSI SQL2003 and other DB2 database products include reserved words that are not enforced by DB2 Database for Linux, UNIX, and Windows; however, it is recommended that these words not be used as ordinary identifiers, because it reduces portability.

For portability across the DB2 database products, the following should be considered reserved words:

ACTIVATE	DOCUMENT	LOCK	ROUND_CEILING
ADD	DOUBLE	LOCKMAX	ROUND_DOWN
AFTER	DROP	LOCKSIZE	ROUND_FLOOR
ALIAS	DSSIZE	LONG	ROUND_HALF_DOWN
ALL	DYNAMIC	LOOP	ROUND_HALF_EVEN
ALLOCATE	EACH	MAINTAINED	ROUND_HALF_UP
ALLOW	EDITPROC	MATERIALIZED	ROUND_UP
ALTER	ELSE	MAXVALUE	ROUTINE

## Reserved schema names and reserved words

AND	ELSEIF	MICROSECOND	ROW
ANY	ENABLE	MICROSECONDS	ROW_NUMBER
AS	ENCODING	MINUTE	ROWNUMBER
ASENSITIVE	ENCRYPTION	MINUTES	ROWS
ASSOCIATE	END	MINVALUE	ROWSET
ASUTIME	END-EXEC	MODE	RRN
AT	ENDING	MODIFIES	RUN
ATTRIBUTES	ERASE	MONTH	SAVEPOINT
AUDIT	ESCAPE	MONTHS	SCHEMA
AUTHORIZATION	EVERY	NAN	SCRATCHPAD
AUX	EXCEPT	NEW	SCROLL
AUXILIARY	EXCEPTION	NEW_TABLE	SEARCH
BEFORE	EXCLUDING	NEXTVAL	SECOND
BEGIN	EXCLUSIVE	NO	SECONDS
BETWEEN	EXECUTE	NOCACHE	SECQTY
BINARY	EXISTS	NOCYCLE	SECURITY
BUFFERPOOL	EXIT	NODENAME	SELECT
BY	EXPLAIN	NODENUMBER	SELECTIVE
CACHE	EXTERNAL	NOMAXVALUE	SEQUENCE
CALL	EXTRACT	NOMINVALUE	SESSION
CALLED	FENCED	NONE	SESSION_USER
CAPTURE	FETCH	NOORDER	SET
CARDINALITY	FIELDPROC	NORMALIZED	SIGNAL
CASCADE	FILE	NOT	SIMPLE
CASE	FINAL	NULL	SNAN
CAST	FOR	NULLS	SOME
CCSID	FOREIGN	NUMPARTS	SOURCE
CHAR	FREE	OBID	SPECIFIC
CHARACTER	FROM	OF	SQL
CHECK	FULL	OLD	SQLID
CLONE	FUNCTION	OLD_TABLE	STACKED
CLOSE	GENERAL	ON	STANDARD
CLUSTER	GENERATED	OPEN	START
COLLECTION	GET	OPTIMIZATION	STARTING
COLLID	GLOBAL	OPTIMIZE	STATEMENT
COLUMN	GO	OPTION	STATIC
COMMENT	GOTO	OR	STATMENT
COMMIT	GRANT	ORDER	STAY
CONCAT	GRAPHIC	OUT	STOGROUP
CONDITION	GROUP	OUTER	STORES
CONNECT	HANDLER	OVER	STYLE
CONNECTION	HASH	OVERRIDING	SUBSTRING
CONSTRAINT	HASHED_VALUE	PACKAGE	SUMMARY
CONTAINS	HAVING	PADDED	SYNONYM
CONTINUE	HINT	PAGESIZE	SYSFUN
COUNT	HOLD	PARAMETER	SYSIBM
COUNT_BIG	HOUR	PART	SYSPROC
CREATE	HOURS	PARTITION	SYSTEM
CROSS	IDENTITY	PARTITIONED	SYSTEM_USER
CURRENT	IF	PARTITIONING	TABLE
CURRENT_DATE	IMMEDIATE	PARTITIONS	TABLESPACE
CURRENT_LC_CTYPE	IN	PASSWORD	THEN
CURRENT_PATH	INCLUDING	PATH	TIME
CURRENT_SCHEMA	INCLUSIVE	PIECESIZE	TIMESTAMP
CURRENT_SERVER	INCREMENT	PLAN	TO
CURRENT_TIME	INDEX	POSITION	TRANSACTION
CURRENT_TIMESTAMP	INDICATOR	PRECISION	TRIGGER
CURRENT_TIMEZONE	INF	PREPARE	TRIM
CURRENT_USER	INFINITY	PREVVAL	TRUNCATE
CURSOR	INHERIT	PRIMARY	TYPE
CYCLE	INNER	PRIQTY	UNDO
DATA	INOUT	PRIVILEGES	UNION
DATABASE	INSENSITIVE	PROCEDURE	UNIQUE
DATAPARTITIONNAME	INSERT	PROGRAM	UNTIL
DATAPARTITIONNUM	INTEGRITY	PSID	UPDATE
DATE	INTERSECT	PUBLIC	USAGE
DAY	INTO	QUERY	USER

## Reserved schema names and reserved words

DAYS	IS	QUERYNO	USING
DB2GENERAL	ISOBID	RANGE	VALIDPROC
DB2GENRL	ISOLATION	RANK	VALUE
DB2SQL	ITERATE	READ	VALUES
DBINFO	JAR	READS	VARIABLE
DBPARTITIONNAME	JAVA	RECOVERY	VARIANT
DBPARTITIONNUM	JOIN	REFERENCES	VCAT
DEALLOCATE	KEEP	REFERENCING	VERSION
DECLARE	KEY	REFRESH	VIEW
DEFAULT	LABEL	RELEASE	VOLATILE
DEFAULTS	LANGUAGE	RENAME	VOLUMES
DEFINITION	LATERAL	REPEAT	WHEN
DELETE	LC_CTYPE	RESET	WHENEVER
DENSE_RANK	LEAVE	RESIGNAL	WHERE
DENSERANK	LEFT	RESTART	WHILE
DESCRIBE	LIKE	RESTRICT	WITH
DESCRIPTOR	LINKTYPE	RESULT	WITHOUT
DETERMINISTIC	LOCAL	RESULT_SET_LOCATOR	WLM
DIAGNOSTICS	LOCALDATE	RETURN	WRITE
DISABLE	LOCALE	RETURNS	XMLELEMENT
DISALLOW	LOCALTIME	REVOKE	XML EXISTS
DISCONNECT	LOCALTIMESTAMP	RIGHT	XMLNAMESPACES
DISTINCT	LOCATOR	ROLE	YEAR
DO	LOCATORS	ROLLBACK	YEARS

The following list contains the ISO/ANSI SQL2003 reserved words that are not in the previous list:

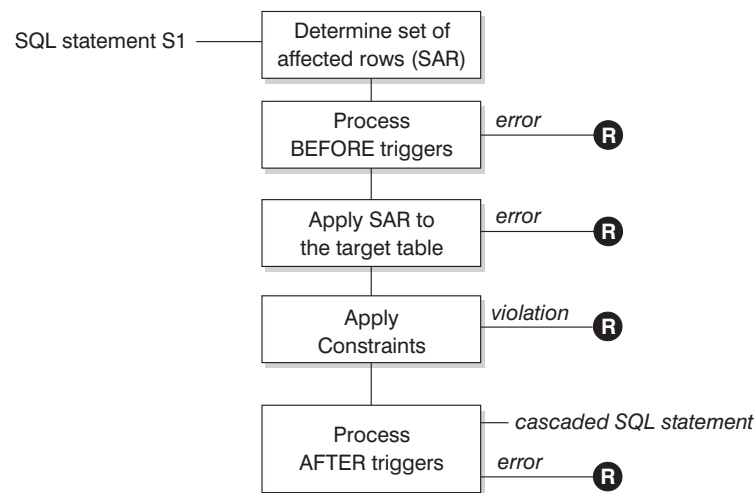
ABS	GROUPING	REGR_INTERCEPT
ARE	INT	REGR_R2
ARRAY	INTEGER	REGR_SLOPE
ASYMMETRIC	INTERSECTION	REGR_SXX
ATOMIC	INTERVAL	REGR_SXY
AVG	LARGE	REGR_SYY
BIGINT	LEADING	ROLLUP
BLOB	LN	SCOPE
BOOLEAN	LOWER	SIMILAR
BOTH	MATCH	SMALLINT
CEIL	MAX	SPECIFICTYPE
CEILING	MEMBER	SQL EXCEPTION
CHAR_LENGTH	MERGE	SQLSTATE
CHARACTER_LENGTH	METHOD	SQLWARNING
CLOB	MIN	SQRT
COALESCE	MOD	STDDEV_POP
COLLATE	MODULE	STDDEV_SAMP
COLLECT	MULTISET	SUBMULTISET
CONVERT	NATIONAL	SUM
CORR	NATURAL	SYMMETRIC
CORRESPONDING	NCHAR	TABLESAMPLE
COVAR_POP	NCLOB	TIMEZONE_HOUR
COVAR_SAMP	NORMALIZE	TIMEZONE_MINUTE
CUBE	NULLIF	TRAILING
CUME_DIST	NUMERIC	TRANSLATE
CURRENT_DEFAULT_TRANSFORM_GROUP	OCTET_LENGTH	TRANSLATION
CURRENT_ROLE	ONLY	TREAT
CURRENT_TRANSFORM_GROUP_FOR_TYPE	OVERLAPS	TRUE
DEC	OVERLAY	UESCAPE
DECIMAL	PERCENT_RANK	UNKNOWN
DEREF	PERCENTILE_CONT	UNNEST
ELEMENT	PERCENTILE_DISC	UPPER
EXEC	POWER	VAR_POP
EXP	REAL	VAR_SAMP
FALSE	RECURSIVE	VARCHAR
FILTER	REF	VARYING
FLOAT	REGR_AVGX	WIDTH_BUCKET
FLOOR	REGR_AVGY	WINDOW
FUSION	REGR_COUNT	WITHIN

## Reserved schema names and reserved words

## Appendix H. Examples of interaction between triggers and referential constraints

Update operations may cause the interaction of triggers with referential constraints and check constraints.

Figure 19 and the associated description are representative of the processing that is performed for an statement that updates data in the database.



**R** = rollback changes to before S1

Figure 19. Processing an statement with associated triggers and constraints

Figure 19 shows the general order of processing for an 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 19.

- statement  $S_1$   
This is the DELETE, INSERT, or UPDATE statement that begins the process. The statement  $S_1$  identifies a table (or an updatable view over some table) referred to as the *subject table* throughout this description.
- Determine set of affected rows  
This step is the starting point for a process that repeats for referential constraint delete rules of CASCADE and SET NULL and for cascaded statements from AFTER triggers.  
The purpose of this step is to determine the *set of affected rows* for the statement. The set of rows included is based on the statement:
  - 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).

## Examples of interaction between triggers and referential constraints

If the set of affected rows is empty, there will be no BEFORE triggers, changes to apply to the subject table, or constraints to process for the 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 the set of affected rows.

An error may occur during the processing of a triggered action in which case all changes made as a result of the original statement  $S_1$  (so far) are rolled back.

If there are no BEFORE triggers or the set of affected is empty, this step is skipped.

- Apply the set of affected rows to the subject table

The actual delete, insert, or update is applied using the set of affected rows to the subject table in the database.

An error may occur when applying the set of affected rows (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 statement  $S_1$  (so far) are rolled back.

- Apply Constraints

The constraints associated with the subject table are applied if set of affected rows 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 the set of affected rows is empty. FOR EACH ROW triggers will process the triggered action once for each row in the set of affected rows.

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 statements that are DELETE, INSERT or UPDATE statements. For the purposes of this description, each such statement is considered a *cascaded statement*.

A cascaded 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 statement as a new  $S_1$  and performing all of the steps described here recursively.

Once all triggered statements from all AFTER triggers activated by each  $S_1$  have been processed to completion, the processing of the original  $S_1$  is completed.

- 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 statement  $S_1$ . The database is therefore back in the same state as immediately prior to the execution of the original statement  $S_1$

---

## Appendix I. Explain tables

The Explain tables capture access plans when the Explain facility is activated. The Explain tables must be created before Explain can be invoked. You can create them using the documented table definitions, or you can create them by invoking the sample command line processor (CLP) script provided in the EXPLAIN.DDL file located in the misc subdirectory of the sqllib directory. To invoke the script, connect to the database where the Explain tables are required, then issue the command:

```
db2 -tf EXPLAIN.DDL
```

The Explain facility uses the following IDs as the schema when qualifying Explain tables that it is populating:

- The session authorization ID for dynamic SQL
- The statement authorization ID for static SQL

The schema can be associated with a set of Explain tables, or aliases that point to a set of Explain tables under a different schema. If no Explain tables are found under the schema, the Explain facility checks for Explain tables under the SYSTOOLS schema and attempts to use those tables.

The population of the Explain tables by the Explain facility will not activate triggers or referential or check constraints. For example, if an insert trigger were defined on the EXPLAIN\_INSTANCE table, and an eligible statement were explained, the trigger would not be activated.

To improve the performance of the Explain facility in a partitioned database system, it is recommended that the Explain tables be created in a single partition database partition group, preferably on the same database partition to which you will be connected when compiling the query.

---

### ADVISE\_INDEX table

The ADVISE\_INDEX table represents the recommended indexes.

*Table 205. ADVISE\_INDEX Table.* PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	No	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	No	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	No	Name of the package running when the dynamic statement was explained or name of the source file when static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	No	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	No	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	No	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	No	Statement number within package to which this explain information is related.

## ADVISE\_INDEX table

Table 205. ADVISE\_INDEX Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
SECTNO	INTEGER	No	No	Section number within package to which this explain information is related.
QUERYNO	INTEGER	No	No	Numeric identifier for explained SQL statement. For dynamic SQL statements (excluding the EXPLAIN SQL statement) issued through CLP or CLI, the default value is a sequentially incremented value. Otherwise, the default value is the value of STMTNO for static SQL statements and 1 for dynamic SQL statements.
QUERYTAG	CHAR(20)	No	No	Identifier tag for each explained SQL statement. For dynamic SQL statements issued through CLP (excluding the EXPLAIN SQL statement), the default value is 'CLP'. For dynamic SQL statements issued through CLI (excluding the EXPLAIN SQL statement), the default value is 'CLI'. Otherwise, the default value used is blanks.
NAME	VARCHAR(128)	No	No	Name of the index.
CREATOR	VARCHAR(128)	No	No	Qualifier of the index name.
TBNAME	VARCHAR(128)	No	No	Name of the table or nickname on which the index is defined.
TBCREATOR	VARCHAR(128)	No	No	Qualifier of the table name.
COLNAMES	CLOB(2M)	No	No	List of column names.
UNIQUERULE	CHAR(1)	No	No	Unique rule: <ul style="list-style-type: none"><li>• D = Duplicates allowed</li><li>• P = Primary index</li><li>• U = Unique entries only allowed</li></ul>
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	BIGINT	No	No	Number of leaf pages; -1 if statistics are not gathered.
NLEVELS	SMALLINT	No	No	Number of index levels; -1 if statistics are not gathered.
FIRSTKEYCARD	BIGINT	No	No	Number of distinct first key values; -1 if statistics are not gathered.
FULLKEYCARD	BIGINT	No	No	Number of distinct full key values; -1 if statistics are not gathered.
CLUSTERRATIO	SMALLINT	No	No	Degree of data clustering with the index; -1 if statistics are not gathered or if detailed index statistics are gathered (in which case, CLUSTERFACTOR will be used instead).
AVGPARTITION_ CLUSTERRATIO	SMALLINT	No	No	Degree of data clustering within a single data partition. -1 if the table is not table partitioned, if statistics are not gathered, or if detailed statistics are gathered (in which case AVGPARTITION_CLUSTERFACTOR will be used instead).



Table 205. ADVISE\_INDEX Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
AVGPARTITION_ CLUSTERFACTOR	DOUBLE	No	No	Finer measurement of the degree of clustering within a single data partition. -1 if the table is not table partitioned, if statistics are not gathered, or if the index is defined on a nickname.
AVGPARTITION_PAGE_ FETCH_PAIRS	VARCHAR(520)	No	No	A list of paired integers in character form. Each pair represents a potential buffer pool size and the corresponding page fetches required to access a single data partition from the table. Zero-length string if no data is available, or if the table is not table partitioned.
DATAPARTITION_ CLUSTERFACTOR	DOUBLE	No	No	A statistic measuring the "clustering" of the index keys with regard to data partitions. This field holds a number between zero and one, with one representing perfect clustering and zero representing no clustering.
CLUSTERFACTOR	DOUBLE	No	No	Finer measurement of degree of clustering, or -1 if detailed index statistics have not been gathered or if the index is defined on a nickname.
USERDEFINED	SMALLINT	No	No	Defined by the user.
SYSTEM_REQUIRED	SMALLINT	No	No	<ul style="list-style-type: none"> <li>• 1 if one or the other of the following conditions is met: <ul style="list-style-type: none"> <li>– This index is required for a primary or unique key constraint, or this index is a dimension block index or composite block index for a multi-dimensional clustering (MDC) table.</li> <li>– This is an index on the (OID) column of a typed table.</li> </ul> </li> <li>• 2 if both of the following conditions are met: <ul style="list-style-type: none"> <li>– This index is required for a primary or unique key constraint, or this index is a dimension block index or composite block index for an MDC table.</li> <li>– This is an index on the (OID) column of a typed table.</li> </ul> </li> <li>• 0 otherwise.</li> </ul>
CREATE_TIME	TIMESTAMP	No	No	Time when the index was created.
STATS_TIME	TIMESTAMP	Yes	No	Last time when any change was made to recorded statistics for this index. Null if no statistics available.
PAGE_FETCH_PAIRS	VARCHAR(520)	No	No	A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the table with this index using that hypothetical buffer. (Zero-length string if no data available.)
REMARKS	VARCHAR(254)	Yes	No	User-supplied comment, or null.
DEFINER	VARCHAR(128)	No	No	User who created the index.
CONVERTED	CHAR(1)	No	No	Reserved for future use.

## ADVISE\_INDEX table

Table 205. ADVISE\_INDEX Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
SEQUENTIAL_PAGES	BIGINT	No	No	Number of leaf pages located on disk in index key order with few or no large gaps between them. (-1 if no statistics are available.)
DENSITY	INTEGER	No	No	Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percent (integer between 0 and 100, -1 if no statistics are available.)
FIRST2KEYCARD	BIGINT	No	No	Number of distinct keys using the first two columns of the index (-1 if no statistics or inapplicable)
FIRST3KEYCARD	BIGINT	No	No	Number of distinct keys using the first three columns of the index (-1 if no statistics or inapplicable)
FIRST4KEYCARD	BIGINT	No	No	Number of distinct keys using the first four columns of the index (-1 if no statistics or inapplicable)
PCTFREE	SMALLINT	No	No	Percentage of each index leaf page to be reserved during initial building of the index. This space is available for future inserts after the index is built.
UNIQUE_COLCOUNT	SMALLINT	No	No	The number of columns required for a unique key. Always <=COLCOUNT. < COLCOUNT only if there a include columns. -1 if index has no unique key (permits duplicates)
MINPCTUSED	SMALLINT	No	No	If not zero, then online index defragmentation is enabled, and the value is the threshold of minimum used space before merging pages.
REVERSE_SCANS	CHAR(1)	No	No	<ul style="list-style-type: none"> <li>• Y = Index supports reverse scans</li> <li>• N = Index does not support reverse scans</li> </ul>
USE_INDEX	CHAR(1)	Yes	No	<ul style="list-style-type: none"> <li>• Y = index recommended or evaluated</li> <li>• N = index not to be recommended</li> <li>• R = an existing clustering RID index was recommended (by the Design Advisor) to be unclustered; this is the case when a new clustering RID index is recommended for the table</li> </ul>
CREATION_TEXT	CLOB(2M)	No	No	The SQL statement used to create the index.
PACKED_DESC	BLOB(1M)	Yes	No	Internal description of the table.
RUN_ID	TIMESTAMP	Yes	FK	A value corresponding to the START_TIME of a row in the ADVISE_INSTANCE table, linking it to the same Design Advisor run.
INDEXTYPE	VARCHAR(4)	No	No	Type of index. <ul style="list-style-type: none"> <li>• CLUS = Clustering</li> <li>• REG = Regular</li> <li>• DIM = Dimension block index</li> <li>• BLOK = Block index</li> </ul>
EXISTS	CHAR(1)	No	No	Set to 'Y' if the index exists in the database catalog.

Table 205. *ADVISE\_INDEX Table (continued)*. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
RIDTOBLOCK	CHAR(1)	No	No	Set to 'Y' if the RID index was used to make a block index in the Design Advisor.

## ADVISE\_INSTANCE table

The ADVISE\_INSTANCE table contains information about db2advis execution, including start time. Contains one row for each execution of db2advis. Other ADVISE tables have a foreign key (RUN\_ID) that links to the START\_TIME column of the ADVISE\_INSTANCE table for rows created during the same Design Advisor run.

Table 206. *ADVISE\_INSTANCE Table*. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
START_TIME	TIMESTAMP	No	PK	Time at which db2advis execution begins.
END_TIME	TIMESTAMP	No	No	Time at which db2advis execution ends.
MODE	VARCHAR(4)	No	No	The value that was specified with the -m option on the Design Advisor; for example, 'MC' to specify MQT and MDC.
WKLD_COMPRESSION	CHAR(4)	No	No	The workload compression under which the Design Advisor was run.
STATUS	CHAR(9)	No	No	The status of a Design Advisor run. Status can be 'STARTED', 'COMPLETED' (if successful), or an error number that is prefixed by 'EI' for internal errors or 'EX' for external errors, in which case the error number represents the SQLCODE.

## ADVISE\_MQT table

The ADVISE\_MQT table contains information about materialized query tables (MQT) recommended by the Design Advisor.

Table 207. *ADVISE\_MQT Table*. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	No	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	No	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	No	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	No	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	No	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	No	Level of Explain information for which this row is relevant.

## ADVISE\_MQT table

Table 207. *ADVISE\_MQT Table (continued)*. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
STMTNO	INTEGER	No	No	Statement number within package to which this Explain information is related.
SECTNO	INTEGER	No	No	Statement number within package to which this Explain information is related.
NAME	VARCHAR(128)	No	No	MQT name.
CREATOR	VARCHAR(128)	No	No	MQT creator name.
IID	SMALLINT	No	No	Internal identifier.
CREATE_TIME	TIMESTAMP	No	No	Time at which the MQT was created.
STATS_TIME	TIMESTAMP	Yes	No	Time at which statistics were taken.
NUMROWS	DOUBLE	No	No	The number of estimated rows in the MQT.
NUMCOLS	SMALLINT	No	No	Number of columns defined in the MQT.
ROWSIZE	DOUBLE	No	No	Average length (in bytes) of a row in the MQT.
BENEFIT	FLOAT	No	No	Reserved for future use.
USE_MQT	CHAR(1)	Yes	No	Set to 'Y' when the MQT is recommended.
MQT_SOURCE	CHAR(1)	Yes	No	Indicates where the MQT candidate was generated. Set to 'I' if the MQT candidate is a refresh-immediate MQT, or 'D' if it can only be created as a full refresh-deferred MQT.
QUERY_TEXT	CLOB(2M)	No	No	Contains the query that defines the MQT.
CREATION_TEXT	CLOB(2M)	No	No	Contains the CREATE TABLE DDL for the MQT.
SAMPLE_TEXT	CLOB(2M)	No	No	Contains the sampling query that is used to get detailed statistics for the MQT. Only used when detailed statistics are required for the Design Advisor. The resulting sampled statistics will be shown in this table. If null, then no sampling query was created for this MQT.
COLSTATS	CLOB(2M)	No	No	Contains the column statistics for the MQT (if not null). These statistics are in XML format and include the column name, column cardinality and, optionally, the HIGH2KEY and LOW2KEY values.
EXTRA_INFO	BLOB(2M)	No	No	Reserved for miscellaneous output.
TBSPACE	VARCHAR(128)	No	No	The table space that is recommended for the MQT.
RUN_ID	TIMESTAMP	Yes	FK	A value corresponding to the START_TIME of a row in the ADVISE_INSTANCE table, linking it to the same Design Advisor run.
REFRESH_TYPE	CHAR(1)	No	No	Set to 'I' for immediate or 'D' for deferred.
EXISTS	CHAR(1)	No	No	Set to 'Y' if the MQT exists in the database catalog.

## ADVISE\_PARTITION table

The ADVISE\_PARTITION table contains information about database partitions recommended by the Design Advisor, and can only be populated in a partitioned database environment.

Table 208. ADVISE\_PARTITION Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	No	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	No	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	No	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	No	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	No	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	No	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	No	Statement number within package to which this Explain information is related.
SECTNO	INTEGER	No	No	Statement number within package to which this Explain information is related.
QUERYNO	INTEGER	No	No	Numeric identifier for explained SQL statement. For dynamic SQL statements (excluding the EXPLAIN SQL statement) issued through CLP or CLI, the default value is a sequentially incremented value. Otherwise, the default value is the value of STMTNO for static SQL statements and 1 for dynamic SQL statements.
QUERYTAG	CHAR(20)	No	No	Identifier tag for each explained SQL statement. For dynamic SQL statements issued through CLP (excluding the EXPLAIN SQL statement), the default value is 'CLP'. For dynamic SQL statements issued through CLI (excluding the EXPLAIN SQL statement), the default value is 'CLI'. Otherwise, the default value used is blanks.
TBNAME	VARCHAR(128)	Yes	No	Specifies the table name.
TBCREATOR	VARCHAR(128)	Yes	No	Specifies the table creator name.
PMID	SMALLINT	Yes	No	Specifies the distribution map ID.
TBSPACE	VARCHAR(128)	Yes	No	Specifies the table space in which the table resides.
COLNAMES	CLOB(2M)	Yes	No	Specifies database partition column names, separated by commas.
COLCOUNT	SMALLINT	Yes	No	Specifies the number of database partitioning columns.
REPLICATE	CHAR(1)	Yes	No	Specifies whether or not the database partition is replicated.
COST	DOUBLE	Yes	No	Specifies the cost of using the database partition.
USEIT	CHAR(1)	Yes	No	Specifies whether or not the database partition is used in EVALUATE PARTITION mode. A database partition is used if USEIT is set to 'Y' or 'y'.
RUN_ID	TIMESTAMP	Yes	FK	A value corresponding to the START_TIME of a row in the ADVISE_INSTANCE table, linking it to the same Design Advisor run.

## ADVISE\_TABLE table

---

### ADVISE\_TABLE table

The ADVISE\_TABLE table stores the data definition language (DDL) for table creation, using the final Design Advisor recommendations for materialized query tables (MQTs), multidimensional clustered tables (MDCs), and database partitioning.

*Table 209. ADVISE\_TABLE Table.* PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
RUN_ID	TIMESTAMP	Yes	FK	A value corresponding to the START_TIME of a row in the ADVISE_INSTANCE table, linking it to the same Design Advisor run.
TABLE_NAME	VARCHAR(128)	No	No	Name of the table.
TABLE_SCHEMA	VARCHAR(128)	No	No	Name of the table creator.
TABLESPACE	VARCHAR(128)	No	No	The table space in which the table is to be created.
SELECTION_FLAG	VARCHAR(4)	No	No	Indicates the recommendation type. Valid values are 'M' for MQT, 'P' for database partitioning, and 'C' for MDC. This field can include any subset of these values. For example, 'MC' indicates that the table is recommended as an MQT and an MDC table.
TABLE_EXISTS	CHAR(1)	No	No	Set to 'Y' if the table exists in the database catalog.
USE_TABLE	CHAR(1)	No	No	Set to 'Y' if the table has recommendations from the Design Advisor.
GEN_COLUMNS	CLOB(2M)	No	No	Contains a generated columns string if this row includes an MDC recommendation that requires generated columns in the create table DDL.
ORGANIZE_BY	CLOB(2M)	No	No	For MDC recommendations, contains the ORGANIZE BY clause of the create table DDL.
CREATION_TEXT	CLOB(2M)	No	No	Contains the create table DDL.
ALTER_COMMAND	CLOB(2M)	No	No	Contains an ALTER TABLE statement for the table.

---

### ADVISE\_WORKLOAD table

The ADVISE\_WORKLOAD table represents the statement that makes up the workload.

*Table 210. ADVISE\_WORKLOAD Table.* PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
WORKLOAD_NAME	CHAR(128)	No	No	Name of the collection of SQL statements (workload) to which this statement belongs.
STATEMENT_NO	INTEGER	No	No	Statement number within the workload to which this explain information is related.
STATEMENT_TEXT	CLOB(1M)	No	No	Content of the SQL statement.
STATEMENT_TAG	VARCHAR(256)	No	No	Identifier tag for each explained SQL statement.
FREQUENCY	INTEGER	No	No	The number of times this statement appears within the workload.

Table 210. ADVISE\_WORKLOAD Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
IMPORTANCE	DOUBLE	No	No	Importance of the statement.
WEIGHT	DOUBLE	No	No	Priority of the statement.
COST_BEFORE	DOUBLE	Yes	No	The cost of the query (in timerons) if the recommendations are not created.
COST_AFTER	DOUBLE	Yes	No	The cost of the query (in timerons) if the recommendations are created. COST_AFTER reflects all recommendations except those that pertain to clustered indexes and multidimensional clustering (MDC).
COMPILABLE	CHAR(17)	Yes	No	Indicates any query compile errors that occurred while trying to prepare the statement. If this column is NULL or does not start with SQLCA, the SQL query could be compiled by db2advis. If a compile error is found by db2advis or the Design Advisor, the COMPILABLE column value consists of an 8 byte long SQLCA.sqlcaid field, followed by a colon (:), and an 8 byte long SQLCA.sqlstate field, which is the return code for the SQL statement.

## EXPLAIN\_ARGUMENT table

The EXPLAIN\_ARGUMENT table represents the unique characteristics for each individual operator, if there are any.

Table 211. EXPLAIN\_ARGUMENT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	FK	Statement number within package to which this Explain information is related.
SECTNO	INTEGER	No	FK	Section number within package to which this Explain information is related.
OPERATOR_ID	INTEGER	No	No	Unique ID for this operator within this query.
ARGUMENT_TYPE	CHAR(8)	No	No	The type of argument for this operator.

## EXPLAIN\_ARGUMENT table

Table 211. EXPLAIN\_ARGUMENT Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
ARGUMENT_VALUE	VARCHAR(1024)	Yes	No	The value of the argument for this operator. NULL if the value is in LONG_ARGUMENT_VALUE.
LONG_ARGUMENT_VALUE	CLOB(2M)	Yes	No	The value of the argument for this operator, when the text will not fit in ARGUMENT_VALUE. NULL if the value is in ARGUMENT_VALUE.

Table 212. ARGUMENT\_TYPE and ARGUMENT\_VALUE column values

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
AGGMODE	COMPLETE PARTIAL INTERMEDIATE FINAL	Partial aggregation indicators.
BITFLTR	INTEGER FALSE	Size of bit filter used by hash join.
BLD_LEVEL	DB2 Build Identifier	Internal identification string for source code version.
BLKLOCK	EXCLUSIVE INTENT EXCLUSIVE INTENT SHARE NONE SHARE UPDATE	Block level lock intent.
CSERQY	TRUE FALSE	Remote query is a common subexpression.
CSETEMP	TRUE FALSE	Temporary Table over Common Subexpression Flag.
DIRECT	TRUE	Direct fetch indicator.
DPESTFLG	TRUE FALSE	Indicates whether or not the DPNUMPRT value is based on an estimate. Possible values are 'TRUE' (DPNUMPRT represents the estimated number of accessed data partitions) or 'FALSE' (DPNUMPRT represents the actual number of accessed data partitions).
DPLSTPRT	NONE CHARACTER	Represents accessed data partitions. It is a comma-delimited list (for example: 1,3,5) or a hyphenated list (for example: 1-5) of accessed data partitions. A value of 'NONE' means that no data partition remains after specified predicates have been applied.
DPNUMPRT	INTEGER	Represents the actual or estimated number of data partitions accessed.



Table 212. ARGUMENT\_TYPE and ARGUMENT\_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
DSTSEVER	Server name	Destination (ship from) server.
DUPLWARN	TRUE FALSE	Duplicates Warning flag.
EARLYOUT	LEFT RIGHT GROUPBY NONE	Early out indicator. LEFT indicates that each row from the outer table only needs to be joined with at most one row from the inner table. RIGHT indicates that each row from the inner table only needs to be joined with at most one row from the outer table. NONE indicates no early out processing. GROUPBY indicates that early out processing is allowed because of a group by operation.
ENVVAR	Each row of this type will contain: <ul style="list-style-type: none"> <li>• Environment variable name</li> <li>• Environment variable value</li> </ul>	Environment variable affecting the optimizer
ERRTOL	Each row of this type will contain an SQLSTATE and SQLCODE pair.	A list of errors to be tolerated.
FETCHMAX	IGNORE INTEGER	Override value for MAXPAGES argument on FETCH operator.
GREEDY	TRUE	Indicates optimizer used greedy algorithm to plan access.
GLOBLOCK	EXCLUSIVE INTENT EXCLUSIVE INTENT NONE INTENT SHARE NO LOCK OBTAINED SHARE SHARE INTENT EXCLUSIVE SUPER EXCLUSIVE UPDATE	Represents global lock intent information for a partitioned table object.
GROUPBYC	TRUE FALSE	Whether Group By columns were provided.
GROUPBYN	Integer	Number of comparison columns.
GROUPBYR	Each row of this type will contain: <ul style="list-style-type: none"> <li>• Ordinal value of column in group by clause (followed by a colon and a space)</li> <li>• Name of column</li> </ul>	Group By requirement.
HASHCODE	24 32	Size (in bits) of hash code used for hash join.

## EXPLAIN\_ARGUMENT table

Table 212. ARGUMENT\_TYPE and ARGUMENT\_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
INNERCOL	Each row of this type will contain: <ul style="list-style-type: none"><li>• Ordinal value of column in order (followed by a colon and a space)</li><li>• Name of column</li><li>• Order value</li></ul> (A) Ascending (D) Descending	Inner order columns.
INPUTXID	A context node identifier	INPUTXID identifies the input context node used by the XSCAN operator.
ISCANMAX	IGNORE INTEGER	Override value for MAXPAGES argument on ISCAN operator.
JN INPUT	INNER OUTER	Indicates if operator is the operator feeding the inner or outer of a join.
LISTENER	TRUE FALSE	Listener Table Queue indicator.
MAXPAGES	ALL NONE INTEGER	Maximum pages expected for Prefetch.
MAXRIDS	NONE INTEGER	Maximum Row Identifiers to be included in each list prefetch request.
NUMROWS	INTEGER	Number of rows expected to be sorted.
ONEFETCH	TRUE FALSE	One Fetch indicator.
OUTERCOL	Each row of this type will contain: <ul style="list-style-type: none"><li>• Ordinal value of column in order (followed by a colon and a space)</li><li>• Name of column</li><li>• Order value</li></ul> (A) Ascending (D) Descending	Outer order columns.
OUTERJN	LEFT RIGHT FULL LEFT (ANTI) RIGHT (ANTI)	Outer join indicator.
PARTCOLS	Name of Column	Partitioning columns for operator.

Table 212. ARGUMENT\_TYPE and ARGUMENT\_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
PREFETCH	LIST NONE SEQUENTIAL	Type of Prefetch Eligible.
REOPT	ALWAYS ONCE	The statement is optimized using bind-in values for parameter markers, host variables, and special registers.
RMTQTEXT	Query text	Remote Query Text
RNG_PROD	Function name	Range producing function for extended index access.
ROWLOCK	EXCLUSIVE NONE REUSE SHARE SHORT (INSTANT) SHARE UPDATE	Row Lock Intent.
ROWWIDTH	INTEGER	Width of row to be sorted.
RSUFFIX	Query text	Remote SQL suffix.
SCANDIR	FORWARD REVERSE	Scan Direction.
SCANGRAN	INTEGER	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 PAGE	Intra-partition parallelism, scan granularity unit.
SHARED	TRUE	Intra-partition parallelism, shared TEMP indicator.
SLOWMAT	TRUE FALSE	Slow Materialization flag.
SNGLPROD	TRUE FALSE	Intra-partition parallelism sort or temp produced by a single agent.
SORTKEY	Each row of this type will contain: <ul style="list-style-type: none"> <li>• Ordinal value of column in key (followed by a colon and a space)</li> <li>• Name of column</li> <li>• Order value <ul style="list-style-type: none"> <li>(A) Ascending</li> <li>(D) Descending</li> </ul> </li> </ul>	Sort key columns.

## EXPLAIN\_ARGUMENT table

Table 212. ARGUMENT\_TYPE and ARGUMENT\_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
SORTTYPE	PARTITIONED SHARED ROUND ROBIN REPLICATED	Intra-partition parallelism, sort type.
SRCSEVER	Server name	Source (ship to) server.
SPILED	INTEGER	Estimated number of pages in SORT spill.
SQLCA	Warning information	Warnings and reason codes issued during Explain operation.
STMTHEAP	INTEGER	Size of statement heap at start of statement compile.
STREAM	TRUE FALSE	Remote source is streaming.
TABLOCK	EXCLUSIVE INTENT EXCLUSIVE INTENT NONE INTENT SHARE REUSE SHARE SHARE INTENT EXCLUSIVE SUPER EXCLUSIVE UPDATE	Table Lock Intent.
TEMPSIZE	INTEGER	Temporary table page size.
TQDEGREE	INTEGER	Intra-partition parallelism, number of subagents accessing Table Queue.
TQMERGE	TRUE FALSE	Merging (sorted) Table Queue indicator.
TQREAD	READ AHEAD STEPPING SUBQUERY STEPPING	Table Queue reading property.
TQSEND	BROADCAST DIRECTED SCATTER SUBQUERY DIRECTED	Table Queue send property.
TQ_TYPE	LOCAL	Intra-partition parallelism, Table Queue.
TQ_ORIGIN	ASYNCHRONY	The reason that Table Queue was introduced into the access plan.

Table 212. ARGUMENT\_TYPE and ARGUMENT\_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
TRUNCTQ	INPUT OUTPUT INPUT AND OUTPUT	Truncated Table Queue indicator. INPUT indicates that truncation occurs on input to the Table Queue. OUPUT indicates that truncation occurs on output from the Table Queue. INPUT and OUTPUT indicates that truncation occurs on both input to the Table Queue and on output from the Table Queue.
TRUNCSRT	TRUE	Truncated sort (limits number of rows produced).
UNIQUE	TRUE FALSE	Uniqueness indicator.
UNIKEY	Each row of this type will contain: <ul style="list-style-type: none"> <li>Ordinal value of column in key (followed by a colon and a space)</li> <li>Name of Column</li> </ul>	Unique key columns.
VOLATILE	TRUE	Volatile table
XDFOUT	DECIMAL	XDFOUT indicates the expected number of documents to be returned by the XISCAN operator for each context node.
XLOGID	An identifier consisting of an SQL schema name and the name of an index over XML data	XLOGID identifies the index over XML data chosen by the optimizer for the XISCAN operator.
XPATH	An XPATH expression and result set in an internal format	This argument indicates the evaluation of an XPATH expression by the XSCAN operator.
XPHYID	An identifier consisting of an SQL schema name and the name of a physical index over XML data	XPHYID identifies the physical index that is associated with an index over XML data used by the XISCAN operator.

## EXPLAIN\_DIAGNOSTIC table

The EXPLAIN\_DIAGNOSTIC table contains an entry for each diagnostic message produced for a particular instance of an explained statement in the EXPLAIN\_STATEMENT table.

The EXPLAIN\_GET\_MSGS table function queries the EXPLAIN\_DIAGNOSTIC and EXPLAIN\_DIAGNOSTIC\_DATA Explain tables and returns formatted messages.

Table 213. EXPLAIN\_DIAGNOSTIC Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	PK, FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	PK, FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	PK, FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.

## EXPLAIN\_DIAGNOSTIC table

Table 213. EXPLAIN\_DIAGNOSTIC Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
SOURCE_SCHEMA	VARCHAR(128)	No	PK, FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	PK, FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	PK, FK	Level of Explain information for which this row is relevant. Valid values are:  O Original text (as entered by user)  P PLAN SELECTION
STMTNO	INTEGER	No	PK, FK	Statement number within package to which this Explain information is related. Set to 1 for dynamic Explain SQL statements. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS system catalog view.
SECTNO	INTEGER	No	PK, FK	Section number within package that contains this SQL statement. For dynamic Explain SQL statements, this is the section number used to hold the section for this statement at run time. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS system catalog view.
DIAGNOSTIC_ID	INTEGER	No	PK	ID of the diagnostic for a particular instance of a statement in the EXPLAIN_STATEMENT table.
CODE	INTEGER	No	No	A unique number assigned to each diagnostic message. The number can be used by a message API to retrieve the full text of the diagnostic message.

## EXPLAIN\_DIAGNOSTIC\_DATA table

The EXPLAIN\_DIAGNOSTIC\_DATA table contains message tokens for specific diagnostic messages that are recorded in the EXPLAIN\_DIAGNOSTIC table. The message tokens provide additional information that is specific to the execution of the SQL statement that generated the message.

The EXPLAIN\_GET\_MSGS table function queries the EXPLAIN\_DIAGNOSTIC and EXPLAIN\_DIAGNOSTIC\_DATA Explain tables, and returns formatted messages.

Table 214. EXPLAIN\_DIAGNOSTIC\_DATA Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	FK	Schema, or qualifier, of source of Explain request.

## EXPLAIN\_DIAGNOSTIC\_DATA table

Table 214. EXPLAIN\_DIAGNOSTIC\_DATA Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant. Valid values are:  <b>O</b> Original text (as entered by user) <b>P</b> PLAN SELECTION
STMTNO	INTEGER	No	FK	Statement number within package to which this Explain information is related. Set to 1 for dynamic Explain SQL statements. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS system catalog view.
SECTNO	INTEGER	No	FK	Section number within package that contains this SQL statement. For dynamic Explain SQL statements, this is the section number used to hold the section for this statement at run time. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS system catalog view.
DIAGNOSTIC_ID	INTEGER	No	PK	ID of the diagnostic for a particular instance of a statement in the EXPLAIN_STATEMENT table.
ORDINAL	INTEGER	No	No	Position of token in the full message text.
TOKEN	VARCHAR(1000)	Yes	No	Message token to be inserted into the full message text; might be truncated.
TOKEN_LONG	BLOB(3M)	Yes	No	More detailed information, if available.

## EXPLAIN\_INSTANCE table

The EXPLAIN\_INSTANCE table is the main control table for all Explain information. Each row of data in the Explain tables is explicitly linked to one unique row in this table. The EXPLAIN\_INSTANCE table gives basic information about the source of the SQL statements being explained as well as information about the environment in which the explanation took place.

Table 215. EXPLAIN\_INSTANCE Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	PK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	PK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	PK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	PK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	PK	Version of the source of the Explain request.

## EXPLAIN\_INSTANCE table

Table 215. EXPLAIN\_INSTANCE Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_OPTION	CHAR(1)	No	No	Indicates what Explain Information was requested for this request.  Possible values are: <b>P</b> PLAN SELECTION
SNAPSHOT_TAKEN	CHAR(1)	No	No	Indicates whether an Explain Snapshot was taken for this request.  Possible values are: <b>Y</b> Yes, an Explain Snapshot(s) was taken and stored in the EXPLAIN_STATEMENT table. Regular Explain information was also captured. <b>N</b> No Explain Snapshot was taken. Regular Explain information was captured. <b>O</b> Only an Explain Snapshot was taken. Regular Explain information was not captured.
DB2_VERSION	CHAR(7)	No	No	Release number for the DB2 product that processed this explain request. Format is <i>vv.rr.m</i> , where: <b>vv</b> Version number <b>rr</b> Release number <b>m</b> Maintenance release number
SQL_TYPE	CHAR(1)	No	No	Indicates whether the Explain Instance was for static or dynamic SQL.  Possible values are: <b>S</b> Static SQL <b>D</b> 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.  Possible values are: <b>N</b> No Blocking <b>U</b> Block Unambiguous Cursors <b>B</b> Block All Cursors



Table 215. EXPLAIN\_INSTANCE Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
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.  Possible values are: <b>RR</b> Repeatable Read <b>RS</b> Read Stability <b>CS</b> Cursor Stability <b>UR</b> 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 <b>avg_appls</b> configuration parameter at the time of the Explain invocation.
SORTHEAP	INTEGER	No	No	Contains the value of the <b>sortheap</b> database configuration parameter at the time of the Explain invocation.
LOCKLIST	INTEGER	No	No	Contains the value of the <b>locklist</b> database configuration parameter at the time of the Explain invocation.
MAXLOCKS	SMALLINT	No	No	Contains the value of the <b>maxlocks</b> database configuration parameter at the time of the Explain invocation.
LOCKS_AVAIL	INTEGER	No	No	Contains the number of locks assumed to be available by the optimizer for each user. (Derived from <b>locklist</b> and <b>maxlocks</b> .)
CPU_SPEED	DOUBLE	No	No	Contains the value of the <b>cpuspeed</b> database manager configuration parameter at the time of the Explain invocation.
REMARKS	VARCHAR(254)	Yes	No	User-provided comment.
DBHEAP	INTEGER	No	No	Contains the value of the <b>dbheap</b> database configuration parameter at the time of Explain invocation.
COMM_SPEED	DOUBLE	No	No	Contains the value of the <b>comm_bandwidth</b> database configuration parameter at the time of Explain invocation.
PARALLELISM	CHAR(2)	No	No	Possible values are: <ul style="list-style-type: none"> <li>• N = No parallelism</li> <li>• P = Intra-partition parallelism</li> <li>• IP = Inter-partition parallelism</li> <li>• BP = Intra-partition parallelism and inter-partition parallelism</li> </ul>
DATAJOINER	CHAR(1)	No	No	Possible values are: <ul style="list-style-type: none"> <li>• N = Non-federated systems plan</li> <li>• Y = Federated systems plan</li> </ul>

## EXPLAIN\_INSTANCE table

---

## EXPLAIN\_OBJECT table

The EXPLAIN\_OBJECT table identifies those data objects required by the access plan generated to satisfy the SQL statement.

Table 216. EXPLAIN\_OBJECT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	FK	Statement number within package to which this explain information is related.
SECTNO	INTEGER	No	FK	Section number within package to which this explain information is related.
OBJECT_SCHEMA	VARCHAR(128)	No	No	Schema to which this object belongs.
OBJECT_NAME	VARCHAR(128)	No	No	Name of the object.
OBJECT_TYPE	CHAR(2)	No	No	Descriptive label for the type of object.
CREATE_TIME	TIMESTAMP	Yes	No	Time of Object's creation; null if a table function.
STATISTICS_TIME	TIMESTAMP	Yes	No	Last time of update to statistics for this object; null if statistics do not exist for this object.
COLUMN_COUNT	SMALLINT	No	No	Number of columns in this object.
ROW_COUNT	INTEGER	No	No	Estimated number of rows in this object.
WIDTH	INTEGER	No	No	The average width of the object in bytes. Set to -1 for an index.
PAGES	BIGINT	No	No	Estimated number of pages that the object occupies in the buffer pool. Set to -1 for a table function.
DISTINCT	CHAR(1)	No	No	Indicates whether the rows in the object are distinct (that is, whether there are duplicates).  Possible values are:  Y      Yes  N      No
TABLESPACE_NAME	VARCHAR(128)	Yes	No	Name of the table space in which this object is stored; set to null if no table space is involved.

Table 216. EXPLAIN\_OBJECT Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
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 $\geq 1$ , this is the CLUSTERRATIO. If $\geq 0$ and $< 1$ , this is the CLUSTERFACTOR. Set to -1 for a table, table function, or if this statistic is not available.
NLEAF	BIGINT	No	No	Number of leaf pages this index object's values occupy. Set to -1 for a table, table function, or if this statistic is not available.
NLEVELS	INTEGER	No	No	Number of index levels in this index object's tree. Set to -1 for a table, table function, or if this statistic is not available.
FULLKEYCARD	BIGINT	No	No	Number of distinct full key values contained in this index object. Set to -1 for a table, table function, or if this statistic is not available.
OVERFLOW	BIGINT	No	No	Total number of overflow records in the table. Set to -1 for an index, table function, or if this statistic is not available.
FIRSTKEYCARD	BIGINT	No	No	Number of distinct first key values. Set to -1 for a table, table function, or if this statistic is not available.
FIRST2KEYCARD	BIGINT	No	No	Number of distinct first key values using the first {2,3,4} 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	BIGINT	No	No	Number of leaf pages located on disk in index key order with few or no large gaps between them. Set to -1 for a table, table function, or if this statistic is not available.
DENSITY	INTEGER	No	No	Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percentage (integer between 0 and 100). Set to -1 for a table, table function, or if this statistic is not available.
STATS_SRC	CHAR(1)	No	No	Indicates the source for the statistics. Set to 1 if from single node.
AVERAGE_SEQUENCE_GAP	DOUBLE	No	No	Gap between sequences.

## EXPLAIN\_OBJECT table

Table 216. EXPLAIN\_OBJECT Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
AVERAGE_SEQUENCE_FETCH_GAP	DOUBLE	No	No	Gap between sequences when fetching using the index.
AVERAGE_SEQUENCE_PAGES	DOUBLE	No	No	Average number of index pages accessible in sequence.
AVERAGE_SEQUENCE_FETCH_PAGES	DOUBLE	No	No	Average number of table pages accessible in sequence when fetching using the index.
AVERAGE_RANDOM_PAGES	DOUBLE	No	No	Average number of random index pages between sequential page accesses.
AVERAGE_RANDOM_FETCH_PAGES	DOUBLE	No	No	Average number of random table pages between sequential page accesses when fetching using the index.
NUMRIDS	BIGINT	No	No	Total number of row identifiers in the index.
NUMRIDS_DELETED	BIGINT	No	No	Total number of psuedo-deleted row identifiers in the index.
NUM_EMPTY_LEAFS	BIGINT	No	No	Total number of empty leaf pages in the index.
ACTIVE_BLOCKS	BIGINT	No	No	Total number of active multidimensional clustering (MDC) blocks in the table.
NUM_DATA_PART	INTEGER	No	No	Number of data partitions for a partitioned table. Set to 1 if the table is not partitioned.

Table 217. Possible OBJECT\_TYPE Values

Value	Description
IX	Index
NK	Nickname
RX	RCT Index
DP_TABLE	Data partitioned table
TA	Table
TF	Table Function
+A	Compiler-referenced Alias
+C	Compiler-referenced Constraint
+F	Compiler-referenced Function
+G	Compiler-referenced Trigger
+N	Compiler-referenced Nickname
+T	Compiler-referenced Table
+V	Compiler-referenced View

## EXPLAIN\_OPERATOR table

The EXPLAIN\_OPERATOR table contains all the operators needed to satisfy the query statement by the query compiler.

**EXPLAIN\_OPERATOR table**

Table 218. EXPLAIN\_OPERATOR Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	FK	Statement number within package to which this explain information is related.
SECTNO	INTEGER	No	FK	Section number within package to which this explain information is related.
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 instructions) of fetching the next row for the chosen access plan up to and including this operator.
COMM_COST	DOUBLE	No	No	Estimated cumulative communication cost (in TCP/IP frames) of 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.
BUFFERS	DOUBLE	No	No	Estimated buffer requirements for this operator and its inputs.

## EXPLAIN\_OPERATOR table

Table 218. EXPLAIN\_OPERATOR Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
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 219. OPERATOR\_TYPE values

Value	Description
DELETE	Delete
EISCAN	Extended Index Scan
FETCH	Fetch
FILTER	Filter rows
GENROW	Generate Row
GRPBY	Group By
HSJOIN	Hash Join
INSERT	Insert
IXAND	Dynamic Bitmap Index ANDing
IXSCAN	Relational index scan
MSJOIN	Merge Scan Join
NLJOIN	Nested loop Join
RETURN	Result
RIDSCN	Row Identifier (RID) Scan
RPD	Remote PushDown
SHIP	Ship query to remote system
SORT	Sort
TBSCAN	Table Scan
TEMP	Temporary Table Construction
TQ	Table Queue
UNION	Union
UNIQUE	Duplicate Elimination
UPDATE	Update
XISCAN	Index scan over XML data
XSCAN	XML document navigation scan
XANDOR	Index ANDing and ORing over XML data

## EXPLAIN\_PREDICATE table

The EXPLAIN\_PREDICATE table identifies which predicates are applied by a specific operator.

## EXPLAIN\_PREDICATE table

Table 220. EXPLAIN\_PREDICATE Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	FK	Statement number within package to which this explain information is related.
SECTNO	INTEGER	No	FK	Section number within package to which this explain information is related.
OPERATOR_ID	INTEGER	No	No	Unique ID for this operator within this query.
PREDICATE_ID	INTEGER	No	No	Unique ID for this predicate for the specified operator.  A value of "-1" is shown for operator predicates constructed by the Explain tool which are not optimizer objects and do not exist in the optimizer plan.
HOW_APPLIED	CHAR(10)	No	No	How predicate is being used by the specified operator.
WHEN_EVALUATED	CHAR(3)	No	No	Indicates when the subquery used in this predicate is evaluated.  Possible values are:  <b>blank</b> This predicate does not contain a subquery.  <b>EAA</b> 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.  <b>EAO</b> 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.  <b>MUL</b> 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\_PREDICATE table

Table 220. EXPLAIN\_PREDICATE Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
SUBQUERY	CHAR(1)	No	No	<p>Whether or not a data stream from a subquery is required for this predicate. There may be multiple subquery streams required.</p> <p>Possible values are:</p> <p><b>N</b> No subquery stream is required</p> <p><b>Y</b> One or more subquery streams is required</p>
FILTER_FACTOR	DOUBLE	No	No	<p>The estimated fraction of rows that will be qualified by this predicate.</p> <p>A value of "-1" is shown when FILTER_FACTOR is not applicable. FILTER_FACTOR is not applicable for operator predicates constructed by the Explain tool which are not optimizer objects and do not exist in the optimizer plan.</p>
PREDICATE_TEXT	CLOB(2M)	Yes	No	<p>The text of the predicate as recreated from the internal representation of the SQL or XQuery statement. If the value of a host variable, special register, or parameter marker is used during compilation of the statement, this value will appear at the end of the predicate text enclosed in a comment.</p> <p>The value will be stored in the EXPLAIN_PREDICATE table only if the statement is executed by a user who has DBADM authority, or if the DB2 registry variable DB2_VIEW_REOPT_VALUES is set to YES; otherwise, an empty comment will appear at the end of the predicate text.</p> <p>Null if not available.</p>
RANGE_NUM	INTEGER	Yes	No	<p>Range of data partition elimination predicates, which enables the grouping of predicates that are used for data partition elimination by range. Null value for all other predicate types.</p>

Table 221. Possible HOW\_APPLIED Values

Value	Description
BSARG	Evaluated as a sargable predicate once for every block
DPSTART	Start key predicate used in data partition elimination
DPSTOP	Stop key predicate used in data partition elimination
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 222. 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.

Table 223. EXPLAIN\_STATEMENT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	PK, FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	PK, FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	PK, FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	PK, FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	PK	Level of Explain information for which this row is relevant.  Valid values are: <b>O</b> Original Text (as entered by user) <b>P</b> 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.

## EXPLAIN\_STATEMENT table

Table 223. EXPLAIN\_STATEMENT Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
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.
QUERYNO	INTEGER	No	No	Numeric identifier for explained SQL statement. For dynamic SQL statements (excluding the EXPLAIN SQL statement) issued through CLP or CLI, the default value is a sequentially incremented value. Otherwise, the default value is the value of STMTNO for static SQL statements and 1 for dynamic SQL statements.
QUERYTAG	CHAR(20)	No	No	Identifier tag for each explained SQL statement. For dynamic SQL statements issued through CLP (excluding the EXPLAIN SQL statement), the default value is 'CLP'. For dynamic SQL statements issued through CLI (excluding the EXPLAIN SQL statement), the default value is 'CLI'. Otherwise, the default value used is blanks.
STATEMENT_TYPE	CHAR(2)	No	No	Descriptive label for type of query being explained.  Possible values are: <b>CL</b> Call <b>CP</b> Compound SQL (Dynamic) <b>D</b> Delete <b>DC</b> Delete where current of cursor <b>I</b> Insert <b>M</b> Merge <b>S</b> Select <b>SI</b> Set Integrity or Refresh Table <b>U</b> Update <b>UC</b> 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.  Possible values are: <b>' '</b> Not applicable (blank) <b>N</b> No <b>Y</b> Yes

## EXPLAIN\_STATEMENT table

Table 223. EXPLAIN\_STATEMENT Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
DELETABLE	CHAR(1)	No	No	Indicates if this statement is considered deletable. This is particularly relevant to SELECT statements which may be determined to be potentially deletable.  Possible values are:  ' ' Not applicable (blank)  N No  Y Yes
TOTAL_COST	DOUBLE	No	No	Estimated total cost (in timerons) of executing the chosen access plan for this statement; set to 0 (zero) if EXPLAIN_LEVEL is 0 (original text) since no access plan has been chosen at this time.
STATEMENT_TEXT	CLOB(2M)	No	No	Text or portion of the text of the SQL statement being explained. The text shown for the Plan Selection 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.  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.

Table 224. EXPLAIN\_STREAM Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.

## EXPLAIN\_STREAM table

Table 224. EXPLAIN\_STREAM Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
SOURCE_NAME	VARCHAR(128)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	FK	Statement number within package to which this explain information is related.
SECTNO	INTEGER	No	FK	Section number within package to which this explain information is related.
STREAM_ID	INTEGER	No	No	Unique ID for this data stream within the specified operator.
SOURCE_TYPE	CHAR(1)	No	No	Indicates the source of this data stream: <b>O</b> Operator <b>D</b> 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: <b>O</b> Operator <b>D</b> Data Object
TARGET_ID	SMALLINT	No	No	Unique ID for the operator within this query that is the target of this data stream. Set to -1 if TARGET_TYPE is 'D'.
OBJECT_SCHEMA	VARCHAR(128)	Yes	No	Schema to which the affected data object belongs. Set to null if both SOURCE_TYPE and TARGET_TYPE are 'O'.
OBJECT_NAME	VARCHAR(128)	Yes	No	Name of the object that is the subject of data stream. Set to null if both SOURCE_TYPE and TARGET_TYPE are 'O'.
STREAM_COUNT	DOUBLE	No	No	Estimated cardinality of data stream.
COLUMN_COUNT	SMALLINT	No	No	Number of columns in data stream.
PREDICATE_ID	INTEGER	No	No	If this stream is part of a subquery for a predicate, the predicate ID will be reflected here, otherwise the column is set to -1.
COLUMN_NAMES	CLOB(2M)	Yes	No	This column contains the names and ordering information of the columns involved in this stream.  These names will be in the format of: NAME1(A)+NAME2(D)+NAME3+NAME4  Where (A) indicates a column in ascending order, (D) indicates a column in descending order, and no ordering information indicates that either the column is not ordered or ordering is not relevant.

Table 224. EXPLAIN\_STREAM Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
PMID	SMALLINT	No	No	Distribution map ID.
SINGLE_NODE	CHAR(5)	Yes	No	Indicates whether this data stream is on a single or on multiple database partitions: <b>MULT</b> On multiple database partitions <b>COOR</b> On coordinator node <b>HASH</b> Directed using hashing <b>RID</b> Directed using the row ID <b>FUNC</b> Directed using a function (HASHEDVALUE() or DBPARTITIONNUM()) <b>CORR</b> Directed using a correlation value <b>Numeric</b> Directed to predetermined single node
PARTITION_COLUMNS	CLOB(2M)	Yes	No	List of the columns on which this data stream is distributed.
SEQUENCE_SIZES	CLOB(2M)	Yes	No	Lists the expected sequence size for XML columns, or shows "NA" (not applicable) for any non-XML columns in the data stream.  Set to null if there is not at least one XML column in the data stream.

## EXPLAIN\_STREAM table

## Appendix J. Explain register values

Following is a description of the interaction of the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values, both with each other and with the PREP and BIND commands.

With dynamic SQL, the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values interact as follows.

Table 225. Interaction of Explain Special Register Values (Dynamic SQL)

EXPLAIN SNAPSHOT values	EXPLAIN MODE values					
	NO	YES	EXPLAIN	REOPT	RECOMMEND INDEXES	EVALUATE INDEXES
NO	<ul style="list-style-type: none"> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated.</li> <li>Results of query not returned (dynamic statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated when a statement qualifies for reoptimization at execution time.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated.</li> <li>Results of query not returned (dynamic statements not executed).</li> <li>Indexes recommended.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated.</li> <li>Results of query not returned (dynamic statements not executed).</li> <li>Indexes evaluated.</li> </ul>
YES	<ul style="list-style-type: none"> <li>Explain Snapshot taken.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated.</li> <li>Explain Snapshot taken.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated.</li> <li>Explain Snapshot taken.</li> <li>Results of query not returned (dynamic statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated when a statement qualifies for reoptimization at execution time.</li> <li>Explain Snapshot taken.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated.</li> <li>Explain Snapshot taken.</li> <li>Results of query not returned (dynamic statements not executed).</li> <li>Indexes recommended.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated.</li> <li>Explain Snapshot taken.</li> <li>Results of query not returned (dynamic statements not executed).</li> <li>Indexes evaluated.</li> </ul>

## Explain register values

Table 225. Interaction of Explain Special Register Values (Dynamic SQL) (continued)

EXPLAIN SNAPSHOT values	EXPLAIN MODE values					
	NO	YES	EXPLAIN	REOPT	RECOMMEND INDEXES	EVALUATE INDEXES
EXPLAIN	<ul style="list-style-type: none"> <li>Explain Snapshot taken.</li> <li>Results of query not returned (dynamic statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated.</li> <li>Explain Snapshot taken.</li> <li>Results of query not returned (dynamic statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated.</li> <li>Explain Snapshot taken.</li> <li>Results of query not returned (dynamic statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated when a statement qualifies for reoptimization at execution time.</li> <li>Explain Snapshot taken when a statement qualifies for reoptimization at execution time.</li> <li>Results of query not returned (dynamic or incremental-bind statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated.</li> <li>Explain Snapshot taken.</li> <li>Results of query not returned (dynamic statements not executed).</li> <li>Indexes recommended.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated.</li> <li>Explain Snapshot taken.</li> <li>Results of query not returned (dynamic statements not executed).</li> <li>Indexes evaluated.</li> </ul>
REOPT	<ul style="list-style-type: none"> <li>Explain Snapshot taken when a statement qualifies for reoptimization at execution time.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated.</li> <li>Explain Snapshot taken when a statement qualifies for reoptimization at execution time.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated.</li> <li>Explain Snapshot taken when a statement qualifies for reoptimization at execution time.</li> <li>Results of query not returned (dynamic or incremental-bind statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated when a statement qualifies for reoptimization at execution time.</li> <li>Explain Snapshot taken when a statement qualifies for reoptimization at execution time.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated.</li> <li>Explain Snapshot taken when a statement qualifies for reoptimization at execution time.</li> <li>Results of query not returned (dynamic or incremental-bind statements not executed).</li> <li>Indexes recommended.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated.</li> <li>Explain Snapshot taken when a statement qualifies for reoptimization at execution time.</li> <li>Results of query not returned (dynamic or incremental-bind statements not executed).</li> <li>Indexes evaluated.</li> </ul>

The CURRENT EXPLAIN MODE special register interacts with the EXPLAIN bind option in the following way for dynamic SQL.



Table 226. Interaction of EXPLAIN Bind Option and CURRENT EXPLAIN MODE

EXPLAIN MODE values	EXPLAIN Bind option values			
	NO	YES	REOPT	ALL
NO	<ul style="list-style-type: none"> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL when statement qualifies for reoptimization at execution time.</li> <li>Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL.</li> <li>Explain tables populated for dynamic SQL.</li> <li>Results of query returned.</li> </ul>
YES	<ul style="list-style-type: none"> <li>Explain tables populated for dynamic SQL.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL.</li> <li>Explain tables populated for dynamic SQL.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL when statement qualifies for reoptimization at execution time.</li> <li>Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL.</li> <li>Explain tables populated for dynamic SQL.</li> <li>Results of query returned.</li> </ul>
EXPLAIN	<ul style="list-style-type: none"> <li>Explain tables populated for dynamic SQL.</li> <li>Results of query not returned (dynamic statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL.</li> <li>Explain tables populated for dynamic SQL.</li> <li>Results of query not returned (dynamic statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL when statement qualifies for reoptimization at execution time.</li> <li>Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time.</li> <li>Results of query not returned (dynamic statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL.</li> <li>Explain tables populated for dynamic SQL.</li> <li>Results of query not returned (dynamic statements not executed).</li> </ul>

## Explain register values

Table 226. Interaction of EXPLAIN Bind Option and CURRENT EXPLAIN MODE (continued)

EXPLAIN MODE values	EXPLAIN Bind option values			
	NO	YES	REOPT	ALL
REOPT	<ul style="list-style-type: none"> <li>Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL when statement qualifies for reoptimization at execution time.</li> <li>Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL when statement qualifies for reoptimization at execution time.</li> <li>Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL when statement qualifies for reoptimization at execution time.</li> <li>Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time.</li> <li>Results of query returned.</li> </ul>
RECOMMEND INDEXES	<ul style="list-style-type: none"> <li>Explain tables populated for dynamic SQL.</li> <li>Results of query not returned (dynamic statements not executed).</li> <li>Recommend indexes.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL.</li> <li>Explain tables populated for dynamic SQL.</li> <li>Results of query not returned (dynamic statements not executed).</li> <li>Recommend indexes.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL when statement qualifies for reoptimization at execution time.</li> <li>Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time.</li> <li>Results of query not returned (dynamic statements not executed).</li> <li>Recommend indexes.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL.</li> <li>Explain tables populated for dynamic SQL.</li> <li>Results of query not returned (dynamic statements not executed).</li> <li>Recommend indexes.</li> </ul>
EVALUATE INDEXES	<ul style="list-style-type: none"> <li>Explain tables populated for dynamic SQL.</li> <li>Results of query not returned (dynamic statements not executed).</li> <li>Evaluate indexes.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL.</li> <li>Explain tables populated for dynamic SQL.</li> <li>Results of query not returned (dynamic statements not executed).</li> <li>Evaluate indexes.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL when statement qualifies for reoptimization at execution time.</li> <li>Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time.</li> <li>Results of query not returned (dynamic statements not executed).</li> <li>Evaluate indexes.</li> </ul>	<ul style="list-style-type: none"> <li>Explain tables populated for static SQL.</li> <li>Explain tables populated for dynamic SQL.</li> <li>Results of query not returned (dynamic statements not executed).</li> <li>Evaluate indexes.</li> </ul>

The CURRENT EXPLAIN SNAPSHOT special register interacts with the EXPLSNAP bind option in the following way for dynamic SQL.

Table 227. Interaction of EXPLSNAP bind Option and CURRENT EXPLAIN SNAPSHOT

EXPLAIN SNAPSHOT values	EXPLSNAP Bind option values			
	NO	YES	REOPT	ALL
NO	<ul style="list-style-type: none"> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain Snapshot taken for static SQL.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time.</li> <li>Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain Snapshot taken for static SQL.</li> <li>Explain Snapshot taken for dynamic SQL.</li> <li>Results of query returned.</li> </ul>
YES	<ul style="list-style-type: none"> <li>Explain Snapshot taken for dynamic SQL.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain Snapshot taken for static SQL.</li> <li>Explain Snapshot taken for dynamic SQL.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time.</li> <li>Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain Snapshot taken for static SQL.</li> <li>Explain Snapshot taken for dynamic SQL.</li> <li>Results of query returned.</li> </ul>
EXPLAIN	<ul style="list-style-type: none"> <li>Explain Snapshot taken for dynamic SQL.</li> <li>Results of query not returned (dynamic statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>Explain Snapshot taken for static SQL.</li> <li>Explain Snapshot taken for dynamic SQL.</li> <li>Results of query not returned (dynamic statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time.</li> <li>Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time.</li> <li>Results of query not returned (dynamic statements not executed).</li> </ul>	<ul style="list-style-type: none"> <li>Explain Snapshot taken for static SQL.</li> <li>Explain Snapshot taken for dynamic SQL.</li> <li>Results of query not returned (dynamic statements not executed).</li> </ul>

## Explain register values

Table 227. Interaction of EXPLSNAP bind Option and CURRENT EXPLAIN SNAPSHOT (continued)

EXPLAIN SNAPSHOT values	EXPLSNAP Bind option values			
	NO	YES	REOPT	ALL
REOPT	<ul style="list-style-type: none"> <li>Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time.</li> <li>Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time.</li> <li>Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time.</li> <li>Results of query returned.</li> </ul>	<ul style="list-style-type: none"> <li>Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time.</li> <li>Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time.</li> <li>Results of query returned.</li> </ul>

---

## Appendix K. Exception tables

Exception tables are user-created tables that mimic the definition of the tables that are specified to be checked using the SET INTEGRITY statement with the IMMEDIATE CHECKED option. They are used to store copies of the rows that violate constraints in the tables being checked.

The exception tables that are used by the load utility are identical to the ones described here, and can therefore be reused during checking with the SET INTEGRITY statement.

### Rules for creating an exception table

The rules for creating an exception table are as follows:

- If the table is protected by a security policy, the exception table must be protected by the same security policy.
- The first “n” columns of the exception table are the same as the columns of the table being checked. All column attributes, including name, data type, and length should be identical. For protected columns, the security label protecting the column must be the same in both tables.
- All of the columns of the exception table must be free of constraints and triggers. Constraints include referential integrity and check constraints, as well as unique index constraints that could cause errors on insert.
- The “(n+1)” column of the exception table is an optional TIMESTAMP column. This serves to identify successive invocations of checking by the SET INTEGRITY statement on the same table, if the rows within the exception table have not been deleted before issuing the SET INTEGRITY statement to check the data.
- The “(n+2)” column should be of type CLOB(32K) or larger. This column is optional but recommended, and will be used to give the names of the constraints that the data within the row violates. If this column is not provided (as could be warranted if, for example, the original table had the maximum number of columns allowed), then only the row where the constraint violation was detected is copied.
- The exception table should be created with both “(n+1)” and “(n+2)” columns.
- There is no enforcement of any particular name for the above additional columns. However, the type specification must be exactly followed.
- No additional columns are allowed.
- If the original table has generated columns (including the IDENTITY property), the corresponding columns in the exception table should not specify the generated property.
- Users invoking the SET INTEGRITY statement to check data must hold the INSERT privilege on the exception tables.
- The exception table cannot be a data partitioned table, a range clustered table, or a detached table.
- The exception table cannot be a materialized query table or a staging table.
- The exception table cannot have any dependent refresh immediate materialized query tables or any dependent propagate immediate staging tables.

The information in the “message” column has the following structure:

## Exception tables

Table 228. Exception Table Message Column Structure

Field number	Contents	Size	Comments
1	Number of constraint violations	5 bytes	Right justified padded with '0'
2	Type of first constraint violation	1 byte	'K' - Check Constraint violation 'F' - Foreign Key violation 'G' - Generated Column violation 'T' - Unique Index violation <sup>a</sup> 'D' - Delete Cascade violation 'P' - Data Partitioning violation 'S' - Invalid Row Security Label 'L' - DB2 LBAC Write rules violation
3	Length of constraint/column <sup>b</sup> /index ID <sup>c</sup>	5 bytes	Right justified padded with '0'
4	Constraint name/Column name <sup>b</sup> /index ID <sup>c</sup>	length from the previous field	
5	Separator	3 bytes	<space><colon><space>
6	Type of next constraint violation	1 byte	'K' - Check Constraint violation 'F' - Foreign Key violation 'G' - Generated Column violation 'T' - Unique Index violation 'D' - Delete Cascade violation 'P' - Data Partitioning violation 'S' - Invalid Row Security Label 'L' - DB2 LBAC Write rules violation
7	Length of constraint/column/index ID	5 bytes	Right justified padded with '0'
8	Constraint name/Column name/Index ID	length from the previous field	
.....	.....	.....	Repeat Field 5 through 8 for each violation

- <sup>a</sup> Unique index violations will not occur during checking using the SET INTEGRITY statement, unless it is after an attach operation. This will be reported, however, when running LOAD if the FOR EXCEPTION option is chosen. LOAD, on the other hand, will not report check constraint, generated column, foreign key, delete cascade, or data partitioning violations in the exception tables.
- <sup>b</sup> To retrieve the expression of a generated column from the catalog views, use a select statement. For example, if field 4 is MYSCHEMA.MYTABLE.GEN\_1, then SELECT SUBSTR(TEXT, 1, 50) FROM SYSCAT.COLUMNS WHERE TABSCHEMA='MYSCHEMA' AND TABNAME='MYNAME' AND COLNAME='GEN\_1'; will return the first fifty bytes of the expression, in the form "AS (<expression>)"
- <sup>c</sup> To retrieve an index ID from the catalog views, use a select statement. For example, if field 4 is 1234, then SELECT INDSHEMA, INDNAME FROM SYSCAT.INDEXES WHERE IID=1234.

### Handling rows in an exception table

The information in exception tables can be processed in various ways. Data can be corrected and rows re-inserted into the original tables.

If there are no INSERT triggers on the original table, transfer the corrected rows by issuing an INSERT statement with a subquery on the exception table.

If there are INSERT triggers, and you want to complete the load operation with the corrected rows from exception tables without firing the triggers:

- Design the INSERT triggers to be fired depending on the value in a column that has been defined explicitly for the purpose.

- Unload data from the exception tables and append it using the load utility. In this case, if you want to recheck the data, note that constraints checking is not confined to the appended rows.
- Save the trigger definition text from the relevant system catalog view. Then drop the INSERT trigger and use INSERT to transfer the corrected rows from the exception tables. Finally, recreate the trigger using the saved trigger definition.

No explicit provision is made to prevent the firing of triggers when inserting rows from exception tables.

Only one violation per row is reported for unique index violations.

If values with long string or LOB data types are in the table, the values are not inserted into the exception table in the case of unique index violations.

### Querying exception tables

The message column structure in an exception table is a concatenated list of constraint names, lengths, and delimiters, as described earlier. This information can be queried.

For example, to retrieve a list of all violations, repeating each row with only the constraint name, assume that the original table T1 had two columns, C1 and C2. Assume also, that the corresponding exception table, E1, has columns C1 and C2, corresponding to those in T1, as well as a message column, MSGCOL. The following query uses recursion to list one constraint name per row (repeating rows that have more than one violation):

```
WITH IV (C1, C2, MSGCOL, CONSTNAME, I, J) AS
 (SELECT C1, C2, MSGCOL,
 CHAR(SUBSTR(MSGCOL, 12,
 INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0)))),
 1,
 15+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0))
 FROM E1
 UNION ALL
 SELECT C1, C2, MSGCOL,
 CHAR(SUBSTR(MSGCOL, J+6,
 INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0)))),
 I+1,
 J+9+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0))
 FROM IV
 WHERE I < INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,1,5)),5,0))
) SELECT C1, C2, CONSTNAME FROM IV;
```

To list all of the rows that violated a particular constraint, the previous query could be extended as follows:

```
WITH IV (C1, C2, MSGCOL, CONSTNAME, I, J) AS
 (SELECT C1, C2, MSGCOL,
 CHAR(SUBSTR(MSGCOL, 12,
 INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0)))),
 1,
 15+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0))
 FROM E1
 UNION ALL
 SELECT C1, C2, MSGCOL,
 CHAR(SUBSTR(MSGCOL, J+6,
 INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0)))),
 I+1,
 J+9+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0))
```

## Exception tables

```
FROM IV
WHERE I < INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,1,5)),5,0))
) SELECT C1, C2, CONSTNAME FROM IV WHERE CONSTNAME = 'constraintname';
```

The following query could be used to obtain all of the check constraint violations:

```
WITH IV (C1, C2, MSGCOL, CONSTNAME, CONSTTYPE, I, J) AS
(SELECT C1, C2, MSGCOL,
CHAR(SUBSTR(MSGCOL, 12,
INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0)))),
CHAR(SUBSTR(MSGCOL, 6, 1)),
1,
15+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0))
FROM E1
UNION ALL
SELECT C1, C2, MSGCOL,
CHAR(SUBSTR(MSGCOL, J+6,
INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0)))),
CHAR(SUBSTR(MSGCOL, J, 1)),
I+1,
J+9+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0))
FROM IV
WHERE I < INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,1,5)),5,0))
) SELECT C1, C2, CONSTNAME FROM IV WHERE CONSTTYPE = 'K';
```



## Appendix L. SQL statements allowed in routines

The following table indicates whether or not an SQL statement (specified in the first column) is allowed to execute in a routine with the specified SQL data access indication. If an executable SQL statement is encountered in a routine defined with NO SQL, SQLSTATE 38001 is returned. For other execution contexts, SQL statements that are not supported in any context return SQLSTATE 38003. For other SQL statements not allowed in a CONTAINS SQL context, SQLSTATE 38004 is returned. In a READS SQL DATA context, SQLSTATE 38002 is returned. During creation of an SQL routine, a statement that does not match the SQL data access indication will cause SQLSTATE 42985 to be returned.

If a statement invokes a routine, the effective SQL data access indication for the statement will be the greater of:

- The SQL data access indication of the statement from the following table.
- The SQL data access indication of the routine specified when the routine was created.

For example, the CALL statement has an SQL data access indication of CONTAINS SQL. However, if a stored procedure defined as READS SQL DATA is called, the effective SQL data access indication for the CALL statement is READS SQL DATA.

When a routine invokes an SQL statement, the effective SQL data access indication for the statement must not exceed the SQL data access indication declared for the routine. For example, a function defined as READS SQL DATA could not call a stored procedure defined as MODIFIES SQL DATA.

*Table 229. SQL Statement and SQL Data Access Indication*

SQL Statement	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
ALTER...	N	N	N	Y
AUDIT	N	N	N	Y
BEGIN DECLARE SECTION	Y(1)	Y	Y	Y
CALL	N	Y	Y	Y
CLOSE	N	N	Y	Y
COMMENT ON	N	N	N	Y
COMMIT	N	N(4)	N(4)	N(4)
COMPOUND SQL	N	Y	Y	Y
CONNECT(2)	N	N	N	N
CREATE	N	N	N	Y
DECLARE CURSOR	Y(1)	Y	Y	Y
DECLARE GLOBAL TEMPORARY TABLE	N	N	N	Y
DELETE	N	N	N	Y
DESCRIBE	N	Y	Y	Y
DISCONNECT(2)	N	N	N	N
DROP ...	N	N	N	Y

## SQL statements allowed in routines

Table 229. SQL Statement and SQL Data Access Indication (continued)

SQL Statement	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
END DECLARE SECTION	Y(1)	Y	Y	Y
EXECUTE	N	Y(3)	Y(3)	Y
EXECUTE IMMEDIATE	N	Y(3)	Y(3)	Y
EXPLAIN	N	N	N	Y
FETCH	N	N	Y	Y
FREE LOCATOR	N	Y	Y	Y
FLUSH EVENT MONITOR	N	N	N	Y
GRANT ...	N	N	N	Y
INCLUDE	Y(1)	Y	Y	Y
INSERT	N	N	N	Y
LOCK TABLE	N	Y	Y	Y
MERGE	N	N	N	Y
OPEN	N	N	Y(5)	Y
PREPARE	N	Y	Y	Y
REFRESH TABLE	N	N	N	Y
RELEASE CONNECTION(2)	N	N	N	N
RELEASE SAVEPOINT	N	N	N	Y
RENAME TABLE	N	N	N	Y
REVOKE ...	N	N	N	Y
ROLLBACK	N	N(4)	N(4)	N(4)
ROLLBACK TO SAVEPOINT	N	N	N	Y
SAVEPOINT	N	N	N	Y
SELECT INTO	N	N	Y(5)	Y
SET CONNECTION(2)	N	N	N	N
SET INTEGRITY	N	N	N	Y
SET special register	N	Y	Y	Y
SET variable	N	Y(6)	Y(5)	Y
TRANSFER OWNERSHIP	N	N	N	Y
UPDATE	N	N	N	Y
VALUES INTO	N	N	Y	Y
WHENEVER	Y(1)	Y	Y	Y

**Note:**

1. Although the NO SQL option implies that no SQL statements can be specified, non-executable statements are not restricted.
2. Connection management statements are not allowed in any routine execution context.
3. It depends on the statement being executed. The statement specified for the EXECUTE statement must be a statement that is allowed in the context of the

particular SQL access level in effect. For example, if the SQL access level READS SQL DATA is in effect, the statement cannot be INSERT, UPDATE, or DELETE.

4. The COMMIT statement and the ROLLBACK statement without the TO SAVEPOINT clause can be used in a stored procedure, but only if the stored procedure is called directly from an application, or indirectly through nested stored procedure calls from an application. (If any trigger, function, method, or atomic compound statement is in the call chain to the stored procedure, COMMIT or ROLLBACK of a unit of work is not allowed.)
5. If the SQL access level READS SQL DATA is in effect, no SQL data change statement can be embedded in the SELECT INTO statement, in the cursor referenced by the OPEN statement, or the right hand side expression of the SET variable statement.
6. If the SQL access level CONTAINS SQL is in effect, no scalar fullselect can be embedded in the right hand side expression of the SET variable statement.

## SQL statements allowed in routines

---

## Appendix M. CALL invoked from a compiled statement

Invokes a procedure stored at the location of a database. A 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 procedure are on the same database partition, the stored procedure is executed locally.

**Note:** This form of the CALL statement is deprecated, and is only being provided for compatibility with previous versions of DB2.

### Invocation

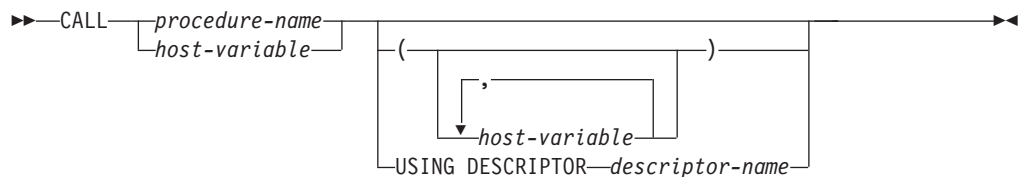
This form of the CALL statement can only be embedded in an application program that is precompiled with the CALL\_RESOLUTION DEFERRED option. It cannot invoke a federated procedure. It cannot be used in triggers, SQL procedures, or any other non-application contexts. It is an executable statement that cannot be dynamically prepared. However, the procedure name can be specified through 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, which achieves an effect similar to that of a dynamically prepared statement.

### Authorization

The privileges held by the authorization ID of the statement *at run time* must include at least one of the following:

- EXECUTE privilege on the package that is associated with the procedure; EXECUTE privilege on the procedure is not checked
- CONTROL privilege on the package that is associated with the procedure
- SYSADM or DBADM authority

### Syntax



### 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 that is not greater than 254 bytes. Because this can only be an ordinary identifier, it cannot

## CALL invoked from a compiled statement

contain blanks or special characters. The value is converted to uppercase. If it is necessary to use lowercase names, blanks, or special characters, the name must be specified via a *host-variable*.

If *host-variable* is specified, it must be a CHAR or VARCHAR variable with a length attribute that is not greater than 254 bytes, and it must not include an indicator variable. The value is *not* converted to uppercase. The character string must be left-justified.

The procedure name can take one of several forms:

### *procedure-name*

The name (with no extension) of the procedure to execute. The procedure that is invoked is determined as follows.

1. The *procedure-name* is used to search the defined procedures (in SYSCAT.ROUTINES) for a matching procedure. A matching procedure is determined using the steps that follow.
  - a. Find the procedures (ROUTINETYPE is 'P') from the catalog (SYSCAT.ROUTINES), where the ROUTINENAME matches the specified *procedure-name*, and the ROUTINESHEMA 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.

If a procedure is selected, DB2 will invoke the procedure defined by the external name.

2. If no matching procedure was found, *procedure-name* is used both as the name of the procedure library, and the function name within that library. For example, if *procedure-name* is `proclib`, the DB2 server will load the procedure library named `proclib` and execute the function routine `proclib()` within that library.

On UNIX systems, the default directory for procedure libraries is `sqllib/function`. The default directory for unfenced procedures is `sqllib/function/unfenced`.

In Windows-based systems, the default directory for procedure libraries is `sqllib\function`. The default directory for unfenced procedures is `sqllib\function\unfenced`.

If the library or function could not be found, an error is returned (SQLSTATE 42884).

### *procedure-library!function-name*

The exclamation character (!) acts as a delimiter between the library name and the function name of the procedure. For example, if `proclib!func` is specified, `proclib` is loaded into memory, and the function `func` from that library is executed. This allows multiple functions to be placed in the same procedure library.

The 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.

## CALL invoked from a compiled statement

On a UNIX system, for example, if `/u/terry/proclib!func` is specified, the procedure library `proclib` is obtained from the directory `/u/terry`, and the function `func` from that library is executed.

In all of these cases, the total length of the procedure name, including its implicit or explicit full path, must not be longer than 254 bytes.

*(host-variable,...)*

Each specification of *host-variable* is a parameter of the CALL statement. The *n*th parameter of the CALL corresponds to the *n*th parameter of the server's procedure.

Each *host-variable* is assumed to be used for exchanging data in both directions between client and server. 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 procedure. The 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 9.1 database server, the parameters must have matching data types in both the client and server program.

### USING DESCRIPTOR *descriptor-name*

Identifies an SQLDA that must contain a valid description of host variables. The *n*th SQLVAR element corresponds to the *n*th parameter of the server's 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:

- LEN.SQLLONGLEN
- SQLDATALEN
- SQLDATATYPE\_NAME

The SQLDA is assumed to be used for exchanging data in both directions between client and server. 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 procedure. The procedure should set the SQLIND field -128 for any parameter that is not used to return data to the client application.

## CALL invoked from a compiled statement

### 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 have been supported by procedures since DB2 Version 2. The LOB data types are not supported by all down level clients or servers.

- *Retrieving the DB2\_RETURN\_STATUS from an SQL procedure:*

If an SQL procedure successfully issues a RETURN statement with a status value, this value is returned in the first SQLERRD field of the SQLCA. If the CALL statement is issued in an SQL procedure, use the GET DIAGNOSTICS statement to retrieve the DB2\_RETURN\_STATUS value. The value is -1 if the SQLSTATE indicates an error.

- *Returning result sets from procedures:*

If the client application program is written using CLI, result sets can be returned directly to the client application. The procedure indicates that a result set is to be returned by declaring a cursor on that result set, opening a cursor on the result set, and leaving the cursor open when exiting the procedure.

At the end of a procedure:

- For every cursor that has been left open, a result set is returned to the 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 procedure at the time the procedure is terminated, rows 151 through 500 will be returned to the procedure.

- *Handling of special registers:*

The settings of special registers for the caller are inherited by the procedure on invocation, and restored upon return to the caller. Special registers may be changed within a procedure, but these changes do not affect the caller. This is not true for legacy procedures (those defined with parameter style DB2DARI, or found in the default library), where the changes made to special registers in a procedure become the settings for the caller.

- *Compatibilities*

There is a newer, preferred, form of the CALL statement that can be embedded in an application (by precompiling the application with the CALL\_RESOLUTION IMMEDIATE option), or that can be dynamically prepared.

### Examples

*Example 1:*

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.



## CALL invoked from a compiled statement

```
struct sqllda *INOUT_SQLDA;
/* Setup code for SQLDA variables goes here */
CALL :SALARY_PROC
USING DESCRIPTOR :*INOUT_SQLDA;
```

Example 3:

A Java procedure is defined in the database, using the following statement:

```
CREATE PROCEDURE PARTS_ON_HAND (IN PARTNUM INTEGER,
 OUT COST DECIMAL(7,2),
 OUT QUANTITY INTEGER)
 EXTERNAL NAME 'parts!onhand'
 LANGUAGE JAVA
 PARAMETER STYLE DB2GENERAL;
```

A Java application calls this procedure using the following code fragment:

```
...
CallableStatement stpCall;

String sql = "CALL PARTS_ON_HAND (?,?,?)";

stpCall = con.prepareStatement(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*, because the procedure name specified on the CALL statement is found in the database and has the external name 'parts!onhand'.



---

## Appendix N. Overview of the DB2 technical information

DB2 technical information is available through the following tools and methods:

- *DB2 Information Center*
  - Topics (Task, concept and reference topics)
  - Help for DB2 tools
  - Sample programs
  - Tutorials
- DB2 books
  - PDF files (downloadable)
  - PDF files (from the DB2 PDF DVD)
  - printed books
- Command line help
  - Command help
  - Message help

**Note:** The *DB2 Information Center* topics are updated more frequently than either the PDF or the hard-copy books. To get the most current information, install the documentation updates as they become available, or refer to the *DB2 Information Center* at [ibm.com](http://ibm.com)<sup>®</sup>.

You can access additional DB2 technical information such as technotes, white papers, and IBM Redbooks<sup>®</sup> publications online at [ibm.com](http://www.ibm.com). Access the DB2 Information Management software library site at <http://www.ibm.com/software/data/sw-library/>.

### Documentation feedback

We value your feedback on the DB2 documentation. If you have suggestions for how to improve the DB2 documentation, send an email to [db2docs@ca.ibm.com](mailto:db2docs@ca.ibm.com). The DB2 documentation team reads all of your feedback, but cannot respond to you directly. Provide specific examples wherever possible so that we can better understand your concerns. If you are providing feedback on a specific topic or help file, include the topic title and URL.

Do not use this email address to contact DB2 Customer Support. If you have a DB2 technical issue that the documentation does not resolve, contact your local IBM service center for assistance.

If you would like to help IBM make the IBM Information Management products easier to use, take the Consumability Survey: <http://www.ibm.com/software/data/info/consumability-survey/>.

---

## DB2 technical library in hardcopy or PDF format

The following tables describe the DB2 library available from the IBM Publications Center at [www.ibm.com/shop/publications/order](http://www.ibm.com/shop/publications/order). English DB2 Version 9.5 manuals in PDF format and translated versions can be downloaded from [www.ibm.com/support/docview.wss?rs=71&uid=swg2700947](http://www.ibm.com/support/docview.wss?rs=71&uid=swg2700947).

## DB2 technical library in hardcopy or PDF format

Although the tables identify books available in print, the books might not be available in your country or region.

The form number increases each time a manual is updated. Ensure that you are reading the most recent version of the manuals, as listed below.

**Note:** The *DB2 Information Center* is updated more frequently than either the PDF or the hard-copy books.

*Table 230. DB2 technical information*

<b>Name</b>	<b>Form Number</b>	<b>Available in print</b>	<b>Last updated</b>
<i>Administrative API Reference</i>	SC23-5842-02	Yes	April, 2009
<i>Administrative Routines and Views</i>	SC23-5843-02	No	April, 2009
<i>Call Level Interface Guide and Reference, Volume 1</i>	SC23-5844-02	Yes	April, 2009
<i>Call Level Interface Guide and Reference, Volume 2</i>	SC23-5845-02	Yes	April, 2009
<i>Command Reference</i>	SC23-5846-02	Yes	April, 2009
<i>Data Movement Utilities Guide and Reference</i>	SC23-5847-02	Yes	April, 2009
<i>Data Recovery and High Availability Guide and Reference</i>	SC23-5848-02	Yes	April, 2009
<i>Data Servers, Databases, and Database Objects Guide</i>	SC23-5849-02	Yes	April, 2009
<i>Database Security Guide</i>	SC23-5850-02	Yes	April, 2009
<i>Developing ADO.NET and OLE DB Applications</i>	SC23-5851-02	Yes	April, 2009
<i>Developing Embedded SQL Applications</i>	SC23-5852-02	Yes	April, 2009
<i>Developing Java Applications</i>	SC23-5853-02	Yes	April, 2009
<i>Developing Perl and PHP Applications</i>	SC23-5854-02	No	April, 2009
<i>Developing User-defined Routines (SQL and External)</i>	SC23-5855-02	Yes	April, 2009
<i>Getting Started with Database Application Development</i>	GC23-5856-02	Yes	April, 2009
<i>Getting Started with DB2 installation and administration on Linux and Windows</i>	GC23-5857-02	Yes	April, 2009
<i>Internationalization Guide</i>	SC23-5858-02	Yes	April, 2009

## DB2 technical library in hardcopy or PDF format

Table 230. DB2 technical information (continued)

Name	Form Number	Available in print	Last updated
<i>Message Reference, Volume 1</i>	GI11-7855-01	No	April, 2009
<i>Message Reference, Volume 2</i>	GI11-7856-01	No	April, 2009
<i>Migration Guide</i>	GC23-5859-02	Yes	April, 2009
<i>Net Search Extender Administration and User's Guide</i>	SC23-8509-02	Yes	April, 2009
<i>Partitioning and Clustering Guide</i>	SC23-5860-02	Yes	April, 2009
<i>Query Patroller Administration and User's Guide</i>	SC23-8507-01	Yes	April, 2009
<i>Quick Beginnings for IBM Data Server Clients</i>	GC23-5863-02	No	April, 2009
<i>Quick Beginnings for DB2 Servers</i>	GC23-5864-02	Yes	April, 2009
<i>Spatial Extender and Geodetic Data Management Feature User's Guide and Reference</i>	SC23-8508-02	Yes	April, 2009
<i>SQL Reference, Volume 1</i>	SC23-5861-02	Yes	April, 2009
<i>SQL Reference, Volume 2</i>	SC23-5862-02	Yes	April, 2009
<i>System Monitor Guide and Reference</i>	SC23-5865-02	Yes	April, 2009
<i>Text Search Guide</i>	SC23-5866-01	Yes	April, 2009
<i>Troubleshooting Guide</i>	GI11-7857-02	No	April, 2009
<i>Tuning Database Performance</i>	SC23-5867-02	Yes	April, 2009
<i>Visual Explain Tutorial</i>	SC23-5868-00	No	
<i>What's New</i>	SC23-5869-02	Yes	April, 2009
<i>Workload Manager Guide and Reference</i>	SC23-5870-02	Yes	April, 2009
<i>pureXML Guide</i>	SC23-5871-02	Yes	April, 2009
<i>XQuery Reference</i>	SC23-5872-02	No	April, 2009

Table 231. DB2 Connect-specific technical information

Name	Form Number	Available in print	Last updated
<i>Quick Beginnings for DB2 Connect Personal Edition</i>	GC23-5839-02	Yes	April, 2009
<i>Quick Beginnings for DB2 Connect Servers</i>	GC23-5840-02	Yes	April, 2009
<i>DB2 Connect User's Guide</i>	SC23-5841-02	Yes	April, 2009

## DB2 technical library in hardcopy or PDF format

Table 232. Information Integration technical information

Name	Form Number	Available in print	Last updated
Information Integration: Administration Guide for Federated Systems	SC19-1020-01	Yes	March, 2008
Information Integration: ASNCLP Program Reference for Replication and Event Publishing	SC19-1018-02	Yes	March, 2008
Information Integration: Configuration Guide for Federated Data Sources	SC19-1034-01	No	
Information Integration: SQL Replication Guide and Reference	SC19-1030-01	Yes	March, 2008
Information Integration: Introduction to Replication and Event Publishing	SC19-1028-01	Yes	March, 2008

## Ordering printed DB2 books

If you require printed DB2 books, you can buy them online in many but not all countries or regions. You can always order printed DB2 books from your local IBM representative. Keep in mind that some softcopy books on the *DB2 PDF Documentation DVD* are unavailable in print. For example, neither volume of the *DB2 Message Reference* is available as a printed book.

Printed versions of many of the DB2 books available on the DB2 PDF Documentation DVD can be ordered for a fee from IBM. Depending on where you are placing your order from, you may be able to order books online, from the IBM Publications Center. If online ordering is not available in your country or region, you can always order printed DB2 books from your local IBM representative. Note that not all books on the DB2 PDF Documentation DVD are available in print.

**Note:** The most up-to-date and complete DB2 documentation is maintained in the DB2 Information Center at <http://publib.boulder.ibm.com/infocenter/db2luw/v9r5>.

To order printed DB2 books:

- To find out whether you can order printed DB2 books online in your country or region, check the IBM Publications Center at <http://www.ibm.com/shop/publications/order>. You must select a country, region, or language to access publication ordering information and then follow the ordering instructions for your location.
- To order printed DB2 books from your local IBM representative:
  1. Locate the contact information for your local representative from one of the following Web sites:
    - The IBM directory of world wide contacts at [www.ibm.com/planetwide](http://www.ibm.com/planetwide)
    - The IBM Publications Web site at <http://www.ibm.com/shop/publications/order>. You will need to select your country, region, or

language to the access appropriate publications home page for your location. From this page, follow the "About this site" link.

2. When you call, specify that you want to order a DB2 publication.
3. Provide your representative with the titles and form numbers of the books that you want to order. For titles and form numbers, see "DB2 technical library in hardcopy or PDF format" on page 755.

## Displaying SQL state help from the command line processor

DB2 returns an SQLSTATE value for conditions that could be the result of an SQL statement. SQLSTATE help explains the meanings of SQL states and SQL state class codes.

To invoke SQL state help, open the command line processor and enter:

```
? sqlstate or ? class code
```

where *sqlstate* represents a valid five-digit SQL state and *class code* represents the first two digits of the SQL state.

For example, ? 08003 displays help for the 08003 SQL state, and ? 08 displays help for the 08 class code.

## Accessing different versions of the DB2 Information Center

For DB2 Version 9.5 topics, the DB2 Information Center URL is <http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/>

For DB2 Version 9 topics, the DB2 Information Center URL is <http://publib.boulder.ibm.com/infocenter/db2luw/v9/>

For DB2 Version 8 topics, go to the Version 8 Information Center URL at: <http://publib.boulder.ibm.com/infocenter/db2luw/v8/>

## Displaying topics in your preferred language in the DB2 Information Center

The DB2 Information Center attempts to display topics in the language specified in your browser preferences. If a topic has not been translated into your preferred language, the DB2 Information Center displays the topic in English.

- To display topics in your preferred language in the Internet Explorer browser:
  1. In Internet Explorer, click the **Tools** —> **Internet Options** —> **Languages...** button. The Language Preferences window opens.
  2. Ensure your preferred language is specified as the first entry in the list of languages.
    - To add a new language to the list, click the **Add...** button.

**Note:** Adding a language does not guarantee that the computer has the fonts required to display the topics in the preferred language.

    - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
  3. Clear the browser cache and then refresh the page to display the DB2 Information Center in your preferred language.
- To display topics in your preferred language in a Firefox or Mozilla browser:

## Displaying topics in your preferred language in the DB2 Information Center

1. Select the button in the **Languages** section of the **Tools** —> **Options** —> **Advanced** dialog. The Languages panel is displayed in the Preferences window.
2. Ensure your preferred language is specified as the first entry in the list of languages.
  - To add a new language to the list, click the **Add...** button to select a language from the Add Languages window.
  - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
3. Clear the browser cache and then refresh the page to display the DB2 Information Center in your preferred language.

On some browser and operating system combinations, you might have to also change the regional settings of your operating system to the locale and language of your choice.

---

## Updating the DB2 Information Center installed on your computer or intranet server

If you have installed the DB2 Information Center locally, you can obtain and install documentation updates from IBM.

Updating your locally-installed *DB2 Information Center* requires that you:

1. Stop the *DB2 Information Center* on your computer, and restart the Information Center in stand-alone mode. Running the Information Center in stand-alone mode prevents other users on your network from accessing the Information Center, and allows you to apply updates. Non-Administrative and Non-Root *DB2 Information Centers* always run in stand-alone mode. .
2. Use the update feature to see what updates are available. If there are updates that you would like to install, you can use the update feature to obtain and install them.

**Note:** If your environment requires installing the *DB2 Information Center* updates on a machine that is not connected to the internet, you have to mirror the update site to a local file system using a machine that is connected to the internet and has the *DB2 Information Center* installed. If many users on your network will be installing the documentation updates, you can reduce the time required for individuals to perform the updates by also mirroring the update site locally and creating a proxy for the update site.

If update packages are available, use the update feature to get the packages. However, the update feature is only available in stand-alone mode.

3. Stop the stand-alone Information Center, and restart the *DB2 Information Center* on your computer.

**Note:** On Windows Vista, the commands listed below must be run as an administrator. To launch a command prompt or graphical tool with full administrator privileges, right-click on the shortcut and then select **Run as administrator**.

To update the *DB2 Information Center* installed on your computer or intranet server:

1. Stop the *DB2 Information Center*.
  - On Windows, click **Start** → **Control Panel** → **Administrative Tools** → **Services**. Then right-click on **DB2 Information Center** service and select **Stop**.



## Updating the DB2 Information Center installed on your computer or intranet server

- On Linux, enter the following command:  
`/etc/init.d/db2icdv95 stop`
  - 2. Start the Information Center in stand-alone mode.
    - On Windows:
      - a. Open a command window.
      - b. Navigate to the path where the Information Center is installed. By default, the *DB2 Information Center* is installed in the `Program_files\IBM\DB2 Information Center\Version 9.5` directory, where `Program_files` represents the location of the Program Files directory.
      - c. Navigate from the installation directory to the `doc\bin` directory.
      - d. Run the `help_start.bat` file:  
`help_start.bat`
    - On Linux:
      - a. Navigate to the path where the Information Center is installed. By default, the *DB2 Information Center* is installed in the `/opt/ibm/db2ic/V9.5` directory.
      - b. Navigate from the installation directory to the `doc/bin` directory.
      - c. Run the `help_start` script:  
`help_start`
- The systems default Web browser launches to display the stand-alone Information Center.
3. Click the **Update** button (🔧). On the right hand panel of the Information Center, click **Find Updates**. A list of updates for existing documentation displays.
  4. To initiate the installation process, check the selections you want to install, then click **Install Updates**.
  5. After the installation process has completed, click **Finish**.
  6. Stop the stand-alone Information Center:
    - On Windows, navigate to the installation directory's `doc\bin` directory, and run the `help_end.bat` file:  
`help_end.bat`

**Note:** The `help_end` batch file contains the commands required to safely terminate the processes that were started with the `help_start` batch file. Do not use `Ctrl-C` or any other method to terminate `help_start.bat`.

    - On Linux, navigate to the installation directory's `doc/bin` directory, and run the `help_end` script:  
`help_end`

**Note:** The `help_end` script contains the commands required to safely terminate the processes that were started with the `help_start` script. Do not use any other method to terminate the `help_start` script.
  7. Restart the *DB2 Information Center*.
    - On Windows, click **Start** → **Control Panel** → **Administrative Tools** → **Services**. Then right-click on **DB2 Information Center** service and select **Start**.
    - On Linux, enter the following command:  
`/etc/init.d/db2icdv95 start`

The updated *DB2 Information Center* displays the new and updated topics.

### DB2 tutorials

The DB2 tutorials help you learn about various aspects of DB2 products. Lessons provide step-by-step instructions.

#### Before you begin

You can view the XHTML version of the tutorial from the Information Center at <http://publib.boulder.ibm.com/infocenter/db2help/>.

Some lessons use sample data or code. See the tutorial for a description of any prerequisites for its specific tasks.

#### DB2 tutorials

To view the tutorial, click on the title.

##### **“pureXML™” in *pureXML Guide***

Set up a DB2 database to store XML data and to perform basic operations with the native XML data store.

##### **“Visual Explain” in *Visual Explain Tutorial***

Analyze, optimize, and tune SQL statements for better performance using Visual Explain.

---

### DB2 troubleshooting information

A wide variety of troubleshooting and problem determination information is available to assist you in using DB2 database products.

#### DB2 documentation

Troubleshooting information can be found in the DB2 Troubleshooting Guide or the Database fundamentals section of the DB2 Information Center. There you will find information on how to isolate and identify problems using DB2 diagnostic tools and utilities, solutions to some of the most common problems, and other advice on how to solve problems you might encounter with your DB2 database products.

#### DB2 Technical Support Web site

Refer to the DB2 Technical Support Web site if you are experiencing problems and want help finding possible causes and solutions. The Technical Support site has links to the latest DB2 publications, TechNotes, Authorized Program Analysis Reports (APARs or bug fixes), fix packs, and other resources. You can search through this knowledge base to find possible solutions to your problems.

Access the DB2 Technical Support Web site at [http://www.ibm.com/software/data/db2/support/db2\\_9/](http://www.ibm.com/software/data/db2/support/db2_9/)

---

### Terms and Conditions

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

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

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

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

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

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

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



---

## Appendix O. Notices

This information was developed for products and services offered in the U.S.A.

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

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

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
U.S.A.

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

IBM World Trade Asia Corporation  
Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106, Japan

**The following paragraph does not apply to the United Kingdom or any other country/region where such provisions are inconsistent with local law:**

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

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

This document may provide links or references to non-IBM Web sites and resources. IBM makes no representations, warranties, or other commitments whatsoever about any non-IBM Web sites or third-party resources that may be referenced, accessible from, or linked from this document. A link to a non-IBM Web site does not mean that IBM endorses the content or use of such Web site or

## Notices

its owner. In addition, IBM is not a party to or responsible for any transactions you may enter into with third parties, even if you learn of such parties (or use a link to such parties) from an IBM site. Accordingly, you acknowledge and agree that IBM is not responsible for the availability of such external sites or resources, and is not responsible or liable for any content, services, products, or other materials on or available from those sites or resources. Any software provided by third parties is subject to the terms and conditions of the license that accompanies that software.

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

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

IBM Canada Limited  
Office of the Lab Director  
8200 Warden Avenue  
Markham, Ontario  
L6G 1C7  
CANADA

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

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

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems, and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information 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.

COPYRIGHT LICENSE:

This information may contain sample application programs, in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (*your company name*) (*year*). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. *\_enter the year or years\_*. All rights reserved.

## **Trademarks**

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

The following terms are trademarks or registered trademarks of other companies

- Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
- Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
- UNIX is a registered trademark of The Open Group in the United States and other countries.
- Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Intel trademark information
- Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.





---

# Index

## A

- ABS scalar function 257
- ABSVAL scalar function 257
- access plans
  - description 41
- ACOS scalar function
  - description 257
  - values and arguments 257
- administrative SQL routines
  - supported 210
- ADVISE\_INDEX table 703
- ADVISE\_INSTANCE table 707
- ADVISE\_MQT table 707
- ADVISE\_PARTITION table 708
- ADVISE\_TABLE table 710
- ADVISE\_WORKLOAD table 710
- aggregate functions
  - ARRAY\_AGG 238
  - COUNT 241
  - description 237
  - MIN 246
- aliases
  - alias name 47
  - chaining process 9
  - description 47
  - overview 9
  - TABLE\_NAME function 371
  - TABLE\_SCHEMA function 372
- ALL clause
  - quantified predicate 193
  - SELECT statement 447
- ALL option 485
- ambiguous reference errors 47
- AND truth table 190
- ANY clause 193
- application design
  - character conversion in SQL statements 24
  - code points for special characters 24
  - double-byte character support (DBCS) 24
- application process
  - definition 15
- applications
  - requesters 25
- arguments of COALESCE 106
- arithmetic
  - AVG function, operation of 239
  - columns, adding values (SUM) 251
  - CORRELATION function operation 240
  - COVARIANCE function operation 243
  - decimal values from numeric expressions 283
  - expressions, adding values (SUM) 251
  - finding maximum value 245
  - floating point values from numeric expressions 291, 345
  - integer values, returning from expressions 260, 309
  - operators 149
  - regression functions 247
  - returning small integer values from expressions 363
  - STDDEV function 250
  - VARIANCE function operation 252
- ARRAY element
  - specification 170

- ARRAY\_AGG function 238
- AS clause
  - in SELECT clause 447
  - ORDER BY clause 447
- ASC clause
  - SELECT statement 447
- ASCII scalar function
  - description 258
  - values and arguments 258
- ASIN scalar function
  - description 258
  - values and arguments 258
- assignments
  - basic SQL operations 92
- asterisk (\*)
  - in COUNT 241
  - in COUNT\_BIG 242
  - in select column names 447
  - in subselect column names 447
- asterisks
  - select column names 447
  - subselect column names 447
- ATAN scalar function
  - description 259
  - values and arguments 259
- ATAN2 scalar function
  - description 259
  - values and arguments 259
- ATANH scalar function
  - description 259
  - values and arguments 259
- attributes
  - attribute name 47
- authority levels
  - See privileges 10
- authorization ID 47
- authorization names
  - definition 47
  - description 47
  - restrictions governing 47
- AVG aggregate function 239

## B

- basic predicate 192
- best fit
  - function 134
  - method 142
- BETWEEN predicate 195
- BIGINT data type 70
  - sign and precision 70
- BIGINT function 260
- binary large objects (BLOBs)
  - definition 76
  - scalar functions 263
- binary string data types 76
- binding
  - data retrieval
    - role in optimizing 1
  - function semantics 134
  - method semantics 134

- bit data 73
- bit manipulation functions 261
- BITAND function 261
- BITANDNOT function 261
- BITNOT function 261
- BITOR function 261
- BITXOR function 261
- BLAST
  - supported versions 36
- BLOBs (binary large objects)
  - description 76
  - scalar functions 263
- books
  - printed
    - ordering 758
- buffer pools
  - names 47
- built-in functions
  - description 134
  - string units 73
- byte length
  - data type values 314

## C

- call level interface (CLI)
  - definition 2
- CALL statement
  - invoked from a compiled statement 749
- CARDINALITY function 263
- CASE
  - expression 163
- case sensitivity
  - in token identifiers 46
- CAST
  - specification 165
- casting
  - between data types 85
  - data type 165
  - reference types 85
  - structured type expression to a subtype 181
  - user-defined types 85
  - XML values 169
- catalog views
  - ATTRIBUTES 532
  - AUDITPOLICIES 533
  - AUDITUSE 535
  - BUFFERPOOLDBPARTITIONS 535
  - BUFFERPOOLS 536
  - CASTFUNCTIONS 536
  - CHECKS 537
  - COLAUTH 538
  - COLCHECKS 539
  - COLDIST 539, 642
  - COLGROUPCOLS 540
  - COLGROUPDIST 540, 643
  - COLGROUPDISTCOUNTS 541, 643
  - COLGROUPS 541, 644
  - COLIDENTATTRIBUTES 542
  - COLOPTIONS 542
  - COLUMNS 543, 644
  - COLUSE 547
  - CONSTDEP 548
  - CONTEXTATTRIBUTES 548
  - CONTEXTS 548
  - DATAPARTITIONEXPRESSION 549
  - DATAPARTITIONS 549
  - catalog views (*continued*)
    - DATATYPES 550
    - DBAUTH 552
    - DBPARTITIONGROUPDEF 554
    - DBPARTITIONGROUPS 554
    - description 15
    - EVENTMONITORS 555
    - EVENTS 556
    - EVENTTABLES 557
    - FULLHIERARCHIES 558
    - FUNCMAPOPTIONS 559
    - FUNCMAPPARMOPTIONS 559
    - FUNCMAPPINGS 559
    - HIERARCHIES 560
    - HISTOGRAMTEMPLATEBINS 561
    - HISTOGRAMTEMPLATES 561
    - HISTOGRAMTEMPLATEUSE 561
    - INDEXAUTH 562
    - INDEXCOLUSE 562
    - INDEXDEP 563
    - INDEXES 564, 645
    - INDEXEXPLOITRULES 569
    - INDEXEXTENSIONDEP 570
    - INDEXEXTENSIONMETHODS 571
    - INDEXEXTENSIONPARMS 571
    - INDEXEXTENSIONS 572
    - INDEXOPTIONS 572
    - INDEXXMLPATTERNS 573
    - KEYCOLUSE 573
    - NAMEMAPPINGS 574
    - NICKNAMES 574
    - overview 527, 528
    - PACKAGEAUTH 577
    - PACKAGEDEP 578
    - PACKAGES 579
    - PARTITIONMAPS 584
    - PASSTHROUGH 584
    - PREDICATESPECS 584
    - read-only 527
    - REFERENCES 585
    - ROLEAUTH 586
    - ROLES 586
    - ROUTINEAUTH 586
    - ROUTINEDEP 587
    - ROUTINEOPTIONS 588
    - ROUTINEPARMOPTIONS 589
    - ROUTINEPARMS 589
    - ROUTINES 591, 648
    - ROUTINESFEDERATED 598
    - SCHEMAAUTH 600
    - SCHEMATA 600
    - SECURITYLABELACCESS 601
    - SECURITYLABELCOMPONENTELEMENTS 601
    - SECURITYLABELCOMPONENTS 602
    - SECURITYLABELS 602
    - SECURITYPOLICIES 602
    - SECURITYPOLICYCOMPONENTRULES 603
    - SECURITYPOLICYEXEMPTIONS 604
    - SEQUENCEAUTH 605
    - SEQUENCES 605
    - SERVEROPTIONS 607
    - SERVERS 607
    - SERVICECLASSES 607
    - STATEMENTS 608
    - SURROGATEAUTHIDS 609
    - SYSDUMMY1 642
    - TABAUTH 610

- catalog views (*continued*)
  - TABCONST 611
  - TABDEP 612
  - TABDETACHEDDEP 613
  - TABLES 614, 649
  - TABLESPACES 619
  - TABOPTIONS 621
  - TBSPACEAUTH 621
  - THRESHOLDS 621
  - TRANSFORMS 623
  - TRIGDEP 624
  - TRIGGERS 624
  - TYPEMAPPINGS 626
  - updatable 527
  - USEROPTIONS 628
  - VARIABLEAUTH 629
  - VARIABLEDEP 629
  - VARIABLES 630
  - VIEWS 631
  - WORKACTIONS 632
  - WORKACTIONSETS 634
  - WORKCLASSES 634
  - WORKCLASSETS 635
  - WORKLOADAUTH 636
  - WORKLOADCONNATTR 636
  - WORKLOADS 637
  - WRAPOPTIONS 638
  - WRAPPERS 638
  - XDBMAPGRAPHS 638
  - XDBMAPSHREDTREES 638
  - XSROBJECTAUTH 639
  - XSROBJECTCOMPONENTS 639
  - XSROBJECTDEP 640
  - XSROBJECTHIERARCHIES 641
  - XSROBJECTS 641
- CEIL scalar function
  - description 264
  - values and arguments 264
- CEILING scalar function
  - description 264
  - values and arguments 264
- CHAR data type
  - description 73
- CHAR scalar function
  - description 264
- character conversion
  - coding SQL statements 24
  - rules for assignments 92
  - rules for comparison 92
  - rules for operations combining strings 109
  - rules when comparing strings 109
- character sets
  - definition 22
  - description 41
- character strings
  - assignment 92
  - BLOB string representation 263
  - comparisons 92
  - data types 73
  - double-byte character string 396
  - equality
    - collating sequence examples 32
    - definition 92
  - POSSTR scalar function 339
  - returning from host variable name 382
  - string constants 113
  - translating string syntax 382
- character strings (*continued*)
  - VARCHAR scalar function 390
  - VARGRAPHIC scalar function 396
- character subtypes 73
- CHARACTER\_LENGTH scalar function
  - description 268
- characters
  - conversion 22
  - SQL language elements 45
- CHR scalar function
  - description 270
  - values and arguments 270
- CLI (call level interface)
  - definition 2
- CLIENT USERID special register 120
- CLIENT WRKSTNNAME special register 121
- CLOBs (character large objects)
  - data type
    - description 73
  - function
    - description 270
    - values and arguments 270
- CLSCHED sample table 669
- COALESCE function 271
- code pages
  - attributes 22
  - definition 22
  - description 41
- code points
  - character conversion 22
- collating sequences
  - COLLATION\_KEY\_BIT scalar function 271
  - description 41
  - planning 41
  - string comparison rules 92
- COLLATING\_SEQUENCE server option
  - example 41
- COLLATION\_KEY\_BIT scalar function
  - description 271
- collocation
  - table 27
- column database functions
  - description 134
- column options
  - description 34
- columns
  - adding values (SUM) 251
  - ambiguous name reference errors 47
  - averaging a set of values (AVG) 239
  - BASIC predicate in matching strings 192
  - BETWEEN predicate in matching strings 195
  - column name
    - definition 47
    - qualification in COMMENT ON statement 47
    - uses 47
  - correlation 240
  - covariance 243
  - EXISTS predicate, in matching strings 195
  - finding maximum value 245
  - GROUP BY
    - use in limiting in SELECT clause 447
  - grouping column names in GROUP BY 447
  - HAVING
    - use in limiting in SELECT clause 447
  - HAVING clause
    - search names rules 447
  - IN predicate, fullselect, values returned 196

- columns (*continued*)
  - LIKE predicate, in matching strings 197
  - names
    - in ORDER BY clause 447
    - qualified conditions 47
    - unqualified conditions 47
  - naming conventions 47
  - nested table expression 47
  - null values
    - in result columns 447
  - qualified column name rules 47
  - result data 447
  - scalar fullselect 47
  - searching using WHERE clause 447
  - SELECT clause syntax diagram 447
  - standard deviation 250
  - string assignment rules 92
  - subquery 47
  - undefined name reference errors 47
  - variance 252
- combining grouping sets 447
- comments
  - host language, format 46
  - SQL, format 46
- commit
  - release of locks 15
- common table expressions
  - definition 489
  - recursive 489
  - select statement 489
- COMPARE\_DECFLOAT scalar function
  - description 273
- comparing 92
  - a value with a collection 195
  - two predicates, truth conditions 192, 203
- comparison
  - SQL operation 92
- compatibility
  - data types 92
  - rules 92
  - rules for operation types 92
- component-name
  - description 47
- composite column values 447
- CONCAT scalar function
  - description 274
  - values and arguments 274
- concatenation
  - distinct type 149
  - operators 149
- concurrent transactions 25
- condition name
  - SQL procedures 47
- consistency
  - points of 15
- constants
  - character string 113
  - decimal 113
  - floating-point 113
  - graphic string 113
  - hexadecimal 113
  - integer 113
  - SQL language element 113
  - with user-defined types 113
- constraints
  - description 4
  - Explain tables 703
- constraints (*continued*)
  - names, definition 47
- conversions
  - CHAR, returning converted datetime values 264
  - character string to timestamp 375
  - datetime to string variable 92
  - DBCS from mixed SBCS and DBCS 396
  - decimal values from numeric expressions 283
  - double-byte character string 396
  - floating point values from numeric expressions 291, 345
  - numeric, scale and precision, summary 92
  - rules
    - assignments 92
    - comparisons 92
    - operations combining strings 109
    - string comparisons 109
- correlated reference
  - in nested table expression 47
  - in scalar fullselect 47
  - in subquery 47
  - in subselect 447
- CORRELATION function 240
- correlation name
  - definition 47
  - FROM clause
    - subselect rules 447
  - in SELECT clause
    - syntax diagram 447
  - qualified reference 47
  - rules 47
- COS scalar function
  - description 274
  - values and arguments 274
- COSH scalar function
  - description 274
  - values and arguments 274
- COT scalar function
  - description 275
  - values and arguments 275
- COUNT function 241
- COUNT\_BIG function
  - detailed format description 242
  - values and arguments 242
- COVARIANCE function 243
- CREATE SERVER statement 36
- cross-tabulation rows 447
- CS (cursor stability)
  - isolation level 17
- CUBE grouping
  - examples 447
  - query description 447
- CURRENT CLIENT\_ACCTNG special register 120
- CURRENT CLIENT\_APPLNAME special register 120
- CURRENT CLIENT\_USERID special register 120
- CURRENT CLIENT\_WRKSTNNAME special register 121
- CURRENT DATE special register 121
- CURRENT DBPARTITIONNUM special register 121
- CURRENT DECFLOAT ROUNDING MODE special register 122
- CURRENT DEFAULT TRANSFORM GROUP special register 123
- CURRENT DEGREE special register
  - description 123
- CURRENT EXPLAIN MODE special register
  - description 124
- CURRENT EXPLAIN SNAPSHOT special register
  - description 125

- CURRENT FEDERATED ASYNCHRONY special register 125
- CURRENT FUNCTION PATH special register
  - description 129
- CURRENT IMPLICIT XMLPARSE OPTION special register 126
- CURRENT ISOLATION special register 126
- CURRENT LOCK TIMEOUT special register 127
- CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register 127
- CURRENT MDC ROLLOUT MODE special register 128
- CURRENT OPTIMIZATION PROFILE special register 128
- CURRENT PACKAGE PATH special register 128
- CURRENT PATH special register
  - description 129
- CURRENT QUERY OPTIMIZATION special register
  - description 129
- CURRENT REFRESH AGE special register
  - description 130
- CURRENT SCHEMA special register 130
- CURRENT SERVER special register 130
- CURRENT SQLID special register 130
- CURRENT TIME special register 131
- CURRENT TIMESTAMP special register 131
- CURRENT TIMEZONE special register 132
- CURRENT USER special register 132
- cursor name
  - definition 47
- cursor stability (CS)
  - isolation level 17

## D

- data
  - partitioning 27
- Data Definition Language (DDL)
  - description 1
  - statements
    - description 1
- data source objects
  - description 34
- data sources 31
  - description 30
  - identifying 47
  - valid server types 651
- data structures
  - packed decimal 517
- data type mappings
  - description 35
  - forward 652
  - reverse 662
- data types
  - BIGINT 70
  - binary string 76
  - BLOB 76
  - casting 85
  - CHAR 73
  - character string 73
  - CLOB 73
  - DATE 78
  - datetime 78
  - DBCLOB 76
  - DECIMAL
    - overview 70
  - DOUBLE 70
  - floating-point
    - overview 70
  - graphic string 76
  - data types (*continued*)
    - INTEGER
      - overview 70
    - numeric
      - overview 70
    - partition compatibility 112
    - promotion
      - overview 84
    - REAL 70
    - result columns 447
    - SMALLINT 70
    - SQL
      - overview 69
    - TIME 78
    - TIMESTAMP 78
    - TYPE\_ID function 386
    - TYPE\_NAME function 387
    - TYPE\_SCHEMA function 387
    - unsupported 35
    - user-defined
      - overview 81
    - VARCHAR
      - overview 73
    - VARGRAPHIC 76
    - XML
      - values 81
  - database manager
    - limits 501
    - SQL interpretation 1
  - database partition compatibility
    - overview 112
  - databases
    - creating
      - sample 669
    - distributed 1
    - erasing sample 669
    - partitioned 1
    - relational 1
  - DATALINK data type
    - unsupported 35
  - DATAPARTITIONNUM scalar function 275
  - DATE data type
    - overview 78
    - WEEK\_ISO scalar function 398
  - date data types
    - operations 158
  - DATE function 276
  - date functions
    - DAY 277
    - DAYS 279
    - MONTH 328
    - YEAR 431
  - dates
    - string representation formats 78
  - DAY scalar function 277
  - DAYNAME scalar function
    - description 278
  - DAYOFWEEK scalar function
    - description 278
  - DAYOFWEEK\_ISO scalar function
    - description 278
  - DAYOFYEAR scalar function
    - description 279
  - DAYS scalar function 279
  - DB2 for Linux, UNIX and Windows
    - default forward type mappings 652
    - default reverse type mappings 662

- DB2 for Linux, UNIX and Windows *(continued)*
  - supported versions 36
- DB2 for System i
  - default forward type mappings 652
  - default reverse type mappings 662
  - supported versions 36
- DB2 for VM and VSE
  - default forward type mappings 652
  - default reverse type mappings 662
  - supported versions 36
- DB2 for z/OS
  - supported versions 36
- DB2 for z/OS and OS/390
  - default forward type mappings 652
  - default reverse type mappings 662
- DB2 Information Center
  - languages 759
  - updating 760
  - versions 759
  - viewing in different languages 759
- db2nodes.cfg file
  - DBPARTITIONNUM function 280
- DBCLOB data type
  - description 76
- DBCLOB function
  - description 280
- DBPARTITIONNUM function
  - description 280
- DDL (Data Definition Language)
  - See Data Definition Language (DDL) 1
- DECFLOAT scalar function 282
- decimal constants 113
- decimal conversion 92
- DECIMAL data type
  - conversion
    - floating-point 92
    - precision 70
    - sign 70
- DECIMAL scalar function 283
- declarations
  - XMLNAMESPACES 411
- declustering
  - partial 27
- DECODE scalar function
  - description 286
- DECRYPT\_BIN function 287
- DECRYPT\_CHAR function 287
- decrypting information 287
- DEGREES scalar function
  - description 288
- delimiters
  - token 46
- DEPARTMENT sample table 669
- DEREF function
  - description 289
- dereference operation 170
- descriptor-name
  - syntax diagram 47
- DIFFERENCE scalar function
  - description 289
- DIGITS function 290
- DISABLE function mapping option
  - valid settings 652
- DISTINCT keyword
  - aggregate function 237
  - subselect statement 447

- distinct types
  - arithmetic operands 149
  - comparisons
    - overview 92
  - concatenation 149
  - constants 113
  - names 47
  - overview 81
- distributed database management system 29
- distributed relational databases
  - connecting to 25
- documentation
  - overview 755
  - PDF 755
  - printed 755
  - terms and conditions of use 762
- DOUBLE data type
  - CHAR scalar function 264
  - precision 70
  - sign 70
- DOUBLE scalar function 291
- double-byte character set (DBCS)
  - characters truncated during assignment 92
  - returning strings 396
- double-precision floating-point data type
  - overview 70
- durations
  - overview 158
- dynamic dispatch
  - description 142
- dynamic SQL
  - SQLDA
    - description 517

## E

- embedded SQL applications
  - Java 2
- EMPACT sample table 669
- EMPLOYEE sample table 669
- EMPPHOTO sample table 669
- EMPRESUME sample table 669
- empty strings
  - character 73
  - graphic 76
- encoding schemes
  - character conversion 22
- ENCRYPT scalar function 292
- encryption
  - ENCRYPT function 292
  - GETHINT function 297
  - XMLGROUP function 254
  - XMLROW function 418
- Entrez
  - supported versions 36
- error messages
  - SQLCA definitions 511
- ESCAPE clauses
  - LIKE predicate 197
- evaluation order
  - expressions 149
- event monitors
  - EVENT\_MON\_STATE function 293
  - names 47
  - overview 26
- Excel files
  - supported versions 36

- EXCEPT operator of fullselect 485
- exception tables
  - structure 741
- exclusive locks 17
- EXECUTE IMMEDIATE statement
  - dynamic SQL 1
- EXECUTE privilege
  - functions 134
  - methods 142
- EXECUTE statement
  - dynamic SQL 1
- EXISTS predicate 195
- EXP scalar function
  - description 294
- explain tables
  - overview 703
- EXPLAIN\_ARGUMENT table 711
- EXPLAIN\_DIAGNOSTIC table 717
- EXPLAIN\_DIAGNOSTIC\_DATA table 718
- EXPLAIN\_INSTANCE table 719
- EXPLAIN\_OBJECT table 722
- EXPLAIN\_OPERATOR table 724
- EXPLAIN\_PREDICATE table 726
- EXPLAIN\_STATEMENT table 729
- EXPLAIN\_STREAM table 731
- exposed correlation names
  - FROM clause 47
- expressions
  - CASE 163
  - description 149
  - GROUP BY clause 447
  - ORDER BY clause 447
  - ROW CHANGE 185
  - SELECT clause 447
  - structured type 181
  - subselect 447
- external functions
  - overview 134

## F

- federated database
  - wrapper modules 31
  - wrappers 31
- federated databases
  - description 31
  - system catalog 40
- federated server 30
  - description 36
- federated systems
  - overview 29
- file reference variables
  - BLOBs 47
  - CLOBs 47
  - DBCLOBs 47
- fixed-length character string 73
- fixed-length graphic string 76
- flat files
  - See also table-structured files 36
- FLOAT data type
  - sign and precision 70
- FLOAT function
  - description 294
  - values and arguments 294
- floating point
  - constant 113
  - to decimal conversion 92

- FLOOR function
  - description 295
  - values and arguments 295
- FOR FETCH ONLY clause
  - SELECT statement 489
- FOR READ ONLY clause
  - SELECT statement 489
- forward type mappings
  - default mappings 652
- FROM clause
  - correlation-name example 47
  - exposed names explained 47
  - non-exposed names explained 47
  - subselect syntax 447
  - use of correlation names 47
- fullselect
  - detailed syntax 485
  - examples 485
  - initialization 489
  - iterative 489
  - multiple operations, order of execution 485
  - ORDER BY clause 447
  - scalar 149
  - subquery role, search condition 47
  - table reference 447
- function mappings
  - mapping name 47
  - options 652
- function name 47
- function signature 134
- functions
  - aggregate
    - ARRAY\_AGG 238
    - COUNT 241
    - description 237
    - MIN 246
    - XMLAGG 252
  - arguments 209
  - bit manipulation 261
  - built-in 134
  - casting
    - CAST 165
    - XMLCAST 169
- column
  - ARRAY\_AGG 238
  - AVG 239
  - CORR 240
  - CORRELATION 240
  - COUNT 241
  - COUNT\_BIG 242
  - COVAR 243
  - COVARIANCE 243
  - description 134, 237
  - MAX 245
  - MIN 246
  - REGR\_AVGX 247
  - REGR\_AVGY 247
  - REGR\_COUNT 247
  - REGR\_ICPT 247
  - REGR\_INTERCEPT 247
  - REGR\_R2 247
  - REGR\_SLOPE 247
  - REGR\_SXX 247
  - REGR\_SXY 247
  - REGR\_SYY 247
  - regression functions 247
  - STDDEV 250

functions (continued)

column (continued)

SUM 251  
 VAR, options 252  
 VAR, results 252  
 VARIANCE, options 252  
 VARIANCE, results 252  
 XMLAGG 252

description 209

external 134

in a Unicode database 257

in expressions 209

OLAP (Online Analytical Processing) 172

overloaded 134

procedures 439

row 134

scalar

ABS 257  
 ABSVAL 257  
 ACOS 257  
 ASCII 258  
 ASIN 258  
 ATAN 259  
 ATAN2 259  
 ATANH 259  
 AVG 239  
 BIGINT 260  
 BITAND 261  
 BITANDNOT 261  
 BITNOT 261  
 BITOR 261  
 BITXOR 261  
 BLOB 263  
 CARDINALITY 263  
 CEIL 264  
 CEILING 264  
 CHAR 264  
 CHARACTER\_LENGTH 268  
 CHR 270  
 CLOB 270  
 COALESCE 271  
 COLLATION\_KEY\_BIT 271  
 COMPARE\_DECFLOAT 273  
 CONCAT 274  
 COS 274  
 COSH 274  
 COT 275  
 DATE 276  
 DAY 277  
 DAYNAME 278  
 DAYOFWEEK 278  
 DAYOFWEEK\_ISO 278  
 DAYOFYEAR 279  
 DAYS 279  
 DBCLOB 280  
 DBPARTITIONNUM 280  
 DECFLOAT 282  
 DECIMAL 283  
 DECODE 286  
 DECRYPTBIN 287  
 DECRYPTCHAR 287  
 DEGREES 288  
 Deref 289  
 description 134, 257  
 DIFFERENCE 289  
 DIGITS 290  
 DOUBLE 291

functions (continued)

scalar (continued)

DOUBLE\_PRECISION 291  
 ENCRYPT 292  
 EVENT\_MON\_STATE 293  
 EXP 294  
 FLOAT 294  
 FLOOR 295  
 GENERATE\_UNIQUE 295  
 GETHINT 297  
 GRAPHIC 297  
 GREATEST 299  
 GROUPING 244  
 HASHEDVALUE 299  
 HEX 300  
 HOUR 302  
 IDENTITY\_VAL\_LOCAL 302  
 INSERT 306  
 INTEGER 309  
 JULIAN\_DAY 310  
 LCASE 311  
 LCASE (locale sensitive) 311  
 LEAST 311  
 LEFT 312  
 LENGTH 314  
 LN 316  
 LOCATE 317  
 LOG10 320  
 LONG\_VARCHAR 320  
 LONG\_VARGRAPHIC 321  
 LOWER 321  
 LOWER (locale sensitive) 321  
 LTRIM 323  
 MAX 324  
 MAX\_CARDINALITY 324  
 MICROSECOND 325  
 MIDNIGHT\_SECONDS 325  
 MIN 326  
 MINUTE 327  
 MOD 328  
 MONTH 328  
 MONTHNAME 329  
 MULTIPLY\_ALT 329  
 NODENUMBER (see DBPARTITIONNUM) 280  
 NORMALIZE\_DECFLOAT 330  
 NULLIF 331  
 NVL 331  
 OCTET\_LENGTH 332  
 OVERLAY 332  
 PARAMETER 336  
 PARTITION (see HASHEDVALUE) 299  
 POSITION 336  
 POSSTR 339  
 POWER 341, 343  
 QUANTIZE 342  
 QUARTER 343  
 RAISE\_ERROR 344  
 RAND 345  
 REAL 345  
 REC2XML 346  
 REPEAT 350  
 REPLACE 350  
 RID 352  
 RID\_BIT 352  
 RIGHT 353  
 ROUND 356  
 RTRIM 358



functions (*continued*)

- scalar (*continued*)
  - SECLABEL 358
  - SECLABEL\_BY\_NAME 359
  - SECLABEL\_TO\_CHAR 360
  - SECOND 361
  - SIGN 362
  - SIN 363
  - SINH 363
  - SMALLINT 363
  - SOUNDEX 364
  - SPACE 365
  - SQRT 365
  - STRIP 366
  - SUBSTR 366
  - SUBSTRING 369
  - TABLE\_NAME 371
  - TABLE\_SCHEMA 372
  - TAN 373
  - TANH 374
  - TIME 374
  - TIMESTAMP 375
  - TIMESTAMP\_FORMAT 376
  - TIMESTAMP\_ISO 379
  - TIMESTAMPDIFF 379
  - TO\_CHAR 380
  - TO\_DATE 381
  - TOTALORDER 381
  - TRANSLATE 382
  - TRIM 384
  - TRUNC 385
  - TRUNCATE 385
  - TYPE\_ID 386
  - TYPE\_NAME 387
  - TYPE\_SCHEMA 387
  - UCASE 388
  - UCASE (locale sensitive) 388
  - UPPER 388
  - UPPER (locale sensitive) 388
  - VALUE 390
  - VARCHAR 390
  - VARCHAR\_FORMAT 393
  - VARGRAPHIC 396
  - WEEK 398
  - WEEK\_ISO 398
  - XMLATTRIBUTES 399
  - XMLCOMMENT 400
  - XMLCONCAT 401
  - XMLDOCUMENT 402
  - XMLELEMENT 403
  - XMLFOREST 408
  - XMLGROUP 254
  - XMLNAMESPACES 411
  - XMLPARSE 413
  - XMLPI 415
  - XMLQUERY 416
  - XMLROW 418
  - XMLSERIALIZE 420
  - XMLTEXT 422
  - XMLVALIDATE 423
  - XMLXSROBJECTID 427
  - XSLTRANSFORM 428
  - YEAR 431
- sourced 134
- SQL 134
- SQL language element 134
- supported 210

functions (*continued*)

- table
  - XMLTABLE 432
- table functions
  - description 134, 432
- user-defined 134, 436

## G

- GENERATE\_UNIQUE function
  - syntax 295
- GETHINT function
  - description 297
  - values and arguments 297
- global catalog
  - description 40
- global variables 133
- grand total row 447
- GRAPHIC data type
  - description 76
- GRAPHIC function
  - description 297
  - values and arguments 297
- GRAPHIC space 24
- graphic string constant
  - description 113
- graphic string data type
  - description 76
- graphic strings
  - returning from host variable name 382
  - translating string syntax 382
- GREATEST function 299
- GROUP BY clause
  - subselect results 447
  - subselect rules and syntax 447
- GROUPING function 244
- grouping-expression 447
- groups
  - defining names 47

## H

- hash partitioning 27
- HASHEDVALUE function 299
- HAVING clause 447
- help
  - configuring language 759
  - SQL statements 759
- HEX function 300
- hexadecimal constant
  - description 113
- HMMER data source
  - supported versions 36
- host identifiers
  - overview 47
- host variables
  - BLOB 47
  - CLOB 47
  - DBCLOB 47
  - indicator variables 47
  - overview 47
  - syntax diagrams 47
- HOURLY scalar function
  - description 302

## I

- identifiers
  - delimited 47
  - host 47
  - length limits 501
  - ordinary 47
  - SQL 47
- IDENTITY\_VAL\_LOCAL function
  - description 302
  - values and arguments 302
- IMPLICIT\_SCHEMA authority 3
- IN predicate 196
- indexes
  - description 5
  - name
    - definition 47
    - overview 5
- indicator variables
  - description 47
  - host variable, uses in declaring 47
- Informix
  - default forward type mappings 652
  - default reverse type mappings 662
  - supported versions 36
- initializing
  - fullselect 489
- INSERT function
  - description 306
  - values and arguments 306
- integer constant
  - description 113
- INTEGER data type
  - sign and precision 70
- INTEGER function
  - description 309
  - values and arguments 309
- integer values from expressions
  - INTEGER function 309
- integers
  - decimal conversion summary 92
  - in ORDER BY clause 447
- interactive SQL 1
- intermediate result tables 447
- INTERSECT operator
  - duplicate rows, use of ALL 485
  - of fullselect, role in comparison 485
- INTO clause
  - FETCH statement, use in host variable 47
  - SELECT INTO statement, use in host variable 47
  - values from applications programs 47
- INTRAY sample table 669
- invoking
  - function 134
  - methods 180
- isolation levels
  - cursor stability 17
  - description 17
  - in DELETE statement 489
  - read stability (RS) 17
  - repeatable read (RR) 17
  - uncommitted read (UR) 17
- iterative fullselect 489

## J

- Java Database Connectivity (JDBC)
  - embedded SQL for Java (SQLJ) 2
  - overview 2
- joins
  - subselect component of fullselect 447
  - tables
    - subselect clause 447
  - types
    - full outer 447
    - inner 447
    - left outer 447
    - right outer 447
- JULIAN\_DAY scalar function
  - description 310

## L

- label-based access control (LBAC)
  - exception tables 741
  - limits 501
- labels
  - durations 158
  - object names in SQL procedures 47
- large integers 70
- large objects (LOBs)
  - behavior 28
  - description 77
  - locator 77
  - with partitioned tables 28
- lateral correlation 447
- LBAC (label-based access control)
  - exception tables 741
  - limits 501
  - security labels
    - component name length 501
    - name length 501
  - security policies
    - name length 501
- LCASE (Locale sensitive) scalar function
  - overview 311
- LEAST function 311
- LEFT scalar function
  - description 312
  - values and arguments 312
- length
  - LENGTH scalar function 314
- LENGTH scalar function
  - description 314
  - values and arguments 314
- LIKE predicate 197
- limits
  - identifier length 501
  - SQL 501
- literals
  - description 113
- LN function
  - description 316
  - values and arguments 316
- LOB locators 77
- LOBs (large objects)
  - description 77
- local catalog
  - See global catalog 40
- LOCATE scalar function
  - description 317

- LOCATE scalar function (*continued*)
  - values and arguments 317
- locators
  - large object (LOB) 77
  - variable description 47
- locks
  - definition 15
  - exclusive (X) 17
  - share (S) 17
  - update (U) 17
- LOG10 scalar function
  - description 320
  - values and arguments 320
- logical operators
  - search rules 190
- LONG\_VARCHAR function
  - description 320
- LONG\_VARGRAPHIC function
  - description 321
- LOWER (Locale sensitive) scalar function
  - description 321
  - values and arguments 321
- LOWER scalar function
  - description 321
  - values and arguments 321
- LTRIM scalar function
  - description 323
  - values and arguments 323

**M**

- MAX function 324
  - detailed format description 245
  - values and arguments 245
- MAX\_CARDINALITY function 324
- method name 47
- method signature 142
- methods
  - built-in 142
  - dynamic dispatch of 142
  - external 142
  - invoking 180
  - overloaded 142
  - SQL 142
  - SQL language element 142
  - type preserving 142
  - user-defined 142
- MICROSECOND function
  - description 325
  - values and arguments 325
- Microsoft Excel
  - See Excel files 36
- Microsoft SQL Server
  - default forward type mappings 652
  - default reverse type mappings 662
  - supported versions 36
- MIDNIGHT\_SECONDS function
  - description 325
  - values and arguments 325
- MIN function 246, 326
- MINUTE scalar function
  - description 327
  - values and arguments 327
- mixed data
  - definition 73
  - LIKE predicate 197

- MOD function
  - description 328
  - values and arguments 328
- monitoring
  - database events 26
- MONTH function
  - description 328
  - values and arguments 328
- MONTHNAME function
  - description 329
  - values and arguments 329
- multibyte character support
  - code points for special characters 24
- multiple row VALUES clause
  - result data type 106
- MULTIPLY\_ALT function
  - detailed format description 329
  - values and arguments, rules for 329

## N

- names
  - subselect columns 447
- naming conventions
  - identifiers 47
  - qualified column rules 47
- nested table expressions 447
- nickname column options
  - description 34
- nicknames
  - definition 47
  - description 34
- FROM clause
  - exposed names 47
  - nonexposed names 47
  - subselect 447
  - qualifying a column name 47
  - SELECT clause 447
- NODENUMBER function
  - DBPARTITIONNUM 280
- nonexposed correlation-name in FROM clause 47
- nonrelational data sources
  - specifying data type mappings 35
- NORMALIZE\_DECFLOAT scalar function
  - description 330
- NOT NULL clause
  - in NULL predicate 202
- notices 765
- NUL-terminated character strings 73
- NULL predicate rules 202
- NULL value
  - SQL
    - assignment 92
    - definition 69
    - grouping-expressions, allowable uses 447
    - occurrences in duplicate rows 447
    - result columns 447
    - specified by indicator variable 47
    - unknown condition 190
- NULLIF function
  - description 331
  - values and arguments 331
- numbers
  - precision 517
  - scale 517
- numeric
  - assignments in SQL operations 92

- numeric (*continued*)
  - comparisons 92
  - data types 70
- NUMERIC or DECIMAL data type
  - sign and precision 70
- NVL scalar function
  - overview 331

**O**

- object table 47
- OCTET\_LENGTH scalar function
  - description 332
- ODBC
  - default forward type mappings 652
  - supported versions 36
- ODBC (open database connectivity)
  - description 2
- OLAP
  - specification 172
- OLAP (Online Analytical Processing)
  - functions 172
- OLE DB
  - supported versions 36
- Online Analytical Processing (OLAP)
  - functions 172
- open database connectivity (ODBC) 2
- operands
  - decimal 149
  - floating-point 149
  - integer 149
  - result data type 106
  - strings 149
- operations
  - assignments 92
  - comparisons 92
  - datetime 158
  - dereference 170
- operators
  - arithmetic 149
- optimizer
  - description 41
- OR truth table 190
- Oracle
  - default forward type mappings 652
  - default reverse type mappings 662
- ORDER BY clause
  - culturally correct collation 271
  - select statement 447
- order of evaluation
  - expressions 149
- ordering DB2 books 758
- ordinary tokens 46
- ORG sample table 669
- outer joins
  - joined table 447
- OVERLAY scalar function
  - description 332
- overloaded function
  - multiple function instances 134
- overloaded method 142
- ownership
  - database objects 10

**P**

- packages
  - authorization IDs
    - binding 47
    - dynamic statements 47
  - names
    - overview 47
    - overview 9
- PARAMETER function
  - description 336
- parameter markers
  - dynamic SQL
    - host variables 47
- parameters
  - naming conventions 47
- PARTITION function
  - substitution for HASHEDVALUE name 299
- partitioned database environments
  - overview 27
  - partition compatibility 112
- partitioned tables
  - large objects (LOBs) 28
- path
  - SQL 134
- pattern matching
  - Unicode databases 111
- phantom row 17
- point of consistency
  - database 15
- POSITION scalar function 336
- POSSTR function 339
- POWER scalar function
  - description 341
- precedence
  - overview 149
- precision
  - numbers
    - SQLLEN variable 517
- precision-integer DECIMAL function 283
- predicates
  - basic 192
  - BETWEEN 195
  - EXISTS 195
  - IN 196
  - LIKE 197
  - NULL 202
  - overview 186
  - quantified 193
  - query processing 187
  - TYPE 203
  - VALIDATED 204
  - XMLEXISTS 206
- PREPARE statement
  - dynamic SQL 1
- privileges
  - EXECUTE
    - functions 134
    - methods 142
  - hierarchy 10
  - implicit for packages 10
  - individual 10
  - overview 10
  - ownership (CONTROL) 10
- problem determination
  - information available 762
  - tutorials 762

- procedures
  - names
    - overview 47
- PROJECT sample table 669
- pushdown analysis
  - description 41

## Q

- qualified column names 47
- qualifiers
  - object name 47
  - reserved 697
- quantified predicate 193
- QUANTIZE scalar function
  - description 342
- QUARTER function
  - description 343
  - values and arguments 343
- queries
  - authorization IDs 447
  - definition 447
  - description 2
  - examples
    - SELECT statement 489
  - fragments 41
  - recursive 489
- query optimization
  - description 41

## R

- RADIANS scalar function
  - description 343
  - values and arguments 343
- RAISE\_ERROR scalar function
  - description 344
  - values and arguments 344
- RAND scalar function
  - description 345
  - values and arguments 345
- read stability (RS)
  - description 17
- REAL function
  - description 345
  - single precision conversion 345
  - values and arguments 345
- REAL SQL data type
  - sign and precision 70
- REC2XML scalar function
  - description 346
  - values and arguments 346
- recursion queries 489
- recursive common table expression 489
- reference types
  - casting 85
  - comparisons 92
  - DEREF function 289
  - description 81
- regression functions
  - description 247
  - REGR\_AVGX 247
  - REGR\_AVGY 247
  - REGR\_COUNT 247
  - REGR\_ICPT 247
  - REGR\_INTERCEPT 247

- regression functions (*continued*)
  - REGR\_R2 247
  - REGR\_SLOPE 247
  - REGR\_SXX 247
  - REGR\_SXY 247
  - REGR\_SYY 247
- remote
  - function name 47
  - type name 47
- remote authorization name 47
- remote catalog information 40
- REMOTE\_NAME function mapping option
  - valid settings 652
- remote-object-name 47
- remote-schema-name 47
- remote-table-name 47
- REPEAT scalar function
  - description 350
  - values and arguments 350
- repeatable read (RR)
  - description 17
- REPLACE scalar function
  - description 350
  - values and arguments 350
- reserved
  - qualifiers 697
  - schemas 697
  - words 697
- resolution
  - function 134
  - method 142
- result columns
  - subselect 447
- result data type
  - arguments of COALESCE 106
  - multiple row VALUES clause 106
  - operands 106
  - result expressions of CASE 106
  - set operator 106
- result expressions of CASE
  - result data type 106
- result tables
  - query 447
- return identity column value
  - IDENTITY\_VAL\_LOCAL function 302
- returning hour part of values
  - HOUR function 302
- returning microsecond from value
  - MICROSECOND function 325
- returning minute from value
  - MINUTE function 327
- returning month from value
  - MONTH function 328
- returning seconds from value
  - SECOND function 361
- returning substrings from a string
  - SUBSTR function 366
- returning timestamp from values
  - TIMESTAMP function 375
- reverse type mappings
  - default mappings 662
- RID function 352
- RID\_BIT function 352
- RIGHT scalar function
  - description 353
  - values and arguments 353

- rollback
  - definition 15
- ROLLUP grouping of GROUP BY clause 447
- ROUND scalar function
  - description 356
  - values and arguments 356
- routines
  - procedures 439
  - SQL administrative
    - supported 210
  - SQL statements allowed 745
- ROW CHANGE
  - expression 185
- row functions
  - description 134
- rows
  - COUNT\_BIG function 242
  - GROUP BY clause 447
  - HAVING clause 447
  - search conditions, syntax 190
  - SELECT clause
    - syntax diagram 447
- RR (repeatable read) isolation level
  - description 17
- RS (read stability) isolation level
  - description 17
- RTRIM scalar function
  - description 358
- run-time authorization ID 47

## S

- SALES sample table 669
- SAMPLE database
  - creating 669
  - description 669
  - erasing 669
- sampling
  - subselect tablesample-clauses 447
- savepoints
  - names 47
- SBCS (single-byte character set) data
  - overview 73
- scalar fullselect expressions 149
- scalar functions
  - DECIMAL function 283
  - regular functions 134
  - scalar functions 257
  - VARCHAR\_BIT\_FORMAT 392
  - VARCHAR\_FORMAT\_BIT 395
- scale
  - data
    - comparisons in SQL 92
    - determined by SQLLEN variable 517
    - number conversion in SQL 92
  - numbers
    - determined by SQLLEN variable 517
- schemas
  - description 3
  - names 47
  - reserved 697
- scope
  - definition 81
- Script
  - supported versions 36
- search conditions
  - AND logical operator 190

- search conditions (*continued*)
  - description 190
  - HAVING clause
    - arguments and rules 447
  - NOT logical operator 190
  - OR logical operator 190
  - order of evaluation 190
  - WHERE clause 447
- SECADM
  - database authority 10
- SECLABEL
  - scalar function 358
- SECLABEL\_BY\_NAME scalar function
  - description 359
- SECLABEL\_TO\_CHAR scalar function
  - description 360
- SECOND scalar function
  - description 361
  - values and arguments 361
- sections
  - definition 9
- security administrator authority (SECADM) 10
- security labels (LBAC)
  - component name length 501
  - name length 501
  - policies
    - name length 501
- security-label-name 47
- security-policy-name 47
- SELECT clause
  - list notation
    - column reference 447
    - with DISTINCT keyword 447
- select list
  - application rules and syntax 447
  - description 447
  - notation rules and conventions 447
- SELECT statement
  - definition 489
  - examples 489
  - fullselect detailed syntax 485
  - subselects 447
  - VALUES clause 485
- sequences
  - ordering 295
  - values 182
- server definitions
  - description 32
- server options
  - description 32
  - temporary 32
- server types
  - valid federated types 651
- servers
  - names 47
- SESSION USER special register 132
- set operators
  - EXCEPT, comparing differences 485
  - INTERSECT, role of AND in comparisons 485
  - result data type 106
  - UNION, correspondence to OR 485
- SET SERVER OPTION statement
  - setting an option temporarily 32
- share locks 17
- shift-in characters
  - not truncated by assignments 92

- SIGN scalar function
  - description 362
  - values and arguments 362
- signatures
  - function 134
  - method 142
- SIN scalar function
  - description 363
  - values and arguments 363
- single-precision floating-point data type 70
- SINH scalar function
  - description 363
  - values and arguments 363
- size limits
  - identifier length 501
  - SQL 501
- small integer values from expressions
  - SMALLINT function 363
- small integers
  - See SMALLINT data type 70
- SMALLINT data type
  - sign and precision 70
- SMALLINT function
  - description 363
  - values and arguments 363
- SOME quantified predicate 193
- sorting 41
  - ordering of results 92
  - string comparisons 92
- SOUNDEX scalar function
  - description 364
  - values and arguments 364
- sourced functions 134
- space
  - rules governing 46
- SPACE scalar function
  - description 365
  - values and arguments 365
- special registers
  - CLIENT ACCTNG 120
  - CLIENT APPLNAME 120
  - CURRENT CLIENT\_ACCTNG 120
  - CURRENT CLIENT\_APPLNAME 120
  - CURRENT CLIENT\_USERID 120
  - CURRENT CLIENT\_WRKSTNNAME 121
  - CURRENT DATE 121
  - CURRENT DBPARTITIONNUM 121
  - CURRENT DECFLOAT ROUNDING MODE 122
  - CURRENT DEFAULT TRANSFORM GROUP 123
  - CURRENT DEGREE 123
  - CURRENT EXPLAIN MODE 124
  - CURRENT EXPLAIN SNAPSHOT 125
  - CURRENT FEDERATED ASYNCHRONY 125
  - CURRENT FUNCTION PATH 129
  - CURRENT IMPLICIT XMLPARSE OPTION 126
  - CURRENT ISOLATION 126
  - CURRENT LOCK TIMEOUT 127
  - CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION 127
  - CURRENT MDC ROLLOUT MODE 128
  - CURRENT NODE (see CURRENT DBPARTITIONNUM) 121
  - CURRENT OPTIMIZATION PROFILE 128
  - CURRENT PACKAGE PATH 128
  - CURRENT PATH 129
  - CURRENT QUERY OPTIMIZATION 129
  - CURRENT REFRESH AGE 130
- special registers (*continued*)
  - CURRENT SCHEMA 130
  - CURRENT SERVER 130
  - CURRENT SQLID 130
  - CURRENT TIME 131
  - CURRENT TIMESTAMP 131
  - CURRENT TIMEZONE 132
  - CURRENT USER 132
  - interaction, Explain 735
  - SESSION USER 132
  - SQL language element 117
  - SYSTEM USER 133
  - updatable 117
  - USER 133
- specific name
  - definition 47
- specifications
  - ARRAY element 170
  - CAST 165
  - OLAP 172
  - XMLCAST 169
- SQL (Structured Query Language)
  - limits 501
  - path 134
- SQL compiler
  - in a federated system 31
- SQL functions 134
- SQL operations
  - basic 92
- SQL path
  - built-in 134
- SQL statements
  - allowed in routines 745
  - CALL 749
  - displaying help 759
  - dynamic SQL
    - definition 1
    - immediate execution 1
  - interactive SQL
    - definition 1
  - preparing and executing dynamic SQL 1
  - static
    - definition 1
- SQL subquery
  - WHERE clause 447
- SQL syntax
  - AVG aggregate function, results on column set 239
  - basic predicate, detailed diagram 192
  - BETWEEN predicate, rules 195
  - comparing two predicates, truth conditions 192, 203
  - CORRELATION aggregate function results 240
  - COUNT\_BIG function, arguments and results 242
  - COVARIANCE aggregate function results 243
  - EXISTS predicate 195
  - GENERATE\_UNIQUE function 295
  - GROUP BY clause
    - subselect 447
  - IN predicate description 196
  - LIKE predicate, rules 197
  - multiple operations, order of execution 485
  - regression functions results 247
  - search conditions, detailed formats and rules 190
  - SELECT clause description 447
  - STDDEV aggregate function, results 250
  - TYPE predicate 203
  - VARIANCE aggregate function results 252
  - WHERE clause search conditions 447

- SQL variable name 47
- SQLCA (SQL communication area)
  - description 511
  - error reporting 511
  - partitioned database systems 511
  - viewing interactively 511
- SQLD field in SQLDA 517
- SQLDA (SQL descriptor area)
  - contents 517
- SQLDABC field in SQLDA 517
- SQLDAID field in SQLDA 517
- SQLDATA field in SQLDA 517
- SQLDATALEN field in SQLDA 517
- SQLDATATYPE\_NAME field in SQLDA 517
- SQLIND field in SQLDA 517
- SQLJ
  - connectivity 2
- SQLLEN field in SQLDA 517
- SQLLONGLEN field in SQLDA 517
- SQLN field in SQLDA 517
- SQLNAME field in SQLDA 517
- SQLSTATE
  - in RAISE\_ERROR function 344
- SQLTYPE field in SQLDA 517
- SQLVAR field in SQLDA 517
- SQRT scalar function
  - description 365
- STAFF sample table 669
- STAFFG sample table 669
- statements
  - names 47
- static SQL
  - description 1
- STDDEV function 250
- stored procedures
  - CALL statement 749
  - XSR\_ADDSCHEMADOC 439
  - XSR\_COMPLETE 440
  - XSR\_DTD 441
  - XSR\_ENTITY 442
  - XSR\_REGISTER 444
  - XSR\_UPDATE 445
- string units
  - built-in functions 73
- strings
  - assignment conversion rules 92
  - collating sequences 41
  - definition 22
  - Unicode comparisons 111
- STRIP scalar function
  - description 366
- Structured Query Language (SQL)
  - assignments 92
  - basic operands, assignments and comparisons 92
  - comparison operation, overview 92
- structured types
  - description 81
  - expression
    - casting to a subtype 181
    - host variables 47
- sub-total rows 447
- subqueries
  - HAVING clause 447
  - using fullselect as search condition 47
  - WHERE clause 447
- subselect
  - description 447
- subselect (*continued*)
  - example sequence of operations 447
  - examples 447
  - FROM clause
    - subselect 447
- SUBSTR function
  - fragments 366
- SUBSTR scalar function
  - description 366
  - values and arguments 366
- SUBSTRING scalar function
  - description 369
- substrings 366
- SUM function
  - detailed format description 251
  - values and arguments 251
- super-aggregate rows 447
- super-groups 447
- supertypes
  - identifier names 47
- supported functions 210
- Sybase
  - default forward type mappings 652
  - default reverse type mappings 662
  - supported versions 36
- symmetric super-aggregate rows 447
- synonyms
  - qualifying a column name 47
- syntax
  - description x
- system catalog views
  - description 15
- system catalogs
  - views on system tables 527
- SYSTEM USER special register 133

## T

- TABLE clause
  - table reference 447
- table functions
  - description 134, 432
- table reference
  - alias 447
  - nested table expressions 447
  - nickname 447
  - table name 447
  - view name 447
- table spaces
  - description 20
  - name 47
- TABLE\_NAME function
  - alias 371
  - description 371
  - values and arguments 371
- TABLE\_SCHEMA function
  - alias 372
  - description 372
  - values and arguments 372
- table-structured files
  - supported versions 36
- tables
  - aliases 9
  - catalog views on system tables 527
  - collocation 27
  - correlation name 47
  - description 4



- tables (*continued*)
  - designator to avoid ambiguity 47
  - exception 741
  - exposed names in FROM clause 47
  - expressions
    - common 2
    - common table expressions 489
    - description 2
  - FROM clause
    - subselect naming conventions 447
  - names
    - description 47
    - in FROM clause 447
    - in SELECT clause, syntax diagram 447
  - nested table expression 47
  - non-exposed names in FROM clause 47
  - qualified column name 47
  - reference 447
  - SAMPLE database 669
  - scalar fullselect 47
  - subquery 47
  - unique correlation names 47
- TAN scalar function
  - description 373
  - values and arguments 373
- TANH scalar function
  - description 374
  - values and arguments 374
- Teradata
  - default forward type mappings 652
  - default reverse type mappings 662
- terms and conditions
  - use of publications 762
- time
  - CHAR, use in format conversion 264
  - hour values, using in an expression (HOUR) 302
  - in expressions, TIME function 374
  - returning
    - microseconds, from datetime value 325
    - minutes, from datetime value 327
    - seconds, from datetime value 361
    - timestamp from values 375
    - values based on time 374
  - string representation formats 78
  - using time in expressions 374
- TIME data types
  - description 78
  - operations 158
- TIME functions
  - description 374
  - values and arguments 374
- TIMESTAMP data type
  - description 78
  - WEEK scalar function 398
  - WEEK\_ISO scalar function 398
- TIMESTAMP function
  - description 375
  - values and arguments 375
- TIMESTAMP\_FORMAT function
  - description 376
  - values and arguments 376
- TIMESTAMP\_ISO function
  - description 379
  - values and arguments 379
- TIMESTAMPDIFF scalar function
  - description 379
  - values and arguments 379
- timestamps
  - GENERATE\_UNIQUE 295
  - string representation formats 78
- TO\_CHAR function
  - description 380
  - values and arguments 380
- TO\_DATE function
  - description 381
  - values and arguments 381
- tokens
  - case sensitivity 46
  - delimiter 46
  - ordinary 46
  - SQL language element 46
- TOTALORDER scalar function
  - description 381
- TRANSLATE scalar function
  - character string 382
  - description 382
  - graphic string 382
  - values and arguments 382
- triggers
  - cascading 6
  - constraints, interaction 701
  - description 6
  - Explain tables 703
  - interactions 701
  - maximum name length 501
  - names 47
- TRIM scalar function
  - description 384
- troubleshooting
  - online information 762
  - tutorials 762
- TRUNCATE scalar function 385
- truncation
  - numbers 92
- truth tables 190
- truth valued logic 190
- tutorials
  - problem determination 762
  - troubleshooting 762
  - Visual Explain 762
- type mapping
  - name 47
- type name 47
- TYPE predicate
  - format 203
- type preserving method 142
- TYPE\_ID function
  - data types 386
  - description 386
  - values and arguments 386
- TYPE\_NAME function
  - description 387
  - values and arguments 387
- TYPE\_SCHEMA function
  - data types 387
  - description 387
  - values and arguments 387
- typed tables
  - names 47
- typed views
  - description 8
  - names 47

- types
  - distinct
    - user-defined 81
  - reference 81
  - structured 81

## U

- UCASE (Locale sensitive) scalar function 388
- UCASE scalar function 388
- UDFs
  - see user-defined functions 436
- unary operators
  - minus sign 149
  - plus sign 149
- uncommitted reads (UR)
  - isolation levels 17
- undefined reference errors 47
- Unicode
  - conversion to uppercase 46
- Unicode (UCS-2)
  - functions in 257
  - pattern matching 111
  - string comparisons 111
- UNION operator
  - role in comparison of fullselect 485
- unique correlation names
  - table designators 47
- units of work (UOW)
  - definition 15
- unknown condition
  - null value 190
- update lock 17
- updates
  - DB2 Information Center 760
  - updatable special registers 117
- UPPER (Locale sensitive) scalar function
  - description 388
  - values and arguments 388
- UPPER scalar function
  - description 388
  - values and arguments 388
- UR (uncommitted read) isolation level 17
- user mappings
  - description 33
  - storing 33
- USER special register 133
- user-defined functions
  - description 134, 209, 436
- user-defined methods
  - description 142
- user-defined types
  - casting 85
  - description 81
  - distinct types
    - description 81
  - reference type 81
  - structured types 81
- user-defined types (UDTs)
  - unsupported data types 35

## V

- VALIDATED predicate
  - description 204

- VALUE function
  - description 390
  - values and arguments 390
- values
  - definition 69
  - null 69
  - sequence 182
- VALUES clause
  - fullselect 485
- VARCHAR data type
  - description 73
  - DOUBLE scalar function 291
  - WEEK scalar function 398
  - WEEK\_ISO scalar function 398
- VARCHAR function
  - description 390
  - values and arguments 390
- VARCHAR\_BIT\_FORMAT function
  - description 392
- VARCHAR\_FORMAT function
  - description 393
  - values and arguments 393
- VARCHAR\_FORMAT\_BIT function
  - description 395
- VARGRAPHIC data type
  - description 76
- VARGRAPHIC function
  - description 396
  - values and arguments 396
- variables
  - global 133
- VARIANCE aggregate function 252
- varying-length character string 73
- varying-length graphic string 76
- view
  - aliases 9
- view name
  - definition 47
- views
  - description 8
  - exposed names in FROM clause 47
  - FROM clause
    - subselect naming conventions 447
  - names in FROM clause 447
  - names in SELECT clause
    - syntax diagram 447
  - non-exposed names in FROM clause 47
  - overview 8
  - qualifying a column name 47
- Visual Explain
  - tutorial 762

## W

- WEEK scalar function
  - description 398
- WEEK\_ISO scalar function
  - description 398
- WHERE clause
  - subselect component of fullselect 447
- wild cards
  - LIKE predicate 197
- WITH common table expression
  - select-statement 489
- words
  - SQL reserved 697

wrappers  
  description 31  
  names 47

## X

XML  
  data type 81  
  supported versions 36  
XMLAGG aggregate function  
  description 252  
XMLATTRIBUTES scalar function  
  description 399  
XMLCAST specification  
  description 169  
XMLCOMMENT scalar function  
  description 400  
XMLCONCAT scalar function  
  description 401  
XMLDOCUMENT scalar function  
  description 402  
XMLELEMENT scalar function  
  description 403  
XML EXISTS predicate 206  
XMLFOREST scalar function  
  description 408  
XMLGROUP scalar function 254  
XMLNAMESPACES declaration  
  description 411  
XMLPARSE scalar function  
  description 413  
XMLPI scalar function  
  description 415  
XMLQUERY scalar function  
  description 416  
XMLROW scalar function 418  
XMLSERIALIZE scalar function  
  description 420  
XMLTABLE table function  
  description 432  
XMLTEXT scalar function  
  description 422  
XMLVALIDATE scalar function  
  description 423  
XMLXSROBJECTID scalar function  
  description 427  
XSLTRANSFORM scalar function  
  description 428  
XSR\_ADDSCHEMADOC stored procedure 439  
XSR\_COMPLETE stored procedure 440  
XSR\_DTD stored procedure 441  
XSR\_ENTITY stored procedure 442  
XSR\_REGISTER stored procedure 444  
XSR\_UPDATE stored procedure 445

## Y

YEAR scalar function  
  description 431  
  values and arguments 431







Printed in USA

SC23-5861-02



Spine information:

DB2 Version 9.5 for Linux, UNIX, and Windows

SQL Reference, Volume 1

